Politecnico Di Torino

MASTER DEGREE IN MECHATRONIC ENGINEERING 2021/2022

Robotics

Hexapod project

Group members
Luca Zaccaria - s301250
Davide Morazzo - s301239
Riccardo De Cesare - s299955
Federico Piovesan - s302998
Francesco Contran - s301330

Introduction

Hexapod robots are a relatively new type of robot structure developed mainly with the general purpose of exploration of low-structured environments. These robots are classified in the category of bio-inspired robots, taking as reference the locomotion system of arachnids (e.g.: spiders). Their movement characteristics are well suited for exploration of unknown or partially known environments due to the high adaptivity and flexibility, indeed they can move virtually on every type of terrain. We thought as possible real-world application of our prototype the general context of disaster relief. In this situations there is normally a great lack of information on the rescue workers' side that can be filled by a flexible, small sized, autonomous robot. In order to avoid to start from scratch with the mechanical design and focus on the algorithmic part of the prototype we chose a low-cost yet complete solution, manufactured and distributed by Adeept.

Figure 1: Hexapod robot by Adeept

This robot is a 12 DOF hexapod (2 servomotors for each leg) and comes equipped with complete mechanical components, a modified Arduino Uno board to support all the electronic communications, an ultrasonic sensor, a 6-axis IMU and a Wi-fi card in order to enable wireless communication.

Components used

Arduino board

The used board is a Arduino Uno clone that came with the Hexapod kit from Adeept. In particular the microprocessor is the same as Arduino Uno so this let us use the Arduino IDE. The board has connectors added to more easily connect the servomotors and sensors. Another important feature is the redesign of the power circuit that uses a LiPo battery in order to feed power to the motors, since the USB can't deliver the power needed.

MPU6050 IMU

As IMU (inertail measurement unit) we adopted the MPU-6050. It is a 6-axis motion tracking device designed for the low power, low cost, and quite good performance requirements needed in common balance and control issues.
MPU6050 is a Micro Electro-mechanical system (MEMS), it consists of three-axis accelerometer and three-axis gyroscope. It helps us to measure velocity, orientation, acceleration, displacement and other motion-like characteristics.
For our purposes the accelerometer was not used as we focused just in the measuring of rotations about x axis. The process behind this result starts with the rotation about the desired axis (x) producing a certain vibration that through Coriolis effect it is detected by the MEMS. Then a 16 bits ADC is used to digitalize the voltage to sample each axis. In our case the sensor then interacts with an Arduino Uno board through an I2C module.

Ultrasonic sensor HC-SR04

The HC-SR04 is a simple ultrasonic sensor. Two pins are used to give power supply and ground to the sensor, the other two pins are called Trigger pin and Echo Pin. The 2 signal pins work as following: the trigger pin is the output, the pin who sends out the ultrasonic wave, and the echo pin receives the echo of this wave and is an input. When activated, the trigger pin sends out an ultrasonic wave and after a short period of time has to be turned down. The wave travels in the air and if it finds any obstacle bounces back. If it bounces back on the sensor the echo pin is activated and produces a signal proportional to the distance travelled.

State machine

In order to deal with a complex, multitasking robot we decided to use a state machine approach, in order to divide the different tasks and try to increase the complexity gradually. Once turned on the robot must always be in one of the possible states and can change state only with an external input. In order to interact with the Hexapod we designed a simple user interface via command line on Matlab.

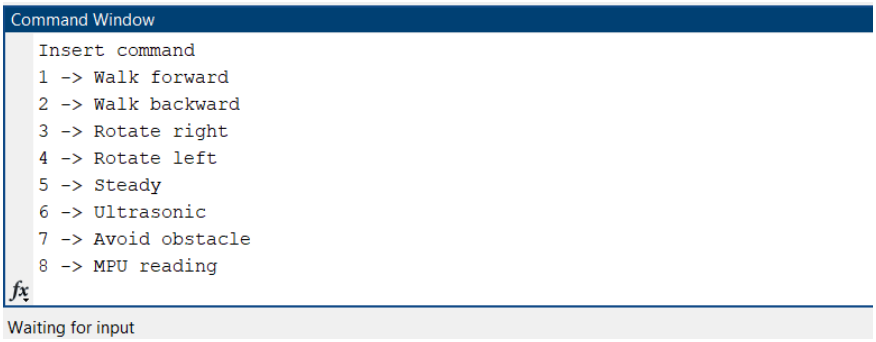

Figure 2: User interface

The possible states are the following:

- Move or rotate: forward, backward, left, right
- Steady: situation with all the legs blocked in the stable position with knees at 90°
- Ultrasonic: hexapod stops the locomotion and starts the ultrasonic reading, coupled with a motion of the head motor
- Avoid obstacle: free non-deterministic movement of the hexapod that tries to explore the environment, avoiding obstacles
- MPU reading: hexapod tries to stabilize with respect to the floor, that can be with a (small) slope
- Wait for input: Hexapod stops, waiting for new inputs

Communication

The main program that run the hexapod has been developed on MATLAB. We connected the MATLAB environment to Arduino using the serial communication: we did not use the official Arduino Support Package but we implemented our own functions from scratch. We did this because we noticed the official library in our application was far too slow to transfer the commands to Arduino, resulting in a almost unusable communication speed.

The MATLAB script can perform actions on the hexapod and read data from the sensor using the following functions:

- Legs servomotors positioning
- Head servomotor positioning
- Read distance from ultrasonic
- Read MPU gyro angles

We also had to write a firmware for the Arduino board, so the commands sent via the serial port can be received and performed correctly. For example to move the head servomotor: first an integer representing the command id is sent followed by the angle requested (all of the data are encoded in binary values), the same is for reading the sensors but this time the board sends back to the computer the data on the serial port.

Modelling and simulation

In order to follow a structured approach we decided to adopt a model-based design. First of all we decided to use the Robotics Toolbox by Peter Corke in order to model one leg of the Hexapod. To achieve a good model we used a Vernier caliper to measure the structural dimension of the leg. Following the Denavit-Hartenberg convention we defined all the parameters needed in the model. In the model all motor angles and positive rotation direction were carefully matched, in order to have highly reliable movement patterns reproduction and avoid to match the angles with a posteriori reasoning. For the hexapod foot trajectory generation we mainly referred to [2] in particular their concept of support and transfer phases and the definition of motion direction. For the gait sequence we used a classical triangle gate, in particular we referred to [3]. The triangle gait defines the synchronization of the 6 legs support phases. Three legs (2 at the extremes of a robot side and the

central one of the opposite side) are always in contact with the floor, thus defining a support polygon. This guarantees that the center of mass of the robot is always inside the support polygon.

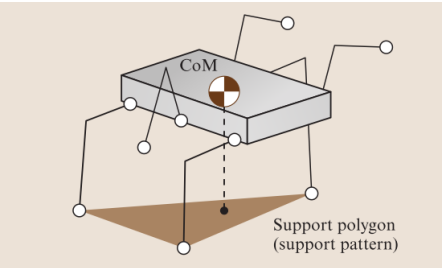

Figure 3: Triangle gait polygon

For our prototype we used the phase concept as defined by the authors but for convenience divided the possible trajectories in 4 phases. We decided to divide further the motion in order to keep a continuity in movement and account for all the possible movement necessities. Specifically we defined the following movement phases:

- Positioning: trajectory needed to position the end-effector of our leg in the first point of the support trajectory
- Support: sliding trajectory of the end-effector on the floor
- Return: trajectory from the last point of the support trajectory to the first point of the next support phase
- Stabilize: trajectory needed at the end of the locomotion cycle in order to reach a stable position of the hexapod

The above mentioned phases are conceptually different, indeed while the Positioning and Stabilize ones are executed at the start and at the end of the locomotion the Support and Return phases can be inserted in a loop in order to keep a continuity in movement. Moreover the only trajectory that required an inverse kinematics computation is the Support one: we divided the trajectory in N points (enough to have a good discretization and a smooth behaviour) and performed the inversion for each point to find joint variables' values. Likewise we did for the other three trajectories, with the only difference that the latter ones didn't require the inverse kinematics since point-to-point trajectory is needed. In this case trajectory are generated in joint space with the toolbox function *jtraj* that creates a fifth order polynomial in q. To allow the robot to walk we impose for each leg a linear support

phase all parallel to each other, in particular changing the angle defined between forward direction and movement direction let the robot translate in any direction (without rotating about its center of mass).

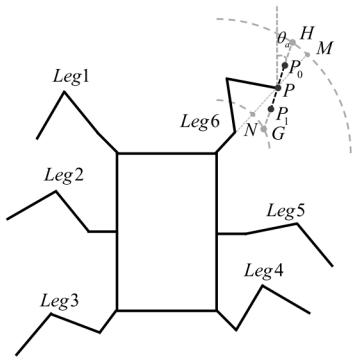

Figure 4: Hexapod leg direction

With this approach we obtain every possible movement, for example purely lateral movement (corresponding to $\theta = 90°$) and backward movement (corresponding to $\theta = 180°$). In order to have a pure rotation about the Hexapod center of mass we defined a virtual circumference passing through the end effectors' TCP of the supporting legs (three at each time) and for each leg defined a support trajectory tangential to the above mentioned circumference.

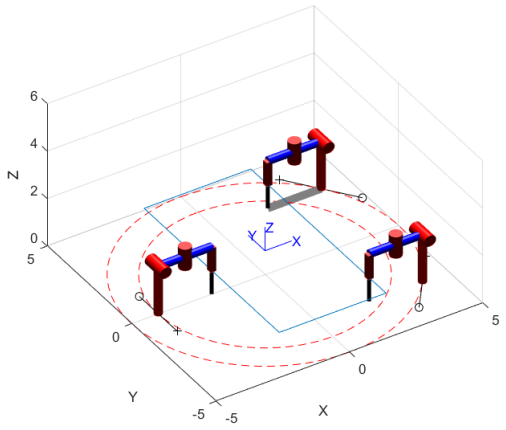

Figure 5: Turning trajetories

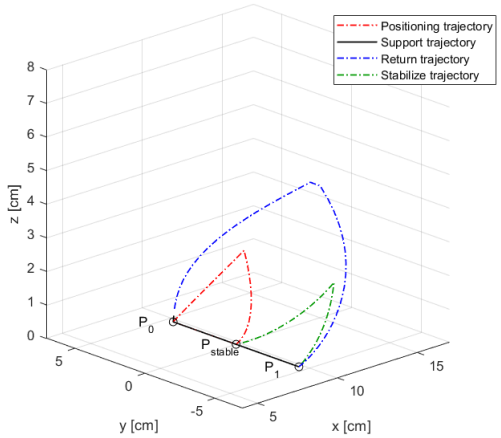

Figure 6: Diagram of the trajectories

Data structures and main functions

In order to give an idea of the working principle of the code and its different levels of abstraction in the following a vertical dive in the forward movement is reported. From the main code, that is the state machine one, the *walk_forward* case is extracted.

```
case 'walk_forward'

    step = 3; % step length
    theta_a = 0; % direction of the hexapod [deg] (0 -> forward, 90 -> right)
    N_routines = 5; % number of moving routines to be executed
    N_points = 16; % number of points in the trajectory discretization
    move_linear(legs, serial_obj, step, theta_a, N_routines, N_points);
    next_state = 'wait_for_input'; % Next state evaluation
```

Figure 7: Case Walk Forward in State Machine

From here the lower level function *move_linear* is called, with in inputs the parameters commented before, the legs' models defined with the Robotics toolbox and the Serial communication object. In the move linear function some main data structures are defined.

```
tj_support = zeros(N_points, 2, 6);
tj_return = zeros(N_points, 2, 6);
tj_positioning = zeros(N_points/2, 2, 6);
tj_stabilize = zeros(N_points/2, 2, 6);
P0 = zeros(6, 3, 1); % initial point of the trajectory
P1 = zeros(6, 3, 1); % end point of the trajectory
group1 = [1 3 5]; % leg group 1
group2 = [2 4 6]; % leg group 2
```

Figure 8: *move_linear* Data Structures

In particular must be noted the 4 tridimensional matrices declared. This matrices are needed in order to store all the computations relative to the different phases. Specifically in the first dimension of the matrix we have the number of points in which the trajectory is divided, in the second dimension the specific motor we are referring for each leg (each leg has 2 motors) and in the third dimension the leg we are considering. In *P0* and *P1* respectively the starting point and ending point of the support trajectory are stored. Moreover legs are divided in groups for the computations in order to comply with the triangle gait. We decided to heavily use the matrix storage and computation to exploit the Matlab optimization with these data structures. In the following the computation and storage of the trajectories is reported.

```
for i=1:6
    % inverse kinematics for each leg
    [tj_support(:, :, i), P0(i, :, :), P1(i, :, :)] = kinematic_inversion(legs, step, theta_a, i, N_points);

    % create joints' routines for each leg
    tj_positioning(:, :, i) = create_joint_traj(tj_support(:, :, i), N_points, 'positioning', i);

    tj_return(:, :, i) = create_joint_traj(tj_support(:, :, i), N_points, 'return', i);
    tj_stabilize(:, :, i) = create_joint_traj(tj_support(:, :, i), N_points, 'stabilizing', i);
end
```

Figure 9: Trajectory computation

Something important is the difference between the support trajectory, for which the legs' models are needed in the kinematic inversion, and the other ones. In the latter ones the computation only needs the support trajectory and the type of trajectory we want to compute, being this information important to define the start and end points. The results of this computations are stored and used afterwards in loop in order to avoid online computations that could slower the process.

Stabilization through MPU6050

Concept phase

After the realization of the walking process (linear, rotational) the next task was represented by the reaction of our Hexapod to external inputs thanks to the 2 sensors it was equipped with. One of them is just one of the most common sensor, the MPU-6050. Given the complexity of working with accelerations and velocities we focused our objective to a specific task: stabilizing the robot horizzontally w.r.t. an inclined plane.
So the basic concept consisted in detecting a reasonable value of the inclination angle allowing our robot to react repositioning itself on the imaginary horizontal plane. The considered rotation axis was the x one.

Geometrical approach

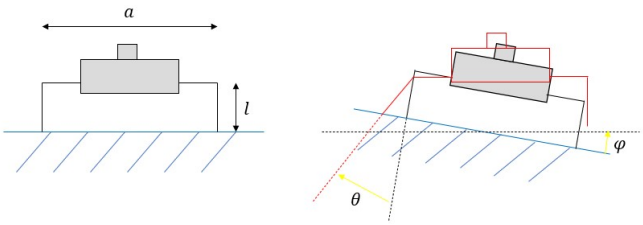

Figure 10: Basic geometrical scheme

The first objective was to find a way capable of getting a part of the Hexapod down when the plane caused a corresponding "positive" inclination. This was reached by imposing a certain angle θ to the second revolute joint (through the corresponding servomotor). We wanted to find a relation between the above mentioned angle and the angle relative to the plane inclination φ.
So the following formulas were be implemented in the further algorithm:

$$\theta \ = \ cos^{-1}(1 - \frac{a \cdot sin\ \varphi}{l})$$

NB: Defining h as the height we have to compensate, must be mentioned that all these formulas are valid as long as the angles are small, because in the real experience h is not a vertical segment but it is an arc of a circle.

Implemented code with practical solutions

In order to receive the data from the MPU6050 we adopted as always the serial communication between Matlab and Arduino Uno. Initially we saw that basic imported libraries had a divergent trend with respect to the output produced (i.e. increasing angles registered in steady position). To overcome such an issue we looked at a more sophisticated architecture, that was the implementation of a Kalman filter in order to clean the output from measurement disturbances/noises. Also this one was possible thanks to a piece of code present on the web specifically developed for the Arduino Uno platform [4].

After this first achievement we concentrated on how to put together the theoretical behaviour of the experiment and the effective code that could give us a good approximation of what we thought. The second problem was to express the angle of the plane: through the sensor we just obtained the angle of the robot plane that for our purpose had to be set on 0°. So for example the sensor could report a value near 0° but the plane was effectively inclined. The choice was to update in real time the plane angle as the sum of the previous one and the actual data of the sensor. From here some empirical solutions were implemented :

```
theta_old = 0; % inclined plane angle (start: 0)
threshold = 4.5; % angle threshold for the inclined plane to decide to take action
K = 0.2; % controller gain

while(1)
    [angleX, ~] = arduino_read_angle(serial_obj)
    theta_new = theta_old+angleX-0.5

    if abs(theta_new)<threshold
        theta_new=angleX;
    else
        theta_new=theta_new;
    end
    theta_abs = K*abs(theta_new);
    arg = (l2-a*sind(theta_abs))/l2;
    q = acosd(arg);
```

Figure 11: Inclined plane stabilization

- **threshold:** we implemented an hysteresis cycle in order to avoid to start reacting to small angles and to avoid bouncing problems, since a sharp change of activated legs in a single central threshold would cause instability.
- **K:** it is a sort of proportional gain with respect to the φ angle (non-linear considering the joint variable control). This was needed in order to limit the

angular speed of leg motors. After a proper tuning we decided to put K = 0.2, this is reasonable considering that the variable q reaches the required reference too fast and so it needs to be slowed down.

Map of the environment

The map of the environment is a function of our robot which uses the ultrasonic sensor and the servo motor of the head of the robot to create a radar map.

The idea is to move the head of the robot 1 degree per cycle, collecting at each position two corresponding readings of the distance and averaging them. The result is then stored into a vector table, along with the corresponding angle of rotation. To have a better estimation of the data, the robot head goes from 20 to 160, but it also comes back, collecting other two measurements and doing the average. Again, this is done to have more accurate data, in order to compensate the uncertainty of the readings.

When the head comes back from 160 degrees to 20 degrees, we decided to implement another check. Since the ultrasonic sensor can measure only data between 2 and 400cm, any data outside 400cm has to be considered wrong. In case we have invalid data they are replaced with the average of right and left adjacent valid values.

At the end of the cycle, a polar plot give us an approximated map of the environment. Here follows an example of a map of an environment.

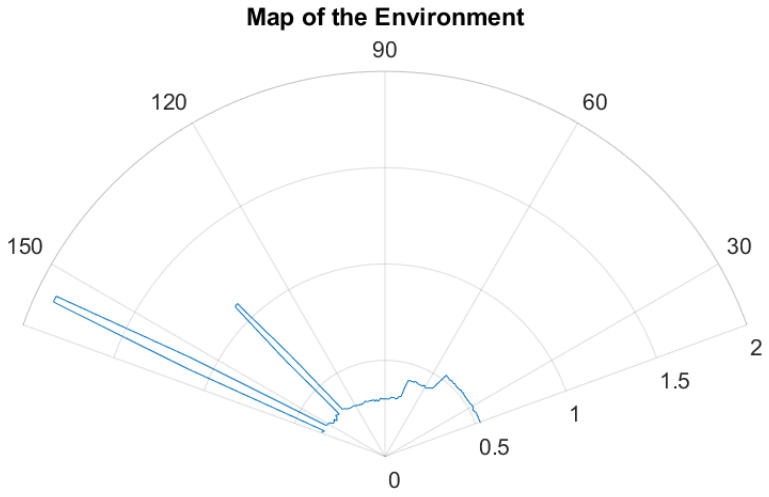

Figure 12: An example of the map of the environment

It has to be mentioned that despite our efforts to clean the data, sometimes we couldn't do that. The reason for this problem is that the wave can rebound all over the room and return to the echo pin with a time value completely wrong, but still in the 2cm to 400cm range. For this reason, in some cases the map of the environment is not accurate.

(Semi-)Autonomous navigation

To try to have autonomous navigation of the unknown environment we tried to couple the Ultrasonic sensor with all the kinematics developed. In particular we developed a loop in which after some forward movement routines the sensor values are read and the decision whether the robot has to change direction is taken. In particular the decision embed a non-deterministic logic that enable the robot (if it has to turn) to rotate in a probabilistic way left or right and by a probabilistic angle. We chose this approach in order to virtually explore all the unknown environment and avoid to limit the range of possible trajectories to a predefined set.

Possible improvements

An important drawback of the used platform is the lack of a third degree of freedom relative to each leg. This could improve considerably the robot resilience to rough terrains. Moreover this would enable a more complex and precise stabilization with respect to slopes and an implementation of the stair climbing feature. Another aspect to be noted is that we avoided the use of the Wi-fi card provided by the manufacturer due to the increasing complexity in dealing with the wireless communication. This feature could be very useful if implemented because it would enable complete autonomous navigation, coupled with our non-deterministic locomotion routine.

Bibliography

[1] P.I. Corke, “Robotics, Vision & Control”, Springer 2017, ISBN 978-3-319-54413-7

[2] Xia, H.; Zhang, X.; Zhang, H. A New Foot Trajectory Planning Method for Legged Robots and Its Application in Hexapod Robots. Appl. Sci. 2021, 11, 9217

[3] Kajita S., Espiau B. (2008) Legged Robots. In: Siciliano B., Khatib O. (eds) Springer Handbook of Robotics. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-540-30301-5_17

[4] https://github.com/TKJElectronics/Example-Sketch-for-IMU-including-Kalman-filter/tree/master/IMU/MPU6050 by Lauszus