
POLITECNICO DI BARI

DEI

LAUREA MAGISTRALE IN INGEGNERIA DELL'AUTOMAZIONE

---

Robotics

Prof. Ing. Paolo Lino
Ing. Nikolai Svishchev

*Progetto:*

# Control of a UUV through the Plankton platform

*Studenti:*
Marcello Bari
Alessandro Mitola

---

ANNO ACCADEMICO 2023-2024

# 1 Introduzione

Il nostro progetto si concentra sull'utilizzo della piattaforma software open-source Plankton UUV (Underwater Unmanned Vehicle), basata su ROS 2.
Essa viene impiegata principalmente per simulare robot marini, permettendo di studiarne il moto, di pianificare e controllare le loro traiettorie in modo efficiente, oltre che a raccogliere dati e misure di interesse attraverso sensori avanzati posti su di essi.
Noi ci siamo concentrati sul controllo del sottomarino soprannominato "rexrov", caratterizzato dall'avere otto propulsori (o *thrusters*) fissi, disposti simmetricamente rispetto l'asse principale del robot, come mostrato nella figura seguente:

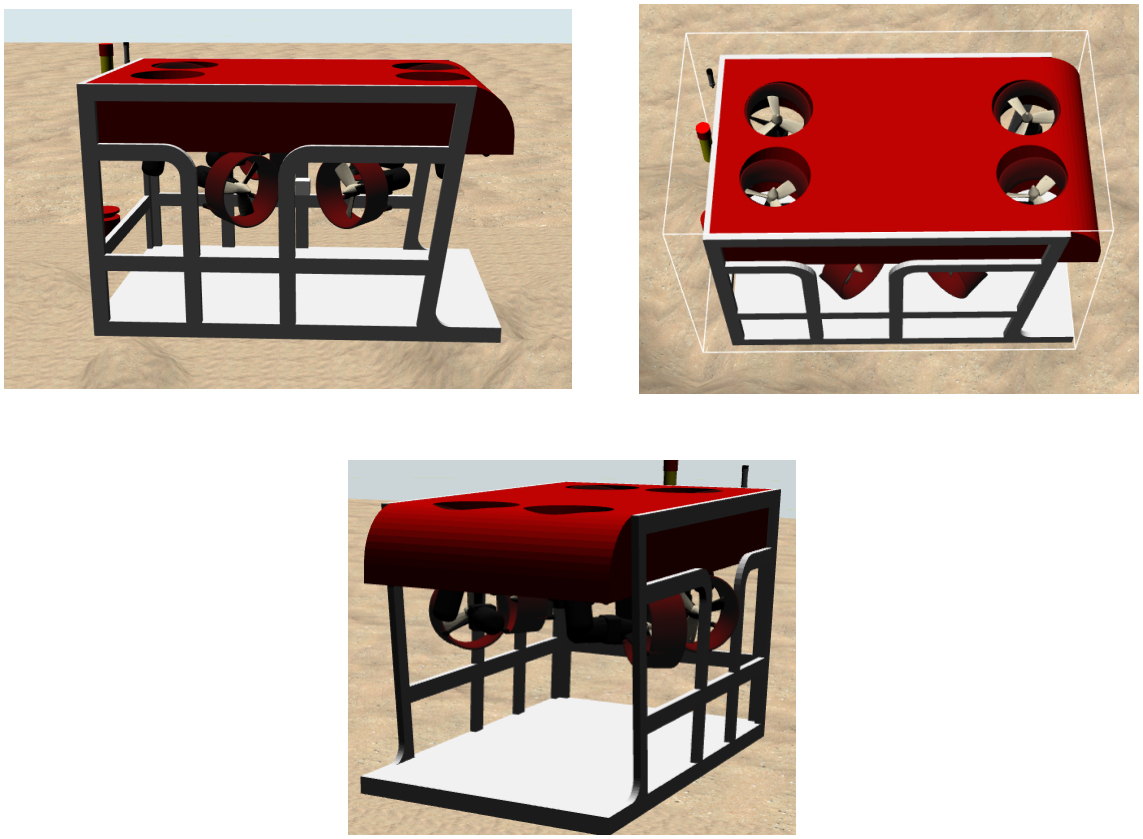

**Figure 1:** Modello 3D del UUV rexrov

Esso dispone anche di un ampio set di sensori, tra cui una telecamera, diversi sonar, IMU (unità di misura inerziale) e magnetometri, per consentire un'accurata raccolta di dati ambientali e il monitoraggio in tempo reale, garantendo precisione nella navigazione e nella mappatura subacquea.
Per quel che riguarda l'installazione di Plankton, avendo a disposizione solo pc con sistema operativo Windows, abbiamo utilizzato WSL, modificandone il kernel per includere i driver necessari per permettere il controllo del ROV tramite joystick.

# 2 Modello matematico del ROV

La maggior parte delle relazioni analitiche riguardanti i ROV sono tratte dai lavori di [1], [2], [3] e riguardano la sua cinematica, dinamica e controllo.

Pertanto, qui mostriamo in che modo tali equazioni vengono utilizzate per il modello di robot da noi studiato.

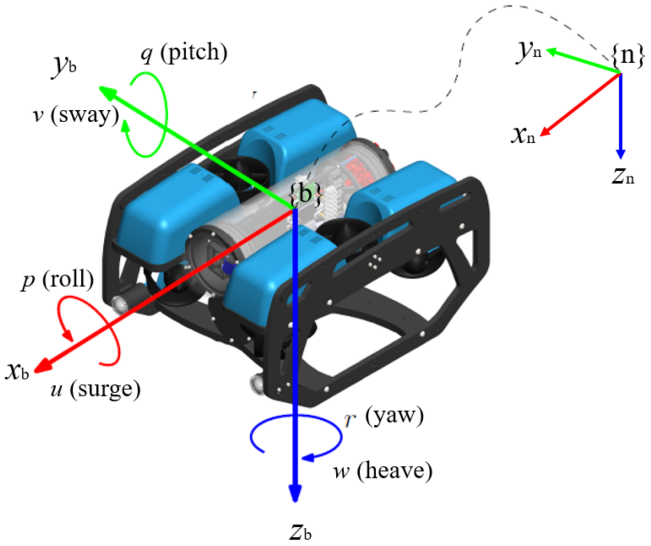


**Figure 2:** Sistemi di riferimento per il ROV

Seguendo un approccio standard alla cinematica, si può ricavare un modello per il ROV, considerando i due sistemi di riferimento mostrati in figura 2 :

- Una terna fissa avente assi $\{x_n, y_n, z_n\}$, all'interno della quale la posizione e l'orientamento del sottomarino sono date dal vettore $[x, y, z, \phi, \theta, \psi]$, dove gli angoli $\phi$, $\theta$ e $\psi$ rappresentano rollio, beccheggio e imbardata rispettivamente. Per semplicità si impone $\eta_1 = [x, y, z]$ e $\eta_2 = [\phi, \theta, \psi]$
- una terna mobile, centrata nel centro di massa del robot e solidale ad esso, avente assi $\{x_b, y_b, z_b\}$. Le componenti di velocità lineare e angolare espresse in questo sistema di riferimento sono rispettivamente $[u, v, w, p, q, r]$. Per semplicità si impone $v_1 = [u, v, w]$ e $v_2 = [p, q, r]$

L'equazione matriciale che lega le componenti della velocità espresse in questi sistemi di riferimento ha la forma seguente:

$$\begin{bmatrix} \dot{\eta_1} \\ \dot{\eta_2} \end{bmatrix} = \begin{bmatrix} R_1(\eta_2) & 0_{3\times3} \\ 0_{3\times3} & R_2(\eta_2) \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} \tag{1}$$

Dove $R_1(\eta_2)$ è la matrice di rotazione che lega le velocità lineari nei due sistemi di riferimento, mentre $R_2(\eta_2)$ mette in relazione le componenti delle velocità angolari espressi nelle due terne.
Il valore di queste matrici varia al variare della posizione del ROV all'interno dell'ambiente di lavoro, e in Plankton vengono continuamente aggiornate nelle seguenti sezione di codice del file *vehicle.py*:

```
    def update_odometry(self, msg):
            # Update the velocity vector
            # Update the pose in the inertial frame
            self._pose['pos'] = np.array([msg.pose.pose.position.x,
                                          msg.pose.pose.position.y,
                                          msg.pose.pose.position.z])
```

```
            # Using the (x, y, z, w) format for quaternions
            self._pose['rot'] = np.array([msg.pose.pose.orientation.x,
                                          msg.pose.pose.orientation.y,
                                          msg.pose.pose.orientation.z,
                                          msg.pose.pose.orientation.w])
            # Linear velocity on the INERTIAL frame
            lin_vel = np.array([msg.twist.twist.linear.x,
                                msg.twist.twist.linear.y,
                                msg.twist.twist.linear.z])
            # Transform linear velocity to the BODY frame
            lin_vel = np.dot(self.rotItoB, lin_vel)
            # Angular velocity in the INERTIAL frame
            ang_vel = np.array([msg.twist.twist.angular.x,
                                msg.twist.twist.angular.y,
                                msg.twist.twist.angular.z])
            # Transform angular velocity to BODY frame
            ang_vel = np.dot(self.rotItoB, ang_vel)
            # Store velocity vector
            self._vel = np.hstack((lin_vel, ang_vel))
```

```
        def q_to_matrix(q):
            e1 = q[0]
            e2 = q[1]
            e3 = q[2]
            eta = q[3]
            R = np.array([[1 - 2 * (e2**2 + e3**2),
                           2 * (e1 * e2 - e3 * eta),
                           2 * (e1 * e3 + e2 * eta)],
                          [2 * (e1 * e2 + e3 * eta),
                           1 - 2 * (e1**2 + e3**2),
                           2 * (e2 * e3 - e1 * eta)],
                          [2 * (e1 * e3 - e2 * eta),
                           2 * (e2 * e3 + e1 * eta),
                           1 - 2 * (e1**2 + e2**2)]])
            return R
        def rotBtoI(self):
            # Using the (x, y, z, w) format to describe quaternions
            return self.q_to_matrix(self._pose['rot'])
        def TBtoIquat(self):
            e1 = self._pose['rot'][0]
            e2 = self._pose['rot'][1]
            e3 = self._pose['rot'][2]
            eta = self._pose['rot'][3]
            T = 0.5 * np.array(
                [[-e1, -e2, -e3],
                 [eta, -e3, e2],
                 [e3, eta, -e1],
                 [-e2, e1, eta]]
            )
            return T
```

La funzione *update_odometry* è invocata ogni qual volta i sensori pubblicano un messaggio di tipo *nav_msgs/Odometry* sul topic *pose_gt* e si occupa di mantenere aggiornate le informazioni riguardanti posa e moto del ROV. I dati riportati dai sensori sono sempre riferiti al sistema di riferimento fisso, infatti si può notare come le velocità recuperate vengano convertite immediatamente mediante matrice di rotazione.
Plankton utilizza sempre quaternioni per rappresentare l'orientamento degli oggetti , pertanto le funzioni *q_to_matrix* e *rotBtoI* si occupano proprio della conversione da quaternione a matrice di rotazione.
Il risultato della funzione *TBtoIquat* è il quaternione necessario a esprimeree le velocità angolari del veicolo nel sistema di riferimento del veicolo, nel sistema di riferimento fisso.

Le equazioni dinamiche del moto, nel sistema mobile del robot, possono esprimersi nella forma seguente:

$$M \begin{bmatrix} \dot{v}_1 \\ \dot{v}_2 \end{bmatrix} + C(v) \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} + D(v) \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} + g(\eta) = \tau \tag{2}$$

Laddove, $M \in \mathbb{R}^{6\times6}$ è la matrice di inerzia (masse e momenti di inerzia), $C(v) \in \mathbb{R}^{6\times6}$ è la matrice contenente i termini di Coriolis e il contributo delle forze centripete, $D(v) \in \mathbb{R}^{6\times6}$ è la matrice che modella gli effetti dello smorzamento, $g(v) \in \mathbb{R}^6$ è il vettore dei termini gravitazionali e $\tau(v) \in \mathbb{R}^6$ è l'output del controllore, il quale specifica le forze e i momenti a cui deve essere sottoposto il ROV per inseguire il riferimento assegnatogli.
Quest'ultimo termine non specifica, quindi, le forze che ciascuno degli 8 propulsori ($n = 8$) deve sviluppare. Esse, infatti, vengono calcolate attraverso l'ausilio di una particolare matrice, il cui nome è *Thruster Allocation Matrix* o *TAM*, che mette in relazione il vettore $\tau$ con il vettore $F \in \mathbb{R}^8$ delle forze di propulsione:

$$\tau = TAM \cdot F$$

Nel caso del *rexrov*, $TAM \in R^{6\times8}$, e ciascuno dei suoi elementi dipende dalla struttura del veicolo e dalla disposizione dei propulsori su di esso.
Detto $p_i \in \mathbb{R}^3$ il vettore posizione dell'i-esimo attuatore rispetto il sistema di riferimento mobile, $Q_i \in \mathbb{R}^{3\times3}$ la matrice di rotazione che descrive l'orientamento dell'i-esimo attuatore rispetto il riferimento del ROV e $a_i \in \mathbb{R}^3$ l'asse di rotazione dell'i-esimo propulsore nel proprio sistema di riferimento, l'i-esima colonna di TAM è data da:

$$TAM_i = \begin{bmatrix} Q_i \cdot a_i \\ p_i \times (Q_i \cdot a_i) \end{bmatrix} \tag{3}$$

Per recuperare i dati necessari a svolgere i calcoli è necessario consultare il file .urdf del *rexrov*, che descrive la struttura del veicolo.

```
<joint name="rexrov/thruster_0_joint" type="continuous">
  <origin rpy="0.0 -1.3007937037 -0.9286895609" xyz="-0.890895 0.334385 ...
  0.528822"/>
  <axis xyz="1 0 0"/>
  <parent link="rexrov/base_link"/>
  <child link="rexrov/thruster_0"/>
</joint>
```

Qui, l'orientamento del propulsore è descritto mediante la terna di angoli RPY, quindi bisogna costruire la matrice di rotazione corrispondente utilizzando le note regole di composizione di rotazioni. Inoltre tutti i propulsori del ROV hanno asse pari all'asse x, come indicato dal tag <axis>.
Una volta ottenuta la TAM, essa viene memorizzata in un file .yaml, in modo da essere riutilizzata. La sua pseudo-inversa (comando *numpy.linalg.pinv* in Python) è utilizzata da tutti i controllori per convertire le azioni di controllo in segnali di forza per gli attuatori.
I propulsori montati sul ROV sono di tipo proporzionale, ovvero caratterizzati da una costante, definita tra i parametri di ROS2 come "gain" (per il *rexrov* è 0.00031), che permette di tradurre il set-point di forza in un set-point di velocità dell'elica. Plankton, infatti, comunica con gli attuatori tramite messaggi di tipo *FloatStamped()* (ovvero attraverso float a 64 bit) relativi a velocità angolari ($\frac{rad}{s}$).
La conversione, per ciascun propulsore, ha la forma seguente:

$$\omega_i = sign(F_i) * \sqrt{\frac{\|F_i\|}{gain_i}} \tag{4}$$

Ed è implementata nel file *thruster_proportional.py*:

```
    def get_command_value(self, thrust):
        return numpy.sign(thrust) * \
            numpy.sqrt(numpy.abs(thrust) / self._gain)
```

# 3 Controllo e struttura della rete ROS2

Per quel che riguarda il controllo del ROV, la piattaforma Plankton implementa nativamente moltissimi tipi di controllori.
Per comprendere al meglio come Plankton strutturi il loro avvio e la loro gestione, nonché il modo con cui esso garantisca la comunicazione fra tutti gli elementi della simulazione, noi ci siamo concentrati sullo studio e analisi di un PID.
La struttura semplificata di controllo predisposta da Plankton ha la forma seguente:

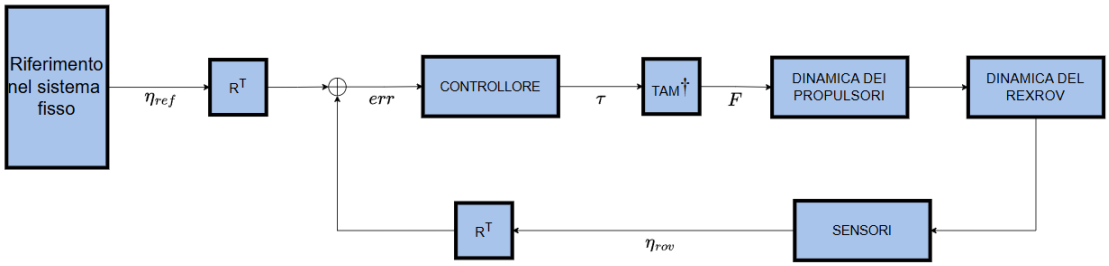


**Figure 3:** Schema di controllo per il ROV

Il primo blocco da analizzare è quello relativo alla generazione del riferimento di posa per il ROV. È possibile scegliere se generare traiettorie a 6 gradi di libertà o solo a 4, questo perché, a differenza di quanto accade con gli aeroplani, i sottomarini vengono operati in modo che il piano individuato dagli assi $x_b$ e $y_b$ rimanga parallelo al piano del livello dell'acqua (quindi beccheggio e rollio sempre nulli). Un'altra opzione disponibile, è quella che permette di scegliere se generare anche riferimenti di velocità e accelerazione per il ROV. Di default, Plankton genererà sempre traiettorie a 4 gradi di libertà, che riguardano solamente la posa del veicolo. Per cambiare questo comportamento è sufficiente alterare i valori degli argomenti di input della classe *TrajectoryGenerator*:

```
    def __init__(self, node: Node, full_dof=False, stamped_pose_only=False)
```

La creazione di una traiettoria avviene sempre mediante interpolazione di waypoint, i quali possono essere specificati singolarmente su un file personalizzato, o generati da script. Qualsiasi sia il modo in cui si desidera dichiararli, la loro creazione effettiva sarà affidata a uno dei servizi ROS, dichiarato nella classe *DPControllerLocalPlanner*:

```
    srv_name = 'hold_vehicle'
            self._services[srv_name] = node.create_service(Hold, srv_name, self....
       hold_vehicle, callback_group=callback_group)

            srv_name = 'start_waypoint_list'
            self._services[srv_name] = node.create_service(
                InitWaypointSet, srv_name, self.start_waypoint_list, ...
       callback_group=callback_group)

            srv_name = 'start_circular_trajectory'
```

```
            self._services[srv_name] = node.create_service(
                InitCircularTrajectory, srv_name, self.start_circle, ...
        callback_group=callback_group)

            srv_name = 'start_helical_trajectory'
            self._services[srv_name] = node.create_service(
                InitHelicalTrajectory, srv_name, self.start_helix, callback_group...
        =callback_group)

            srv_name = 'init_waypoints_from_file'
            self._services[srv_name] = node.create_service(
                InitWaypointsFromFile, srv_name, self.init_waypoints_from_file, ...
        callback_group=callback_group)

            srv_name = 'go_to'
            self._services[srv_name] = node.create_service(GoTo, srv_name, self....
        go_to, callback_group=callback_group)

            srv_name = 'go_to_incremental'
            self._services[srv_name] = node.create_service(
                GoToIncremental, srv_name, self.go_to_incremental, callback_group...
        =callback_group)
```

Noi ci siamo concentrati sul servizio che permette la generazione di una traiettoria elicoidale. Per utilizzarlo, è necessario lanciare un nuovo nodo, avente lo stesso nome ( *start_helical_trajectory*), che possa invocarlo, iscrivendosi ad esso sotto forma di client. Il launch file di questo nuovo nodo è qui riportato:

```
<launch>
  <arg name="uuv_name" />
  <arg name="start_time" default="-1"/>
  <arg name="radius" default="8"/>
  <arg name="center_x" default="0"/>
  <arg name="center_y" default="0"/>
  <arg name="center_z" default="-30"/>
  <arg name="n_points" default="50"/>
  <arg name="n_turns" default="1"/>
  <arg name="Δ_z" default="5.0"/>
  <!-- Heading offset given in degrees -->
  <arg name="heading_offset" default="0"/>
  <arg name="duration" default="150"/>
  <arg name="max_forward_speed" default="0.3"/>

  <group>
    <push-ros-namespace namespace="$(var uuv_name)"/>
    <node pkg="uuv_control_utils"
      exec="start_helical_trajectory.py"
      name="start_helical_trajectory"
      output="screen">
      <param name="start_time" value="$(var start_time)"/>
      <param name="radius" value="$(var radius)"/>
      <param name="center" value="$(var center_x), $(var center_y), $(var ...
    center_z)" value-sep=", "/>
      <param name="n_points" value="$(var n_points)"/>
      <param name="heading_offset" value="$(var heading_offset)"/>
      <param name="max_forward_speed" value="$(var max_forward_speed)"/>
      <param name="duration" value="$(var duration)"/>
      <param name="n_turns" value="$(var n_turns)"/>
      <param name="Δ_z" value="$(var Δ_z)"/>
    </node>
  </group>
</launch>
```

L'argomento *uuv_ name* (per noi è *rexrov*) è utilizzato da ogni launch file di Plankton, e assieme al tag *push-ros-namespace*, fa si che i topic relativi allo stesso veicolo comincino tutti per *uuv_ name*, per non creare ambiguità qualora più veicoli siano presenti nella simulazione. Gli altri argomenti servono semplicemente a creare la geometria dell'elica.
Lo script *start_ helical_ trajectory.py* avvia effettivamente il nodo e richiede il servizio di generazione di traiettoria:

```
    node = rclpy.create_node(
        'start_helical_trajectory',
        allow_undeclared_parameters=True,
        automatically_declare_parameters_from_overrides=True,
        parameter_overrides=[sim_time_param])

    srv_name = 'start_helical_trajectory'
    traj_gen = node.create_client(InitHelicalTrajectory, '...
    start_helical_trajectory')

    req = InitHelicalTrajectory.Request()
    req.start_time = rclpy.time.Time(seconds=sec, nanoseconds=nsec).to_msg()
    req.start_now = start_now
    req.radius = float(params['radius'])
    req.center = Point(x=float(params['center'][0]),
                       y=float(params['center'][1]),
                       z=float(params['center'][2]))
    req.is_clockwise = False
    req.angle_offset = 0.0
    req.n_points = params['n_points']
    req.heading_offset = float(params['heading_offset'] * pi / 180)
    req.max_forward_speed = float(params['max_forward_speed'])
    req.duration = float(params['duration'])
    # As partial turns can be given, it is a float
    req.n_turns = float(params['n_turns'])
    req.△_z = float(params['△_z'])

    future = traj_gen.call_async(req)
    rclpy.spin_until_future_complete(node, future)
    try:
        response = future.result()
```

La variabile *req* è la "richiesta" contenente tutti i parametri necessari per generare l'elica. A operazione conclusa, il servizio risponde con un dato booleano e il nodo viene distrutto.
Il servizio in sé, richiama un'istanza della classe WaypointSet, sulla quale avvia il metodo *generate_ helix* che restituisce l'insieme di waypoints desiderato. Inoltre, viene selezionato il tipo di interpolatore da utilizzare. Nel nostro caso, si tratta di curve di Bezier del terzo ordine :

```
try:
    wp_set = uuv_waypoints.WaypointSet(inertial_frame_id=self....
    inertial_frame_id)
    success = wp_set.generate_helix(radius=request.radius,
                                    center=request.center,
                                    num_points=request.n_points,
                                    max_forward_speed=request....
    max_forward_speed,
                                    △_z=request.△_z,
                                    num_turns=request.n_turns,
                                    theta_offset=request.angle_offset,
                                    heading_offset=request.heading_offset)

    self._lock.acquire()
    self.set_station_keeping(True)
    self._traj_interpolator.set_interp_method('cubic')
```

```
    def generate_helix(self, radius, center, num_points, max_forward_speed, Δ...
    _z,
                       num_turns, theta_offset=0.0, heading_offset=0.0,
                       append=False):
        if not append:
            # Clear current list
            self.clear_waypoints()

        total_angle = 2 * np.pi * num_turns
        step_angle = total_angle / num_points
        step_z = float(Δ_z) / num_points
        for i in range(num_points):
            angle = theta_offset + i * step_angle
            x = radius * np.cos(angle) + center.x
            y = radius * np.sin(angle) + center.y
            z = step_z * i + center.z

            wp = Waypoint(x, y, z, max_forward_speed,
                          heading_offset)
            self.add_waypoint(wp)
```

L'interpolazione comincerà successivamente, quando il controllore richiederà un riferimento attraverso il metodo *interpolate* della classe *DPControllerLocalPlanner*.
Ora l'analisi di concentrerà sul controllore e su come esso interagisca con gli altri elementi della catena di controllo. Il launch file relativo a un controllore PID è molto grande, quindi metterò in evidenza solo le parti fondamentali:

```
    <arg name="uuv_name"/>
    <arg name="model_name" default="$(var uuv_name)"/>
    <arg name="saturation" default="1200"/>
    <arg name="gui_on" default="true"/>
    <arg name="use_params_file" default="false"/>
    <arg name="use_ned_frame" default="false"/>
    <arg name="controller_config_file" default="$(find-pkg-share ...
      uuv_trajectory_control)/config/controllers/pid/$(var model_name)/params....
      yaml"/>
    <arg name="thruster_manager_output_dir" default="$(find-pkg-share ...
      uuv_thruster_manager)/config/$(var model_name)"/>
    <arg name="thruster_manager_config_file" default="$(find-pkg-share ...
      uuv_thruster_manager)/config/$(var model_name)/thruster_manager.yaml"/>
    <arg name="tam_file" default="$(find-pkg-share uuv_thruster_manager)/config...
      /$(var model_name)/TAM.yaml"/>
```

L'argomento *use_params_file* deve sempre essere posto al valore *true*, in modo da garantire che i parametri $K_p$, $K_i$, $K_d$ vengano recuperati dal file individuato dall'argomento *controller_config_file*.
I valori che abbiamo imposto sono quelli "ottimali" consigliati dagli autori dei paper sopra citati, ovvero $K_p = [11993.888, 11993.888, 11993.888, 19460.069, 19460.069, 19460.069]$, $K_d = [9077.459, 9077.459, 9077.459, 18880.925, 18880.925, 18880.925]$ e $K_i = [321.417, 321.417, 321.417, 2096.951, 2096.951, 2096.951]$.
L'argomento *use_ned_frame*, invece, definisce il tipo di convenzione utilizzata per i sistemi di riferimento. La convenzione NED (North-East-Down) è utilizzata sopratutto in campo aeronautico e navale, in quando permette di definire distanze positive fra veicolo e suolo, ed ha asse x rivolto a nord, asse y rivolto ad est e asse z rivolto in basso. La convenzione ENU (East-North-Up), invece, è caratterizzata dall'avere l'asse z rivolto verso l'alto e gli assi x e y invertiti rispetto alla NED. Plankton attiverà sia i riferimento ENU che NED all'avvio della simulazione, ma solo quello selezionato sarà utilizzato per svolgere calcoli.
In RVIZ i due frame appaiono così:

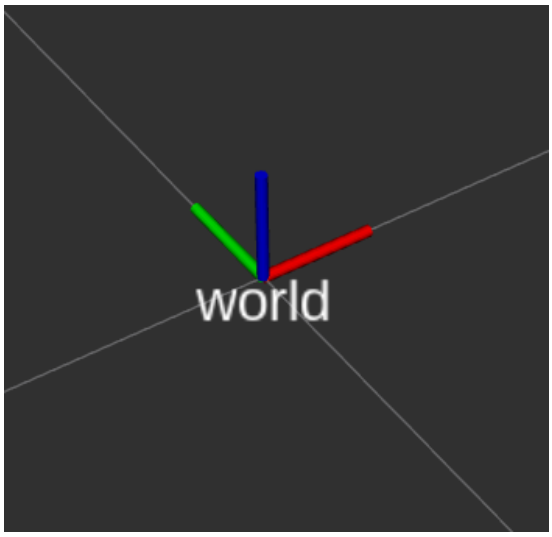


**Figure 4:** frame ENU

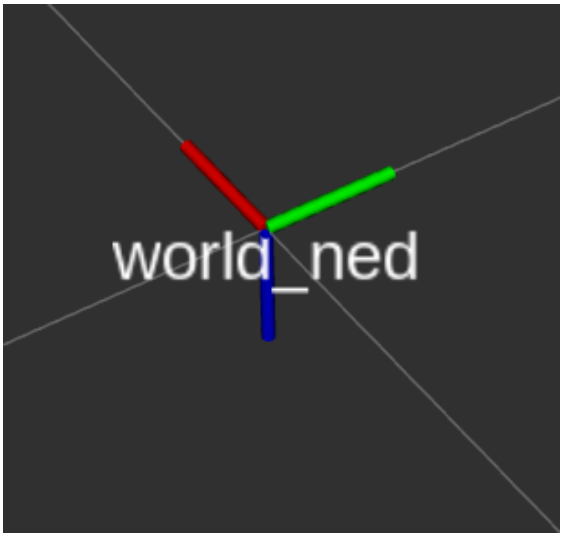


**Figure 5:** frame NED

Il resto degli argomenti è stato già affrontato nel capitolo 2.
Il launch file prosegue con l'inizializzazione di tre nodi: uno al solo scopo di pubblicare su RVIZ traiettorie e waypoint, e gli altri due dedicati al controllore e ai propulsori, chiamati *rov_pid_controller* e *thruster_allocator* rispettivamente. La classe su cui poggiano tutti gli script dei controllori è chiamata *DPControllerBase* . Essa si occupa principalmente di mantenere aggiornati i punti che il ROV deve inseguire, nonché gli errori di posa del *rexrov* rispetto al riferimento, e di inviare l'output di controllo ai nodi che gestiscono i propulsori.

```
        self._reference = dict(pos=np.zeros(3),
                               rot=np.zeros(4),
                               vel=np.zeros(6),
                               acc=np.zeros(6))

        self._errors = dict(pos=np.zeros(3),
                            rot=np.zeros(4),
                            vel=np.zeros(6))

        self._odometry_callbacks = [self.update_errors,
                                    self.update_controller]
        self._odom_topic_sub = self.create_subscription(
            Odometry, 'odom', self._odometry_callback, 10,
            callback_group=self._callback_group)

        def _odometry_callback(self, msg):
            self._vehicle_model.update_odometry(msg)
            if not self._init_odom:
                self._init_odom = True
            if len(self._odometry_callbacks):
                for func in self._odometry_callbacks:
                    func()
```

Il dizionario *_reference* contiene, istante per istante, il punto di riferimento per la traiettoria del sottomarino, espresso nel sistema di coordinate fisso. Invece, il dizionario *_erros* contiene gli errori di posa espressi nel sistema di coordinate mobile.
Ogni qual volta che un messaggio è inviato sul topic */pose_gt* (il topic /odom è rinominato a /pose_gt per permettere la comunicazione, fra gli output dei sensori, trattati brevemente nel capitolo 2 e il nodo del controllore) vengono aggiornate le seguenti entità: riferimento, errori e

azione di controllo. Il nuovo punto di riferimento e il nuovo valore dell'errore vengono pubblicati sui topic */dp_ controller/reference* e */dp_ controller/error* rispettivamente.
Il nodo pubblica anche sul topic */thruster_ manager/input_ stamped* le azioni di controllo, calcolate con il metodo *update_ control*, che viene implementato dalla classe *DPPIDControllerBase*, specifica del PID, la quale eredita dalla classe di controllo generale. Ovviamente, qui, *update_ control* si presenta nel modo seguente:

```
        def update_controller(self):
            if not self._is_init:
                return False
            self._tau = self.update_pid()
            self.publish_control_wrench(self._tau)
            return True

        def update_pid(self):
            if not self.odom_is_init:
                return
            # Update integrator
            self._int += 0.5 * (self.error_pose_euler + self._error_pose) * self....
        _dt
            # Store current pose error
            self._error_pose = self.error_pose_euler
            return np.dot(self._Kp, self.error_pose_euler) \
                + np.dot(self._Kd, self._errors['vel']) \
                + np.dot(self._Ki, self._int)

        def publish_control_wrench(self, force):
            if not self.odom_is_init:
                return
            # Apply saturation
            for i in range(6):
                if force[i] < -self._control_saturation:
                    force[i] = -self._control_saturation
                elif force[i] > self._control_saturation:
                    force[i] = self._control_saturation
            if not self.thrusters_only:
                surge_speed = self._vehicle_model.vel[0]
                self.publish_auv_command(surge_speed, force)
                return
            force_msg = WrenchStamped()
            force_msg.header.stamp = self.get_clock().now().to_msg()
            force_msg.header.frame_id = '%s/%s' % (self._namespace, self....
        _vehicle_model.body_frame_id)
            force_msg.wrench.force.x = force[0]
            force_msg.wrench.force.y = force[1]
            force_msg.wrench.force.z = force[2]

            force_msg.wrench.torque.x = force[3]
            force_msg.wrench.torque.y = force[4]
            force_msg.wrench.torque.z = force[5]

            self._thrust_pub.publish(force_msg)
```

Il messaggio sul topic *thruster_ manager/input_ stamped*, pubblicato con il metodo *_ thrust_ pub.publish* è "catturato" dal nodo *thruster_ allocator*, che si occupa di tramutare le azioni di controllo in forze (come già spiegato precedentemente) per i propulsori e reindirizzare il messaggio ai singoli attuatori.
Le immagini dei nodi e topic attivi, riassumono quanto descritto:

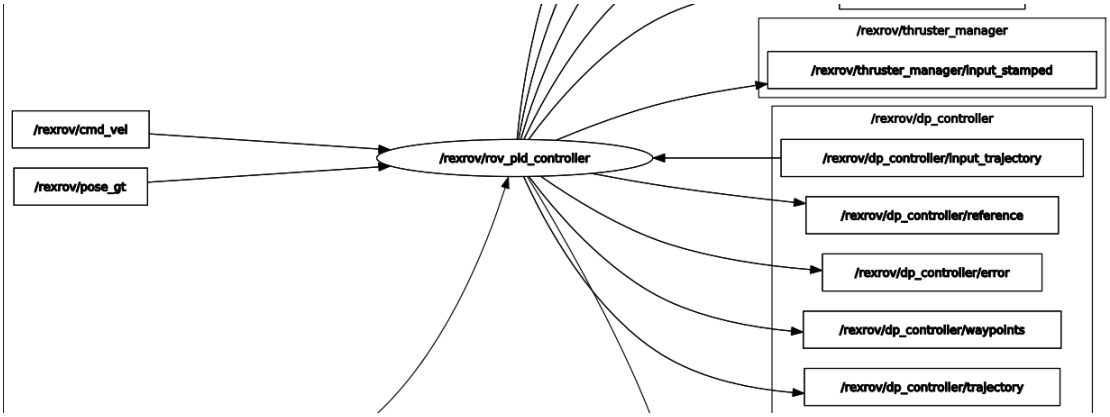


**Figure 6:** Nodi e topic del controllore

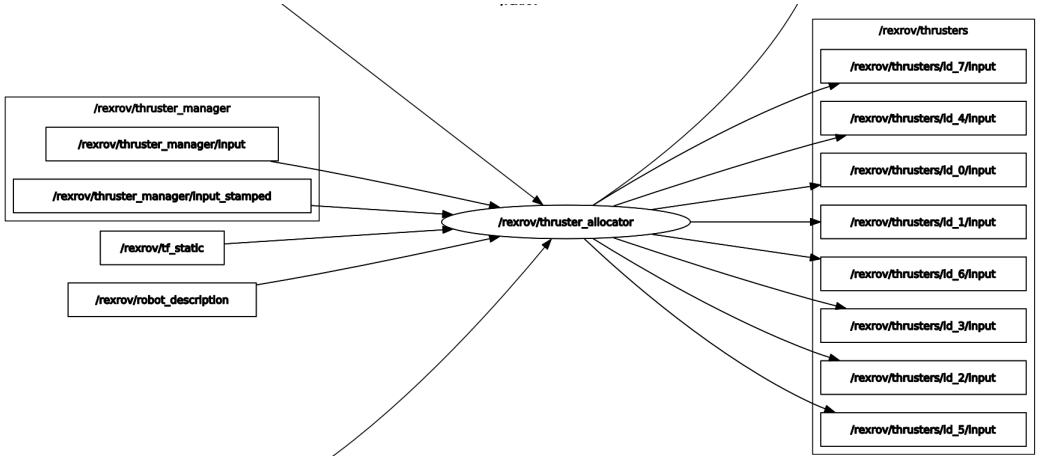


**Figure 7:** Nodi e topic del gestore dei propulsori

Abbiamo anche creato un file di configurazione RVIZ, per visualizzare correttamente tutti i sistemi di riferimento e le traiettorie percorse dal ROV. Infine avviando da terminale i seguenti comandi:

```
ros2 launch uuv_gazebo_worlds ocean_waves.launch
```

```
ros2 launch uuv_descriptions upload_rexrov.launch mode:=default namespace:=...
    rexrov
```

```
ros2 launch uuv_trajectory_control rov_pid_controller.launch uuv_name:=...
    rexrov
```

```
ros2 launch uuv_control_utils start_helical_trajectory.launch uuv_name:=...
    rexrov
```

Si ottengono i seguenti risultati:

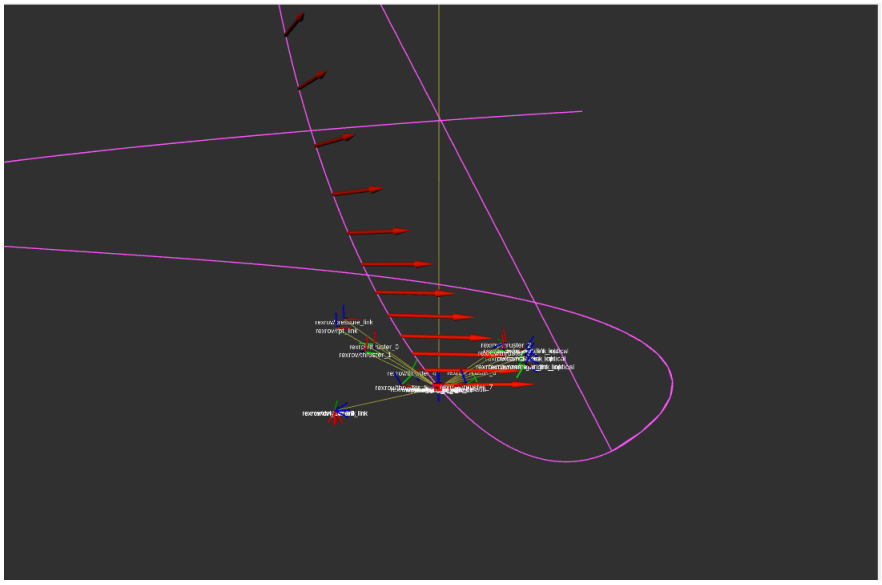

**Figure 8:** Risultato finale

# 4 Conclusioni

Quanto riportato in questo testo, è da intendersi come una sintesi del nostro lavoro svolto finora su Plankton e sul controllo del ROV.
Non avendo ricevuto istruzioni precise riguardo agli obiettivi di questo progetto, abbiamo anche provato a esplorare lo stack di navigazione ROS Nav2 con scarsi risultati. Nav2, infatti, necessita di un precisa organizzazione dei frame del robot. Il sistema *base_footprint* deve essere localizzato al di sotto del veicolo che si vuole controllare, e non al di sopra, come accade con il rexrov, dove è posto a livello dell'acqua. Ci sono complicazioni dovuta anche ai parametri che Nav2 utilizza per coordinare le operazioni di controllo.

# References

[1] O.-E. Fjellstad and T.I. Fossen. "Singularity-free tracking of unmanned underwater vehicles in 6 DOF". In: *Proceedings of 1994 33rd IEEE Conference on Decision and Control*. Vol. 2. 1994, 1128–1133 vol.2. DOI: 10.1109/CDC.1994.411068.

[2] Luis Govinda García-Valdovinos et al. "Modelling, Design and Robust Control of a Remotely Operated Underwater Vehicle". In: *International Journal of Advanced Robotic Systems* 11.1 (2014), p. 1. DOI: 10.5772/56810. eprint: https://doi.org/10.5772/56810. URL: https://doi.org/10.5772/56810.

[3] Tomás. Salgado-Jiménez, Luis G. García-Valdovinos, and Guillermo Delgado-Ramírez. "Control of ROVs using a Model-free 2nd-Order Sliding Mode Approach". In: *Sliding Mode Control*. Ed. by Andrzej Bartoszewicz. Rijeka: IntechOpen, 2011. Chap. 18. DOI: 10.5772/15951. URL: https://doi.org/10.5772/15951.