İTÜ Rover Team
Otonomi Dokümantasyonu

Sarp Sonal

August 9, 2025

Contents

1 Dosyalar

Otonom sürüş için kullanılan script, launch ve parametre dosyaları.

1.1 lidar.launch

lidar launch dosyası.

```
<arg name="rviz_config" default="$(find ouster_ros)/config/viz.rviz" doc="
    optional rviz config file"/>
```

(pointcloud)

1.2 sbg_device.launch

imu launch dosyası. parametreleri rover23_localization/config/imu.yaml'dan çekiyor.

1.3 udp.py

bir ros node'u oluşturup udp köprüsü/arayüzü sağlar.

gelen yön: vcu ve gnss'ten gelen veri paketini binary olarak çözer, içindeki gnss verisini çıkarır (little-endian) ve bu verileri Float64MultiArray halinde /gnss_data topic'inde yayınlar

giden yön: ros içinden /drive_system/vcu_Data isimli topic'e gelen (string) mesajları alır, bunları belirli bir protokol çerçevesinde belirlenen ip:port adresine yollar

BUFFER_SIZE = * -> recvfrom ile tek seferde en fazla kaç byte okunabildiğini belirtir

self.sock = socket.socket(AF_INET, SOCK_DGRAM) ile UDP socket oluşturulur,
self.sock.bind((LOCAL_IP, LOCAL_PORT)) ile bindlanır.

1.4 control_udp.launch

başlattığı üç node ile birlikte kumanda kolu, otonom navigasyon ve güvenlik kilitlerinden gelen twist mesajlarını tek hatta toplar, ardından vcu'ya iletir.

1.4.1 joy2twist_vcu_old.py

joystick'ten (/joy) ve otonom sistemden (/nav_vel) gelen hız komutlarını işler.
teleop/otonom mod geçişini ve hız ölçeklendirmesini yapar.
çıkış olarak /drive_system/twist yayınlar.

abonelikler:
/joy (gamepad): eksenler ve butonlar; teleop kontrol.
/locomove_base/is_reached (String): hedefe ulaşıldı bildirimi.
/locomove_base/current_goal (PoseStamped): yeni hedef bilgisi (ulaştı bilgisini sıfırlamak için).
/nav_vel (Twist): otonom planlayıcıdan hız.
/switch (Int32): otonomi-teleop anahtarı.

yayınlar:
/nav_vel1 (Twist): twist_mux için otonom hızın iletilmesi.
/move_base/cancel (GoalID): teleop'a dönüşte mevcut hedefi iptal eder.
/drive_system/twist (Twist): vcu'ya gidecek nihai twist.
/drive_system/status (String): led için vcu'ya mod bildirimi (teleop/otonomi).

1.4.2 vcu_udp.py

/drive_system/twist mesajını alır. VCU'ya uygun string formata çevirir (direction + velocity + LED color). serial/UDP üzerinden araca gönderir.

abonelikler:
/drive_system/twist: hızları almaya devam eder.
/drive_system/status: teleop, auto, success ile led rengi ayarlanır.

/manipulator/current_mode: manipülatör moduna göre renk önceliği.

yayınlar:
/drive_system/wheel_angular_velocities (Float64MultiArray): teker rpm.
weight_data (Float32): tartı.
/drive_system/vcu_data (String): vcu'ya gönderilen paket string'i.

komut paketinin kodlanması (String):
her döngüde angular_z varsa bir paket inşa edilir:
yaw direction (angular_dir): 0 (0) / 1 (>0).
yaw magnitude: abs(int(round(angular_z, 2)*100)) -> 3 hane zfill (ör. 0.45 rad/s -> "045").
x direction (linear_dir): 0 (0) / 1 (>0).
x magnitude: abs(int(round(linear_x, 3)*1000)) -> 4 hane zfill (ör. 0.123 m/s -> "0123").
led (bgr): "100" kırmızı, "001" mavi, "010" yeşil
bu adımlar tek bir stringe ardışık eklenip /drive_system/vcu_data olarak yayınlanır.

1.4.3 twist_mux.yaml

bir multiplexer. birden fazla cmd_vel kaynağını priority ile sıraya koyar ve en yüksek öncelikli geçerli kaynaktan tek bir cmd_vel_out üretir. bu çıkış /drive_system/twist olarak remap edilmiştir.

topics listesi:
navigation: /nav_vel1, timeout: 0.5, priority: 50 —> otonomi hattı.
joystick: /joystick/twist, timeout: 0.5, priority: 100 —> operatör teleop hattı.
robot_steering: /rqt_vel, priority: 90 —> RQT GUI gibi yardımcı sürüş.
hand_drive: /cmd_hand_drive, priority: 100 —> manuel el sürüşü.
lora_twist: /lora/rover/received/twist, timeout: 5.0, priority: 120 —> LoRa üzerinden telekomut (en yüksek önceliklerden).
e_stop: /stop, priority: 255 —> en yüksek önceliği olan emergency stop, geldiği zaman diğer her şey devre dışı kalır.

locks listesi:
e_stop_lock: /emergency_stop, priority: 255 —> emergency lock sinyali, var oldukça çıkış baskılanır.

1.5 ublox_odom.launch

odometri launch dosyası, built in aragaz ile
çalıştırıldığı zaman sensörleri kalibre etmek için 2 saniye boyunca ara_gaz.py dosyası ile obje algılama olmaksızın ileri doğru hareket eder
parametreleri ekf_params2.yaml dosyasından çeker

1.5.1 ekf_params2.yaml

extended kalman filtresi ve parametre dosyası
in robot_localization, each _config array has 15 booleans, one per state variable, in this exact order:
[x, y, z,
roll, pitch, yaw,
vx, vy, vz,
vroll, vpitch, vyaw,
ax, ay, az]
true -> fuse that variable from the given sensor; false -> ignore it.
15x15 covariance matrisleri de aynı sırayı izler, row-major olarak tek satıra açılmışlardır.

1.5.1.1 ekf_se_odom

yerel, kaymayan odom çerçevesi için ekf
sensor_timeout: t -> bir sensörden t sn içerisinde veri gelmez ise o sensör verisi stale sayılır ve yeni data gelene kadar ignorelanır
two_d_mode: true -> filtre 2d modda çalışır; z, roll ve pitch 0'lanır/ignorelanır.
transform_time_offset: t -> bu node'un paylaştığı TF'lara (odom->base_link gibi) t saniye offset ekler
transform_timeout: 0.0 -> yayınlanan TF'leri bekleme süresi yok, anında iletilmemiş ise hata verir.
map_frame: map -> global frame'nin adı (kullanılmıyor ancak robot_localization yine de tanımlanmasını talep ediyor)
odom_frame: odom -> odometry frame
base_link_frame: base_link -> state verilerinin referans alındığı frame'i belirtir.

yani, tüm pozisyon/ivme/hız gibi state bilgileri, robotun gövdesine ait base_link koordinat sistemine göre ifade edilmiştir.
world_frame: odom -> filtre çıkışının doğrudan odom -> base_link dönüşümünü yayınlamasını sağlar
predict_to_current_time: true -> yayınlamadan önce mevcut zamana kadar forward prediction yapar

1.5.1.1.1 odom0 (topic: ublox/odom)

odom0: ublox/odom -> ilk odometri source topic'i
odom0_config:
[true, true, false,
false, false, true,
true, true, false,
false, false, false,
false, false, false]
ublox/odom topicinden x, y, yaw, vx ve vy'yi füzyona dahil eder. yani bu odometri planar

position, heading ve planar linar velocity'i iletir.
odom0_queue_size: 10 -> subscriber queue size
odom0_differential: false -> absolute valueleri füzyonlar. eğer true olsaydı, filtre onları füzyonlamadan önce deltaya çevirirdi.
odom0_relative: false -> data are not relative to the first measurement (normal setting)

1.5.1.1.2 imu0: (topic: imu/data) imu0: imu/data -> ilk IMU source topic'i
imu0_config:
[false, false, false,
false, false, false,
false, false, false,
true, true, true,
false, false, false]
sadece angular velocityleri füzyonlar(vroll, vpitch, vyaw).
imu0_differential: false -> don't turn IMU values into deltas (IMU is already reporting rates and orientations).
imu0_relative: false -> not using relative measurements.
imu0_remove_gravitational_acceleration: true -> tell the filter the IMU driver is giving you linear acceleration including gravity, so please remove gravity before use. (You aren't fusing linear acceleration anyway, but leaving it consistent is good.)

1.5.1.1.3 control input use_control: false -> state predictora control inputları vermiyorsun onu belirtiyor (sanırım)

1.5.1.1.4 process noise covariance (Q) 15x15 flattened matrix. ekf modelin statelerinin zamanla nasıl wanderleyebileceğini belirleyen bir kovaryans matrisi.
elemanlar ne kadar büyükse, filtre modele o kadar az güveniyor ve state değerlerinin ölçümler arasında daha serbestçe değişmesine izin veriyor. tersi durumda (küçük değerler) filtre daha katı ve modelin önerdiği evrime daha bağlı kalıyor.

değerler:
positions (x,y,z): 1e-3 -> konumda çok ufak sapma izni (filtre modele çok güveniyor)
orientation (roll, pitch): 0.3 -> roll ve pitch açılarına orta düzeyde sapma izni
orientation (yaw): 0.01 -> yaw açısına çok ufak sapma (modele yüksek güven)
linear velocities (vx, vy): 0.5 -> yatay hızlara nispeten yüksek sapma izni
linear velocities (vz): 0.1 -> dikey hıza yataylara kıyasla daha sınırlı sapma izni
angular velocities (vroll, vpitch, vyaw): 0.3 -> açısal hızlara orta düzey sapma izni
linear accelerations (ax, ay, az): 0.3 -> ivme bileşenlerine orta düzey sapma izni

two_d_mode: true olduğundan dolayı z, roll ve pitch durumları kısıtlandığı için o satırlardaki değerlerin pratik etkisi azalır

1.5.1.1.5 initial estimate covariance (P) initial_estimate_covariance: [...] başka bir 15x15 flattened matrix. ekf'nin başlangıçtaki uncertainty'liğini belirtiyor. küçük değerler başlangıçtaki statelerden çok emin olunduğunu belirtiyor (x, y, z ve yaw rate accel'in 0 olmalarından neredeyse emin olması), tersi olarak da 1.0 değerine sahip statelerden uncertain olunduğunu belirtiyor.

1.5.1.2 ekf_se_map ikinci ekf konfigürasyonu. kullanılmıyor.

1.5.1.3 navsat_transform robot_localization/navsat_transform_node'in configi.
frequency: 30 -> transformlanmış gps datasını 30 Hz'de publishler.
delay: 3.0 -> 3sn gecikmeli yayın. sanırım tf ve ekf verilerini doğru zamanlarda bulup kullanabimek için.
magnetic_declination_radians: -0.286 -> manyetik sapma (radyan halinde). değer 55.944831, -3.186998 lat/long koordinatlarına göre, gerçek lokasyonuna göre bu değeri değiştirin.
yaw_offset: 2.79 -> imu 0 değerini manyetik kuzeye dönükken okuyor, bunu ENU (east-north-up) ile hizalama amaçlı ek bir yaw offseti. (160 dereceye tekabül ediyor, kontrol etmek lazım)
zero_altitude: true -> gps yüksekliğini 0a sabitliyor.
broadcast_cartesian_transform: true -> node, gps verilerinden oluşturduğu kartezyen çerçeveyi(ilk fix etrafında) TF olarak yayınlar.
publish_filtered_gps: true -> filtrelenmiş gps mesajını yayınlar.
use_odometry_yaw: false -> gps'i dönüştürürken yaw için imu'yu kullanır. (ekf'nin odom yaw'ını değil.)
wait_for_datum: false -> bir datum beklemez. yani, ilk gps fix'i yerel orijin olarak alınır.

1.5.2 ara_gaz.py

ara_gaz_node adlı bir node oluşturuyor.
bu node, araca 2.5 saniye boyunca 1.5m/s ileri hız vererek aragaz yapıyor, bu sırada /drive_system/status'a AUTO ve /switch'e 2 gönderiyor.
ve bu node, /gnss_Data (Float64MultiArray) akışından gelen verilerdeki data[3] alanını kullanarak bir average cog hesaplıyor.
2.5s sonunda aracı durduruyor, /odom_start'a 1.0 yollayarak odometriye hazır olduğunu belirtiyor.

1.5.3 ublox_odom.py

odometri scripti.
lat=latitude, long=longitude, sog=speed-over-ground (m/s), cog=course-over-ground (heading in radians), and yaw=rotation around the vertical axis

abonelikler:
/gnss_data: GNSS verisi
/imu/data: IMU yönelimi (quaternion)
/drive_system/vcu_data: araç hareket yönü, string içerisinde

yayınlar:
/ublox/odom: odometri mesajı (konum + hız)
/gps/fix: NavSatFix (lat/long)
/initial_yaw: başlangıçta ölçülen yaw (IMU bazlı)
kod, TF ağacında odom ile base_link arasındaki dönüşümün hazır olmasını en fazla 5 sn bekler.

imu_cb -> imu'dan gelen quaternion euler'a dönüştürülür ve self.rpy güncellenir (roll, pitch, yaw)
bu yaw, başlangıç yaw'ına göre normalize edilerek odom çıktısına yazılacaktır
wheel_cb -> msg.data adlı string'in 8. karakteri 1 ise ileri, değilse geri sayılıyor

çalışmanın başında bir imu verisi beklenir (ara_gaz.py'dan), euler'e çevrilir ve /initial_yaw üzerinde yayınlanır
ileride oryantasyon hesaplanırken anlık yaw'dan bu başlangıç yaw'ı çıkarılarak görece yönelim üretilir. böylecek odom'daki yaw sıfır referanslı başlatılmış olur.

```
rot = self.listener.lookupTransform('/odom', '/base_link', rospy.Time(0))[1]
theta = tf.transformations.euler_from_quaternion((rot[0], rot[1], rot[2], rot
    [3]))[2]
```

TF ağacından odom -> base_link dönüşümünün quaternion kısmı alınır.
bu quaternion euler'e çevrilir; theta = yaw (radyan).

```
        self.pos.x += self.sog * cos(theta) * self.direction * dt
        self.pos.y += self.sog * sin(theta) * self.direction * dt
```

-> odom çerçevesinde konum güncellemesi yapılır.
hız büyüklüğü: self.sog
yön: theta (yaw, radyan)
ileri/geri: self.direction ile çarpılır
zaman: dt
ros rep-105'e uygun olarak x ekseni ileri, y ekseni sola varsayılır. cos(theta) x bileşeni,

sin(theta) y bileşenidir.
w: açısal hız hesabı

```
        [self.qua[0], self.qua[1], self.qua[2], self.qua[3]] =
            quaternion_from_euler(0, 0, (self.rpy[2]-initial_rpy[2]))
```

oryantaston; imu yaw - initial yaw olarak hesaplanır, roll ve pitch 0 kabul edilir. bu, odom'daki quaternion'u imu tabanlı yapar. konum integrasyonu ise tf tabanlı yaw ile yapılıyor.

```
        self.odom.header.frame_id = "odom"
        self.odom.child_frame_id = "base_link"
        self.odom.header.stamp = self.current
        self.odom.pose.pose = Pose(self.pos, Quaternion(self.qua[0], self.qua
            [1], self.qua[2], self.qua[3]))
        self.odom.twist.twist = Twist(Vector3(self.sog * cos(theta) * self.
            direction, self.sog * sin(theta) * self.direction, 0), Vector3(0,
            0, 0))
        self.odom_pub.publish(self.odom)
```

/ublox/odom mesajı hazırlanır ve yayınlanır.
frame_id: odom (pose bu çerçevede)
pose: konum self.pos, oryantasyon self.qua
twist.linear: odometri hız bileşenleri (theta yönünde büyüklük sog)
twist.angular:0 0 0 yollanıyor
lat/long gnss tarafından yayınlanır.

genel özet (gptden)
. başlangıçta /odom_start ve bir imu mesajı beklenir, ilk yaw yayınlanır.
. döngüde:
dt hesaplanır, tf'den odom -> base_link yaw (theta) değeri okunur
sog, theta ve yön ile x/y konumu entegre edilir
odom mesajına pozisyon ve imu'ya göre normalize yaw yazılır, lineer hız bileşenleri doldurulur, açısal hız 0 alınır
gnss lat/long /gps/fix olarak yayınlanır
zaman/cog güncellenir, hız sıfırlanır.

1.5.4 module_robot_state_publisher.launch

no_steering_urdf.urdf dosyasından parametre çeker.

1.5.4.1 no_steering_urdf.urdf roverin çerçevelerini (frames), bunlar arasındaki kinematik bağlantıları (joints), ayrıca görselleri, çarpışma şekillerini ve kütle/atalet (inertial)

bilgilerini tanımlar. konumlar metre, açılar radyan cinsindendir. eksen düzeni x ileri, y sol, z yukarı şeklindedir.
robot_state_publisher urdf'yi ve eklem durumlarını okup tf çerçeve ağacını yayınlar.

static_transform_publisher argümanları:
x y z yaw pitch roll parent_frame child_frame period_in_ms_or_rate

1.6 locomove.launch

navfn -> navigation function'un kısaltılmış hali, 2d costmap'te klasik dijkstra search ile global path-planning yapan bir plugin. final pose list nav_msgs/Path olarak /global_plan'da yayınlanır.

önemli parametreleri:
allow_unknown -> unknown (-1) olarak işaretlenen cell'erin üzerinden geçilip geçilemeyeceğini belirtmeyi sağlayan parametre
default_tolerance -> navfn'in durması için hedefe minimum yakınlığı belirten parametre

nav_core_adapter::GlobalPlannerAdapter -> move_base (ROS 1) expects nav_core plugins (nav_core::BaseGlobalPlanner). Newer planners—such as Dlux, Smac, Theta*—were written for the nav_core2 API.
nav_core_adapter::GlobalPlannerAdapter is a bridge: it lets you load any nav_core2 global planner while still presenting the old nav_core interface to move_base. In other words, it makes modern planners "look like" classic ones. The class description in the ROS docs states exactly that.

short version: use nav_core2 planners without migrating the rest of your stack.

1.6.1 planner.yaml

path-planning ve controller parametreleri
recovery_behavior_enabled: false –> disables move_base's classic spin-and-clear behaviours; you rely on the local planner's critics instead
global_plan_topic: global_plan -> global planın yayınlanacağı topic adı.
global_plan_type: Path3D -> global plan mesaj tipi.
global_plan_epsilon: -1.0 -> global plan tolerans parametresi. -1.0 varsayılan veya kapalı anlamına gelmekte, emin değilim
twist_topic: /nav_vel -> local planner'ın üreteceği hız komutlarının gideceği topic.
twist_type: Twist3D -> hız mesaj tipi.
recovery_behavior_enabled: false -> move_base'in klasik recovery behaviorlarını devre dışı bırakır. (rotate, clear costmap gibi)

1.6.1.1 navfn allow_unknown: true -> unknown (-1) olarak işaretlenen cell'erin üzerinden geçilip geçilemeyeceğini belirten parametre.
default_tolerance: 1.0 -> navfn'in durması için hedefe minimum yakınlığı belirten parametre.

1.6.1.2 DluxGlobalPlanner/ (A* tabanlı) neutral_cost: 50 -> A* potansiyel alanı için temel hücre maliyeti.
scale: 3.0 -> maliyet ölçekleme katsayısı.
unknown_interpretation: free -> unknown hücreleri free olarak yorumla. (serbest)
path_caching: false -> hesaplanan yollar önbelleğe alınmaz
improvement_threshold: -1.0 -> iyileştirme eşiği (A* tekrar/erken durma için) devre dışı.
publish_potential: true -> potansiyel alanı yayınla
potential_calculator: dlux_plugins::AStar -> heuristic hesaplayıcı plugin AStar.
traceback: dlux_plugind:GradientPath -> yol geri-izleyici

1.6.1.2.1 A* ayarları manhattan_heuristic: false -> heuristic olarak manhattan(dört yön) yerine euclidian kullan
use_kernel: true -> kernel optimizasyonları ve komşuluklarını kullanıyor
minimum_requeue_change: 1.0 -> bir düğümün tekrar kuyruğa alınması için gereken minimum maliyet değişimi

1.6.1.2.2 GradientPath ayarları step_size: 0.5 -> gradient izleme adım boyu (metre)
lethal_cost: 150.0 -> lethal (geçilemez) maliyet eşiği
iteration_factor: 4.0 -> geri-izleme yineleme katsayısı
grid_step_near_high: false -> true olsaydı high costların yakınında küçük adımlar atılıyor olurdu, burada uniform bir şekilde devam ediyor

1.6.1.3 LocalPlannerAdapter move_base'nin nav_core2 local planner'ini çağırmasını sağlayan bridge
planner_name: dwb_local_planner::DWBLocalPlanner -> DWBLocalPlanner'i çağırmanızı sağlayan underlying local planner class

1.6.1.4 DWBLocalPlanner update_costmap_before_planning: true -> trajectoryleri belirlemeden önce her cyclede costmap update forceler
prune_plan: true -> üstünden geçilen waypointleri global path tailden kaldırır
prune_distance: 1.0 -> üsttekinin gerçekleşmesi için gereken waypointe yakınlık mesafesi

short_circuit_trajectory_evaluation: true -> while scoring, stop evaluating a candidate as soon as its partial score already exceeds the best-so-far (big CPU saver).
trajectory_generator_name: dwb_plugins::StandardTrajectoryGenerator -> standard lattice sampler over velocity space.
goal_checker_name: dwb_plugins::SimpleGoalChecker -> uses the simple tolerance-based goal test.
xy_goal_tolerance: 0.8 -> accept goal when within 0.8 m in the plane (quite large; pairs with Navfn's 1.0 m).
yaw_goal_tolerance: 0.5 -> accept yaw within 0.5 rad 28.6°.
latch_xy_goal_tolerance: true -> once inside XY tolerance, keep it latched "true" while finishing yaw alignment (prevents flip-flop).
sim_time: 0.8 -> predict trajectories 0.8 s into the future.
linear_granularity: 0.05 -> place trajectory points every 0.05 m of linear travel (denser = costlier but more accurate).
sim_granularity: 0.025 -> minimum temporal/arc step between simulated points (further increases path resolution).
vx_samples: 20 / vy_samples: 0 / vtheta_samples: 20 -> sample 20 linear and 20 angular velocities (0 lateral) → 400** candidate trajectories per cycle.

critic list (evaluation order)
critics -> ordered stack; each critic adds to a total cost × its scale. Lowest total wins.
- PreferForward
- RotateToGoal (scale 0, disabled ama not commented out)
- Oscillation
- PathDist
- PathAlign
- GoalDist
- GoalAlign
- RotateBeforeStart
Notes: ObstacleFootprintCritic is commented out; obstacles are still accounted for implicitly via costmap footprint checks inside other critics/feasibility.

PreferForward: bias toward forward motion
backward_cost: 1.0 -> backwards gets penalized by 1.0.
forward_cost: 0.0
scale: 1

RotateToGoal: guide heading toward goal orientation during approach
scale: 0 -> bu critici disable'lar. parametrelerin etkisini görmek için yükseltmek gerek.

Oscillation: penalize stuck back-and-forth patterns.
x_only_threshold: -0.05 -> negative means treat small motions as potentially oscillatory

unless good progress appears.

oscillation_reset_dist: 0.1 / oscillation_reset_angle: 0.1 / oscillation_reset_time: 2 -> any of these resets the oscillation watchdog (move 10 cm, rotate 0.1 rad, or wait 2 s).

scale: 20.0 -> strongly penalizes oscillatory candidates.

PathDist: distance from the global path centerline.

scale: 48.0 -> high weight → hug the global plan closely.

GoalDist: encourages progress toward the goal.

scale: 25.0

PathAlign: align robot heading with path tangent.

scale: 64.0 -> very strong; prioritizes facing along the path.

forward_point_distance: 0.325 -> look-ahead distance (m) for heading alignment computation.

GoalAlign: align heading with goal orientation near the end.

scale: 32.0

RotateBeforeStart: enforce a pre-move orientation step.

scale: 100.0 -> extremely strong; robot will prefer to rotate to a good heading before translating.

1.6.1.5 costmap_common.yaml footprint_topic: "footprint" -> costmap listens on topic /footprint for a dynamic footprint (polygon). if your controller publishes a polygon here, it overrides the static radius polygon at runtime.
robot_radius: 1.5 -> sets a circular footprint of radius 1.5 m (used if no polygon footprint: is active). larger radius → safer but more conservative paths; smaller -> tighter clearance.
footprint_padding: 0.1 -> expands the robot footprint by 10 cm when checking collisions and computing cost inflation. adds safety margin on top of radius/polygon.
robot_base_frame: base_link -> base frame used by the costmap for footprint projection and TF lookups.
transform_tolerance: 1.0 -> how much TF latency/jitter (seconds) the costmap will tolerate. if TF is older than this, data may be rejected to avoid ghosting.
resolution: 0.1 -> grid cell size 0.10 m.
obstacle_range: 4 -> max range (meters) at which obstacles are marked from sensor returns. returns beyond 4 m are ignored for marking.
raytrace_range: 5 -> max range to clear free space by raytracing from the sensor origin. usually obstacle_range so you can clear farther than you mark.

Active obstacle layer (PointCloud2)

obstacles_laser: -> begins the ObstacleLayer configuration.
observation_sources: point_cloud_sensor -> declares one observation source named point_cloud_sensor for this layer.
point_cloud_sensor: -> opens the per-source settings.
data_type: PointCloud2, -> input is sensor_msgs/PointCloud2 (3D points).
clearing: true, -> enables clearing: free-space rays are traced from the sensor origin to each return (up to raytrace_range), clearing cells along the way.
min_obstacle_height: 0.1, -> rejects points below 0.1 m as obstacles (treats them as ground/ignored). useful to ignore near-ground noise.
marking: true, -> enables marking: any valid return within obstacle_range can mark obstacle cells.
topic: /ouster/points -> subscribes to /ouster/points (Ouster LiDAR point cloud). ensure the topic exists and TF for the sensor frame is valid.

inflation: -> begins InflationLayer settings shared by all costmaps that load this YAML.
inflation_radius: 1 -> cells within 1 m of an obstacle receive inflated costs that taper with distance. larger radius -> paths keep farther from walls; smaller -> riskier but shorter paths.

1.6.1.6 costmap_local.yaml global_frame: odom -> the local costmap is expressed in the odom frame. this makes the map continuous (no jumps when localization corrects the global pose) and is standard for local avoidance.
rolling_window: true -> the grid slides with the robot, centering around it. only the nearby 20×20 m area is stored and updated.
width: 20 -> total width of the local costmap in meters. at 20 m, you cover ±10 m around the robot.
height: 20 -> total height in meters. with width, this defines the local field of view for planning.
resolution: 0.1 -> grid cell size = 0.10 m. with 20×20 m, that's 200 × 200 = 40,000 cells per layer.

plugins: -> ordered list of Costmap2D layers. they are combined to produce the final costmap used by the local planner.

- name: obstacles_laser, type: "costmap_2d::ObstacleLayer" -> adds an ObstacleLayer named obstacles_laser.
it relies on observation_sources defined in costmap_common.yaml under the local_costmap namespace (point_cloud_sensor on /ouster/points, with marking: true, clearing: true, min_obstacle_height: 0.1).
this layer writes LETHAL and FREE cells based on incoming sensor data and obstacle_range / raytrace_range.

- name: inflation, type: "costmap_2d::InflationLayer" -> adds InflationLayer named inflation.
uses the inflation: settings from costmap_common.yaml (inflation_radius: 1).
produces a graded cost buffer around obstacles so the local planner prefers the center of free corridors.

1.6.1.7 costmap_global.yaml global_frame: odom -> the global costmap is expressed in odom. that makes it drift-safe (no sudden jumps from map corrections), but it's not truly global over long distances—odometry drift will slowly shift where obstacles "are" relative to a fixed world. most stacks use map for the global costmap; using odom here is intentional only if you don't have a reliable global map frame. (Muhammed'in belirttiği drift buradan, burayı incele)
rolling_window: true -> the global grid slides with the robot, just like a local map—but at a much larger size. this creates a very large "moving world" rather than a fixed global map.
track_unknown_space: false -> do not track unknown cells; treat them as free (0) instead of NO_INFORMATION (1). implication: the planner can freely route through areas you haven't seen yet. if you want conservative behavior ("plan only through known free space"), set this to true.
map width is 300m and map height is 300m.
resolution: 0.1 -> grid cell size 0.10 m. matches your local costmap, which avoids interpolation when plans cross between maps.

layer stack:
plugins: ordered list of Costmap2D layers that combine into the master grid.

- name: obstacles_laser, type: "costmap_2d::ObstacleLayer" -> adds an ObstacleLayer named obstacles_laser. it will use the observation_sources defined in costmap_common.yaml under the global_costmap namespace (point_cloud_sensor on /ouster/points, with marking/clearing). because track_unknown_space: false, everything starts as free; marking will carve obstacles where the sensor sees them, and clearing will ray-erase free space along beams.

- name: inflation, type: "costmap_2d::InflationLayer" -> adds InflationLayer named inflation. uses the "inflation:" block from costmap_common.yaml (you set inflation_radius: 1). this grows costs around obstacles so global plans prefer corridor centers.

note: There is no StaticLayer here. That means this "global" map is sensor-only and sliding, not a fused, fixed occupancy map. it's a valid design for exploration or GPS-denied setups without a global mapper, but different from the usual "map+static layer" pattern.

1.7 aruco_approach.py

ne yaptığı isimde yazıyor

subscriber:
/konum/taha nereden publishleniyor-> arucoSheesh.py
/ublox/odom -> to keep track of the robots current position
/locomove_base/is_reached -> done_cb

publisher:
/move_base_simple/goal (type PoseStamped) -> for setting a goal for navigation stacks
/drive_system/status -> AUTO, SUCCESS publishlemek için
/switch -> otonom için olması lazım

self.aruco_listener = tf.TransformListener() -> instantiates a TF TransformListener (to listen to TF frames).
self.aruco = Point() -> holds the relative offset received from /konum/taha. Defaults to 0,0,0.
self.toggle = 1 -> a one-shot latch: when 1, the node will send one goal; then set to 0 to prevent re-sending repeatedly.
self.done = 0 -> a completion flag to ensure success actions run only once.
def pos_cb(self, msg:Twist): -> callback for /konum/taha. receives a Twist carrying relative offsets.
self.goalpos.x = self.pos.x + self.aruco.x -> computes absolute goal x = current x + relative x offset.
self.goalpos.y = self.pos.y + self.aruco.y -> computes absolute goal y = current y + relative y offset.
self.pos = msg.pose.pose.position -> updates current position (Point) from odometry.
def set_goal(self, x, y): -> helper to build and publish a PoseStamped goal at (x,y) with yaw facing the target.
yaw = atan2(y-self.pos.y, x-self.pos.x) -> computes the heading (yaw) from current position to the target point using atan2(dy, dx).
goal_quaternion = tf.transformations.quaternion_from_euler(0, 0, yaw) -> converts roll=0, pitch=0, yaw into a quaternion.
self.pos_pub.publish(goal) -> publishes the goal to /move_base_simple/goal.
self.status_pub.publish('AUTO') -> publishes AUTO status to indicate autonomous movement is active.
if (not (self.aruco.x == 0 and self.aruco.y == 0)) and self.toggle: -> if an offset has been received (not both zero) and we haven't sent a goal yet (toggle==1), then proceed. (this prevents sending a meaningless zero-offset goal and ensures a one-shot behavior.)
self.switch_pub.publish(Int32(1)) -> sends 1 on /switch
self.set_goal(self.goalpos.x, self.goalpos.y) -> publishes the absolute goal computed from current pose + aruco offset.

self.toggle = 0 -> latch off so it doesnt keep publishing the goal every cycle.

1.8 gps_goal_controller.py

başlangıçta rosrun gps_goal gps_goal_controller.py –lat *** –long *** –aruco(eğer arucoya gidiliyorsa)
x1 kuzey, y1 batı, x2 roverin önü, y2 roverin solu
o'lar kontrolcülerin araca belirttiği ara hedef konum, 1ler odoma göre anlık konumun, 2ler odom frame'inde hedef

genel yapılan (özet misali):
convert the target lat/long (supports DMS like 43,39,31 or decimal like 43.658) to decimal degrees.
Read a published local XY origin (/local_xy_origin) that maps GPS to your local metric frame (meters).
geodesic math turns the GPS delta into a local (x,y) target (meters) using WGS-84.
face the goal (coarse alignment), then "chase" a moving setpoint using a PI-like controller that publishes pose goals in the odom frame to /move_base_simple/goal. (zıpzıp kontrol)
monitor for oscillations and "goal reached," then publish status/switching flags to other nodes.

abonelikler: /ublox/odom (Odometry) -> pose ve twist
/gps/fix (NavSatFix) -> current GPS fix
/nav_vel (Twist) -> navigation command feedback (used to detect rotating in place)
/konum/taha (Twist) -> sets self.stop=1
/local_xy_origin (PoseStamped) -> lat(->x) and long(->y) of the local origin
/odom_start (Float32) -> gating flag (başlamak için onay gelmesini bekliyor)
/initial_cog (Float32) -> initial cog in degrees

yayıncılar:
/drive_system/status (String) -> e.g. AUTO ve SUCCESS
/spiral_start [Float32(1.0)] -> spiral search start command
/drive_system/twist (Twist) -> used to rotate for alignment
/move_base_simple/goal (PoseStamped) -> the immediate goal the rover chases
/switch (Int32) -> 1 while aligning, 0 when done

def DMS_to_decimal_format(lat,long) -> if a comma is present, it treats the string as "deg,minutes,seconds". it negates minutes/seconds if the first character is -. returns floats and logs the normalized goal.
def get_origin_lat_long(): -> what it does is commented into the code. the system upstream must be publishing a GPS origin as x=lat, y=long each time you call this. you block each time you call this (it waits for a fresh message).

def calc_goal(self, origin_lat, origin_long, goal_lat, goal_long): -> computes great-circle distance and initial azimuth between origin and goal. converts that to a 2D offset (x1,y1) in meters. applies a rotation by self.cog (course-over-ground) to align with robot's reference. returns goal offset relative to the origin, in the local metric frame.
geod = Geodesic.WGS84 -> defines the earth model to be used for calculations
g['s12'] -> the true distance in meters (hypotenuse).
g['azi1'] -> the initial forward azimuth (bearing) in degrees from the origin to the goal
the rest of the function converts this polar coordinate (distance and angle) into a cartesian coordinate (x, y) in the local map frame, correcting for the initial orientation (self.cog - course over ground) of the map frame itself relative to true north.

def do_gps_goal(self, goal_lat, goal_long, z=0, yaw=0, roll=0, pitch=0): -> the main control loop of the entire script
initialization: it sets up the matplotlib plots for real-time visualization and waits for the first odometry and "course over ground" messages to get a reliable starting pose and heading.
calculate initial goal: it does an initial calculation to find the goal's (x, y) position in the odom frame.
controller setup: it initializes PI controller variables (I_x, I_y for the integral term) and offsets.
alignment: it calls self.align_to_goal(...) to first rotate the robot to face the target before it starts moving forward.
while (not rospy.is_shutdown()) loop:
recalculate goal: the x_2, y_2 goal is recalculated in every loop using the most recent GPS reading as the origin. this makes the system adaptive to drift.
error calculation: e_x = x_2 - self.pos.x and e_y = y_2 - self.pos.y. this is the "P" (Proportional) part—the current error vector.
integral calculation: I_x += e_x * dt. This accumulates the error over time. Its purpose is to overcome steady-state errors (e.g., if the robot is pushed off course by wind or a slope, the accumulating integral term will eventually create a strong enough correction to push it back on track).
controller output: x_o = (e_x * Kp) + (K_i_x * I_x) .... This line calculates the intermediate goal point. This is the output of the PI controller.
publish goal: self.publish_goal(x=(x_o), y=(y_o), ...) sends this intermediate point to move_base.
anti-windup: The if ((abs(x_o - self.pos.x) <= 2)) ... blocks are a crucial feature called "integral anti-windup". If the integral term I_x grows too large, it can cause a massive overshoot. This logic resets the integral (I_x = 0) when the robot is close to the intermediate goal, preventing the overshoot.
oscillation detection: it calls self.detect_sign_oscillation to check if the robot is weaving back and forth. if so, it can trigger an override to stop.
exit condition: the loop breaks when the robot is within a very small tolerance of the goal

(<= 0.000008 degrees, or about 0.88 meters) or if the oscillation override is triggered. final actions: once the goal is reached, it stops the robot, publishes a message to /spiral_start (to start ss.py), and shuts down the plotting.

@click.command()
@click.option('–lat', prompt='Latitude', help='Latitude')
@click.option('–long', prompt='Longitude', help='Longitude')
@click.option('–aruco', help='Aruco', default=0.0)
@click.option('–roll', '-r', help='Set target roll for goal', default=0.0)
@click.option('–pitch', '-p', help='Set target pitch for goal', default=0.0)
@click.option('–yaw', '-y', help='Set target yaw for goal', default=0.0)
def cli_main(lat, long, aruco, roll, pitch, yaw):
Send goal to move_base given latitude and longitude

Two usage formats:

```
Two usage formats:
       gps_goal_controller.py --lat 43.658 --long -79.379 # decimal format        yakin       rosrun gps_goal gps_goal_controller.py --lat 39.6628797 --long -110.8568183
       gps_goal_controller.py --lat 43,39,31 --long -79,22,45 # DMS format        uzak        rosrun gps_goal gps_goal_controller.py --lat 39.6628976 --long -110.857000
                                                                                  aruco       rosrun gps_goal gps_goal_controller.py --lat 39.6629092 --long -110.8568063 --aruco "1"
'
```

yardımcı olabilecek:
0 1.0 111 km
1 0.1 11.1 km
2 0.01 1.11 km
3 0.001 111 m
4 0.0001 11.1 m
5 0.00001 1.11 m
6 0.000001 111 mm
7 0.0000001 11.1 mm
8 0.00000001 1.11 mm

1.9 publish_origin.py

it subscribes to a stream of GPS data and for each incoming message, it creates and publishes a new message that establishes a local coordinate frame's origin.

abonelikler:
/gps/fix (NavSatFix)

yayıncılar:
/local_xy_origin (PoseStamped) -> where you publish the origin

1.10 move_base_origin.py

defines the MoveBaseSpiral class, which serves as the central hub for the rover's state, sensory information, and action capabilities. an instance of this class is used as a global node object throughout the application, providing a shared interface for the various behaviors in the behavior tree. basically the backend for ss.py.

class MoveBaseSpiral: -> this class encapsulates all the data and methods required for the rover's navigation logic used in the spiral search algorithm..

abonelikler:
/ublox/odom
/fiducial_transforms (FiducialTransformArray)
/ublox/fix
/konum/taha

initializes tf2_ros.Buffer and tf2_ros.TransformListener for managing coordinate transformations.
tf.TransformBroadcaster is created to publish coordinate frames, such as the calculated midpoint of a gate.

66-69 / ros services:
mission -> to start a mission with a target x,y coordinate and pole IDs.
gps_goal ve gps_goal_fix -> to start a mission with a target lat/long and pole IDs.
stop_mission

yayıncılar:
/spiral_markers -> to visualize search points in rviz
/e_stop -> emergency stop
/move_base/cancel -> to cancel the current navigation goal
/rover_velocity_controller/cmd_vel -> to send direct velocity commands for actions like rotation
/navigation_status -> to publish readable status updates

80-88 / action client and initialization:
creates a SimpleActionClient to communicate with the move_base action server.
self.move_base.wait_for_server() -> blocks until a connection to the move_base server is established
rospy.wait_for_message("/spiral_start", Float32) -> pauses execution until a message is received on this topic
rospy.Publisher("/switch", Int32, queue_size=10) -> a publisher for the /switch topic is cre-

ated here

def gps_changed(self):
a method to check a value on the py_trees.blackboard. the blackboard is a key-value store used for communication between different behaviors in the tree. this method checks if a new GPS goal has been set.

def pos_cb(self, msg):
the callback for /konum/taha subscriber. when a message is received on this topic it effectively stops the rover's current movement.

def mission_completed(self):
this method is called when a mission is successfully finished. it clears the GPS goal from the blackboard, deletes all visual markers and publishes a mission completed status.

def reset(self, stop=False):
resets the rover's state to prepare for a new mission. it clears all data structures, resets counters, and re-initializes blackboard variables. if stop=True, it sets a flag to indicate the mission was stopped manually.

def begin_mission(self):
calls reset() and logs that a new mission is starting.

def handle_stop_service(self, data):
the callback for the stop_mission service. if the request data is True, it resets the rover's state and calls self.stop() to halt movement.

def handle_mission_service(self, req):
the callback for the mission service (x, y goals). it starts a new mission, sets the target (x, y) coordinates on the blackboard, and stores the list of target pole IDs.

def handle_gps_goal_service(self, req):
the callback for the gps_goal service (latitude, longitude goals). it starts a new mission, converts the incoming lat/long to local x/y coordinates using calc_gps_goal, sets the result on the blackboard, and stores the pole IDs.

def calc_goal(self, curr_lat, curr_long, goal_lat, goal_long):
a utility function that uses the geographiclib library to calculate the forward geodesic problem: given a starting lat/long, a goal lat/long, it computes the distance (hypotenuse) and initial bearing (azimuth). it then converts these polar coordinates into cartesian (x, y) offsets.

def do_gps_goal(self, goal_lat, goal_long, z=0, yaw=0, roll=0, pitch=0):

this method calculates the x and y offsets from the current GPS position to a target GPS position and then calls publish_goal to send the robot to the resulting absolute coordinate in the odom frame.

def handle_gps_goal_fix_callback(self, data):
the callback for the gps_goal_fix service. this implements a more robust navigation loop for GPS goals. it repeatedly calls do_gps_goal to send updated goals to move_base. the frequency of these updates changes as the rover gets closer to the destination to avoid recovery behaviors. the loop continues until the rover is within a 10-meter tolerance of the goal.

def publish_goal(self, x=0, y=0, z=0, yaw=0, roll=0, pitch=0):
a method for sending a goal to move_base. it takes x, y, z, and orientation as input, packages them into a MoveBaseGoal message, and sends it via the action client. it also logs the status of the goal execution.

def calc_gps_goal(self, goal_lat, goal_long):
this method is specifically for converting a target latitude and longitude into x and y coordinates relative to the rover's starting GPS position (self.start_point_lat, self.start_point_lon). the calculated x and y are the goal's position in the local odom frame, assuming the rover started at (0,0) in that frame.

def abort_goal(self):
publishes an empty GoalID message to the /move_base/cancel topic, which instructs move_base to cancel its current goal.

def create_points(self, start, end, angle_step,init = False):
generates a series of points in a spiral pattern. the spiral is defined by the formula x = (r+1) * cos(theta)/2 and y = (r+1) * sin(theta)/2, where r increases iteratively and theta is a function of r. these points are relative to the rover's current position when the method is called. it then calls create_markers to visualize these points.

def delete_all_markers(self):
publishes a marker with the DELETEALL action to clear all previously published markers from rviz.

def create_markers(self, arr_circle):
this method populates a MarkerArray message to visualize points in rviz. it can visualize either the spiral search points or the circular search points around a gate. it sets the properties of each marker (shape, size, color, position) and adds it to the self.markers array.

def find_shortest_index(self):
when searching for a second gate post, the rover moves along a circle of points. this method

calculates which of these circle points is closest to the rover's current position to start the search efficiently.

def calculate_circle_points(self, r, center_x, center_y):
generates a set of four points in a circle around a given center point (the location of the first detected gate post). it then calls find_shortest_index to determine the best starting point for the search and create_markers to visualize them.

def datapos(self,data):
the callback for the odometry subscriber. it continuously updates the rover's current x, y position and yaw (self.curr_x, self.curr_y, self.yaw). on the very first call, it records the starting position (self.start_point_x, self.start_point_y), which serves as the origin for the spiral search pattern.

def gps_cb(self,data):
the callback for the GPS subscriber. it updates the rover's current latitude and longitude. on the first call, it records the starting GPS coordinates, which are used as the origin for converting subsequent GPS goals into the local frame.

def rotate(self, angle):
publishes a Twist message to the velocity command topic. it sets the linear velocities to zero and the angular z velocity to the provided angle value, causing the rover to rotate in place.

def publish_markers(self):
publishes the self.markers array to the /spiral_markers topic for visualization.

def fiducial_cb(self,fiducials):
the callback for the /fiducial_transforms subscriber. this method is crucial for perception.
it first re-broadcasts the transforms of any previously saved gate coordinates. This is done to keep their positions available in the TF tree.
it then iterates through the newly detected fiducial markers. For each marker that hasn't already been visited, it uses the tf_buffer to look up its transform from its own frame (e.g., fiducial_14) to the fixed odom frame.
the transformed coordinates (x, y, z, and quaternion) are then stored in the self.detected_posts dictionary, with the fiducial ID as the key.

def get_goal(self, x, y, rotate_to_center = False):
a helper method that constructs a MoveBaseGoal message. given target x and y coordinates, it calculates the required yaw to face the goal from the rover's current position. it then populates the goal message with the position and the calculated orientation (as a quaternion) and returns it. it has an option rotate_to_center which is used when navigating the circle points around a gate.

def stop(self, time):
a method to stop the rover's movement. it publishes True to the /e_stop topic, waits for a specified time, and then publishes False to release the stop.

def update_gate(self):
this method checks the self.detected_posts dictionary. if any of the detected posts have IDs matching the target gate post IDs (self.gate1_id, self.gate2_id), their information is copied into the self.gate_coordinates dictionary. this isolates the gate posts from other detected posts.

1.11 ss.py

the spiral search algorithm.

from node_config import node, node_config -> this is a crucial import. it imports the globally accessible node object (an instance of the MoveBaseSpiral class) and a node_config object.

behaviours:
from points_available import PointsAvailable
from post_detected import PostDetected
from one_gate_detected import OneGateDetected
from gate_detected import GateDetected
from all_posts_visited import AllPostsVisited

decorators:
from decorators import RepeatDecorator, OscillationDetected, GetPathTimeout

rotate right left:
from rotate_left_right import RotateLeft, RotateRight

actions:
from get_path import GetPath
from go_post import GoPost
from go_gate import GoGate
from go_circle_point import GoCirclePoint
from go_gps import GoGPS

def create_root():
this function is responsible for building the entire behavior tree.

bt_root = py_trees.composites.Parallel("Behavior Tree Root"): the root of the tree is a Parallel node. this means its children will be ticked (executed) concurrently. a Parallel node's success/failure policy determines its own status based on its children's statuses.

nav_node = py_trees.composites.Selector("Autonomous Navigation"): a Selector node (also known as a fallback or OR node) is created. it ticks its children from left to right and stops as soon as one returns SUCCESS or RUNNING. it only returns FAILURE if all its children fail. This structure defines the priority of actions.

osc_detected = OscillationDetected("Oscillation Detected?"): instantiates the (currently placeholder) behavior to detect oscillation.

get_goal, post_control, gate_control: these are Sequence nodes. a Sequence (or AND node) ticks its children from left to right and stops as soon as one returns FAILURE or RUNNING. it only returns SUCCESS if all its children succeed.

GetPath and GetPathTimeout: An instance of the GetPath action is created to get a spiral search point. This action is then wrapped in a GetPathTimeout decorator. This decorator will cause the GetPath action to fail if it runs for more than 12 seconds, preventing the rover from getting stuck.

RepeatDecorator: this custom decorator is used to repeat a child behavior a specific number of times.

RotateRight and RotateLeft: instances of the simple rotation behaviors.

rotate_post and rotate_gate: two Sequence nodes are created to define rotation patterns (e.g., rotate right 20 times, then left 40 times) used when searching for a post or a gate.

py_trees.blackboard.Blackboard().set(...): The blackboard is initialized with several key-value pairs that control the state of the system, such as the number of remaining_points for the spiral search and flags like post_detected.

repeat_gate = RepeatDecorator(...): a RepeatDecorator is set up to wrap the go_gate action. this will make the rover execute the go_gate behavior 3 times, which corresponds to navigating to the back, mid, and front points of a detected gate.

170-177. satırlar: tree construction
bt_root.add_children([nav_node,osc_detected]): the Parallel root has two children: the main navigation logic and the oscillation detector. they run concurrently.

nav_node.add_children([go_gps, all_posts_visited,gate_control,post_control,get_goal]):

the Selector nav_node defines the core logic priority:
go_gps: if a GPS goal is active, handle it first.
all_posts_visited: if all posts are visited, succeed.
gate_control: if a gate is partially or fully detected, try to handle it.
post_control: if a single post is detected, try to go to it.
get_goal: if none of the above are true, get the next spiral search point.

get_goal.add_children([check_points, new_goal_timeout]): the get_goal Sequence: first check if Points Available?, then execute Get Path (with a timeout).
post_control.add_children([post_detected, rotate_post, go_post]): the post_control Sequence: check if a Post Detected?, if so, Rotate for Post to face it, then Go to Post.

gate_control.add_children([one_gate_detected, rotate_gate, gate_detected, repeat_gate]): the gate_control Sequence:
check if one gate has been detected.
if so, Rotate for Gate to find the second one.
then check if a full Gate has been detected.
if so, execute the Go to Gate action three times to pass through it.

def pos_cb(msg): this is the callback for the /konum/taha topic. It calls the shutdown function.

if __name__ == '__main__': block:
rospy.Subscriber("/konum/taha", Twist, pos_cb) -> subscribes to the topic /konum/taha

node.create_points(0,25,math.pi/4,True) -> initializes the spiral search points
node.publish_markers(): -> Visualizes the initial points.

root = create_root(): Builds the behavior tree structure.
behaviour_tree = py_trees_ros.trees.BehaviourTree(root): -> creates a py_trees_ros wrapper for the tree, which handles the ROS integration (e.g., spinning).
behaviour_tree.setup(timeout=15): initializes the tree and all its nodes.
behaviour_tree.tick_tock(node_config.TICK_TOCK): starts the main execution loop of the behavior tree. the tree will be "ticked" at a regular interval defined by node_config.TICK_TOCK, causing the behaviors to be executed repeatedly.

1.11.1 points_available.py

this file defines a simple condition behavior that checks if there are any spiral search points left to visit.

1.11.2 post_detected.py

this file defines a condition behavior that checks if a new, unvisited post has been detected.

if len(node.poles) == 2: -> if the current mission is to find a gate (which involves two poles), this behavior immediately fails. This ensures that the post_control sequence doesn't run when the rover should be in gate_control mode.
if node.gps_changed(): -> if a new GPS goal has been issued, it fails, allowing the go_gps behavior to take priority.
if len(node.detected_posts) > 0:: Checks if any fiducial markers are currently detected.
node.stop(1.): If posts are detected, it momentarily stops the rover.
for goal in list(node.detected_posts.items()):: it iterates through all currently detected posts.
if goal[0] not in node.gate_coordinates.keys() and goal[0] in node.poles:: this is the core logic. it checks if the detected post (goal[0] is the ID) is one of the target poles for the current mission (in node.poles) and that it is not a post that has already been identified as part of a gate (not in node.gate_coordinates.keys()).
py_trees.blackboard.Blackboard().set("post_goal", goal): if a valid target post is found, its information (goal, which is a (key, value) tuple of (id, coordinates)) is written to the blackboard.
return py_trees.common.Status.SUCCESS: the behavior succeeds, allowing the parent Sequence (post_control) to proceed to the next step (rotating and moving to the post).
if the loop finishes without finding a valid target post, or if no posts were detected initially, the method returns py_trees.common.Status.FAILURE.

1.11.3 one_gate_detected.py

this behavior checks if one of the two target gate posts has been detected. If so, it prepares for the search for the second post.

if node.gps_changed(): -> fails if a new GPS goal is active.
node.update_gate(): calls the helper method to update the gate_coordinates dictionary with any newly detected gate posts.
if len(node.gate_coordinates) > 0: -> checks if at least one of the target gate posts is now in the gate_coordinates dictionary.
first_pole = list(node.gate_coordinates.items())[0]: gets the information of the first detected gate post.
node.calculate_circle_points(node_config.CIRCLE_RADIUS, first_pole[1][0], first_pole[1][1]): calls the method to generate a set of search points in a circle around the detected post. the radius is taken from node_config.
node.publish_markers(): visualizes the new circular search points.
py_trees.blackboard.Blackboard().set("circle_point_goal", node.circle_points[node.shortest_index]): writes the closest circle point (as determined by calculate_circle_points) to the

blackboard as the next navigation goal.
return py_trees.common.Status.SUCCESS: succeeds, allowing the parent gate_control sequence to proceed.

1.11.4 gate_detected.py

this behavior checks if both target gate posts have been detected. if so, it calculates the points needed to navigate through the gate.

if node.gps_changed():: fails if a new GPS goal is active.
if len(node.gate_coordinates) == 2:: the main condition. it checks if both gate posts have been found and their coordinates are stored.
it calculates three goal points relative to the gate's midpoint and orientation:
goal_back: A point 4 meters "behind" the gate.
goal_mid: The midpoint of the gate itself.
goal_front: A point 4 meters "in front" of the gate.

it also broadcasts TF frames for these points for visualization.

py_trees.blackboard.Blackboard().set("gate_goals",[goal_back,goal_mid,goal_front]): it writes the list containing these three goal points to the blackboard.
return py_trees.common.Status.SUCCESS: The behavior succeeds, indicating that the go_gate action can now proceed.
if fewer than two gate posts are detected, it returns FAILURE.

1.11.5 all_posts_visited.py

a simple condition that checks if the total number of visited posts has reached a predefined number.

post_count = node.post_counter: gets the number of visited posts from the global node object.
if post_count == node_config.POST_NUMBER:: compares the current count to a configured total number of posts.
if they are equal, it logs a success message and returns SUCCESS. This would cause the main nav_node Selector to succeed, effectively ending the mission. otherwise, it returns FAILURE.

1.11.6 decorators.py

this file defines custom decorators that modify the behavior of their child nodes.

RepeatDecorator:
inherits from py_trees.decorators.Decorator. this decorator is designed to make its child run until it has succeeded a specified number of times.

1.11.7 rotate_left_right.py

this file defines two very simple behaviors for rotating the rover.

1.11.8 get_path.py

this file defines an action client behavior that sends a goal to move_base to navigate to the next spiral search point.

1.11.9 go_post.py

this action client behavior navigates the rover to a detected post.

1.11.10 go_gate.py

this action client behavior navigates the rover through a detected gate by visiting the three pre-calculated points (back, mid, front).

1.11.11 go_circle_point.py

this action client is used when searching for the second gate post. it navigates to one of the points on the search circle.

1.11.12 go_gps.py

this action client navigates the rover to a specific x,y coordinate goal, received from the gps.

1.11.13 node_config.py

the big picture: what node_config.py does

defines a "Settings Menu": it creates a NodeConfig class that acts like a settings menu or control panel for the robot's behavior.

sets the Default Values: it defines the default values for these settings, such as the robot's rotation speed and the number of posts in the mission.

initializes the ROS Node: it performs the fundamental ROS startup command, rospy.init_node(), which officially connects this program to the larger ROS network.

Creates the Global "Brain": it creates the single, global instance of the MoveBaseSpiral class, naming it node. This node object is then imported and used by almost every other file to access state, publishers, and helper methods.

2 Otonom sürüşü başlatma

2.1 Sıra

2.1.0.1 roslaunch rover23_localization lidar.launch lidar / pointcloud
hep açık

2.1.0.2 roslaunch rover23_localization sbg_device.launch imu / 6 eksen
hep açık

2.1.0.3 rosrun rover23_localization udp.py udp server açıyor / gnss verisi alıyor
hep açık

2.1.0.4 roslaunch rover_23_control control_udp.launch twist mesajlarını birleştirir vcuya yollar
hep açık

2.1.0.5 roslaunch rover23_localization ublox_odom.launch odometri
hep açık, her varılan hedefte sıfırla

2.1.0.6 roslaunch locomove_base locomove.launch pathplanning, rvizden yol göndermeyi de sağlıyor
hep açık

2.1.0.7 rosrun rover23_localization aruco_approach.py aruco

2.1.0.8 rosrun gps_goal gps_goal_controller.py –lat –long –aruco(opsiyonel)

asıl hedefe gitme, dokümantasyonda kullanım şekilleri bulunuyor.

2.1.0.9 rosrun gps_goal publish_origin.py origin lat/long | gnss topic

2.1.0.10 rosrun auto_nav ss.py spiral search