Arjuna Gazebo Simulation — Complete Guide (v2)

ros_sim Workspace | ROS Noetic | Ubuntu 20.04

LIDAR · Camera · IMU · Teleop · Obstacle Avoidance · SLAM · Navigation

---

WORKSPACE STRUCTURE

```
ros_sim/
└── src/
    └── arjuna_gazebo/
        ├── package.xml
        ├── CMakeLists.txt
        ├── urdf/
        │   └── arjuna_sim.urdf          ← robot model + all sensor plugins
        ├── worlds/
        │   └── arjuna_world.world       ← 10×10 m room + 4 obstacle boxes
        ├── launch/
        │   ├── arjuna_sim.launch        ← Gazebo + robot + all sensors
        │   ├── teleop.launch            ← keyboard control (standard)
        │   ├── obstacle_avoidance.launch ← autonomous obstacle avoidance
        │   ├── slam_mapping.launch      ← build a map while driving
        │   └── navigation.launch        ← autonomous point-to-point nav
        ├── config/
        │   ├── costmap_common_params.yaml    ← shared costmap settings
        │   ├── global_costmap_params.yaml    ← global planner costmap
        │   ├── local_costmap_params.yaml     ← local planner costmap
        │   ├── base_local_planner_params.yaml ← DWA planner speeds
        │   ├── move_base_params.yaml         ← move_base general settings
        │   └── amcl_params.yaml              ← localisation settings
        ├── maps/
        │   └── README.txt               ← save your maps here after SLAM
        └── scripts/
            ├── teleop_key.py            ← custom keyboard teleop (w/a/s/d)
            ├── obstacle_avoidance.py    ← LIDAR-based obstacle avoidance
            ├── go_to_point.py           ← navigate to one point
            └── multi_point_nav.py       ← navigate through multiple waypoints
```

PART 1 — INSTALLATION

Step 1 — Install All Required Packages

```
sudo apt update && sudo apt upgrade -y
```

```
sudo apt install -y \
  ros-noetic-gazebo-ros-pkgs \
  ros-noetic-gazebo-ros-control \
  ros-noetic-robot-state-publisher \
  ros-noetic-joint-state-publisher \
  ros-noetic-teleop-twist-keyboard \
  ros-noetic-gmapping \
  ros-noetic-map-server \
  ros-noetic-amcl \
  ros-noetic-move-base \
  ros-noetic-dwa-local-planner \
  ros-noetic-navfn \
  ros-noetic-costmap-2d \
  ros-noetic-nav-core \
  ros-noetic-move-base-msgs \
  ros-noetic-actionlib \
  xterm
```

Installation Table

Package	Why It Is Needed
`ros-noetic-gazebo-ros-pkgs`	Core Gazebo-ROS integration. Without this Gazebo cannot publish or subscribe to any ROS topic.
`ros-noetic-gazebo-ros-control`	Allows Gazebo plugins to control robot joints via `/cmd_vel` commands.
`ros-noetic-robot-state-publisher`	Reads URDF and publishes TF coordinate frame transforms for every robot link.
`ros-noetic-joint-state-publisher`	Publishes wheel joint positions. Required for TF and robot model rendering.
`ros-noetic-teleop-twist-keyboard`	Standard keyboard teleoperation package.

Package	Why It Is Needed
`ros-noetic-gmapping`	SLAM mapping algorithm. Builds a 2D occupancy grid map from LIDAR and odometry.
`ros-noetic-map-server`	Loads a saved map file and publishes it on `/map`. Used in navigation.
`ros-noetic-amcl`	Adaptive Monte Carlo Localisation. Locates the robot on a saved map using particle filters.
`ros-noetic-move-base`	The ROS Navigation Stack core. Handles path planning, costmaps, and sends velocity commands to drive the robot to a goal.
`ros-noetic-dwa-local-planner`	Dynamic Window Approach local planner. Computes safe velocity commands in real time to follow the global path.
`ros-noetic-navfn`	Global path planner (Dijkstra algorithm). Finds the shortest path from current position to the goal on the map.
`ros-noetic-costmap-2d`	Costmap library. Converts sensor data and map data into a grid showing where the robot can safely drive.
`ros-noetic-nav-core`	Common interfaces (base classes) for all navigation planners. Required by move_base.
`ros-noetic-move-base-msgs`	Message types for sending goals to move_base (`MoveBaseAction`, `MoveBaseGoal`).
`ros-noetic-actionlib`	Action library for sending long-running goals with feedback. Used by go_to_point.py and multi_point_nav.py.
`xterm`	Opens a separate terminal window for teleop. Required by `teleop.launch`.

Step 2 — Unzip and Build

```
unzip ros_sim.zip
cd ros_sim
catkin_make
```

Expected output ends with:

```
[100%] Built target arjuna_gazebo_generate_messages
```

If errors:

```
rosdep install --from-paths src --ignore-src -r -y
catkin_make
```

Step 3 — Source the Workspace

Add to `.bashrc` so every terminal has it automatically:

```
echo "source ~/ros_sim/devel/setup.bash" >> ~/.bashrc
source ~/.bashrc
```

Or source manually in each terminal:

```
source ~/ros_sim/devel/setup.bash
```

PART 2 — FILE-BY-FILE CODE EXPLANATION

FILE 1 — package.xml

Purpose: Tells ROS the package name, version, and every other ROS package it depends on.

Tag	Explanation
`<n>arjuna_gazebo</n>`	Package name. Must match the folder name exactly. Used in all `roslaunch arjuna_gazebo ...` commands.
`<version>2.0.0</version>`	Version number. Updated to 2.0 to reflect SLAM and navigation additions.
`<buildtool_depend>catkin</buildtool_depend>`	Build system — catkin is the standard ROS build tool. Required in every ROS package.
`<depend>rospy</depend>`	Python ROS client library. Needed to write ROS nodes in Python.

Tag	Explanation
`<depend>geometry_msgs</depend>`	Contains `Twist` message for `/cmd_vel`. Used by teleop and obstacle avoidance.
`<depend>sensor_msgs</depend>`	Contains `LaserScan`, `Imu`, `Image`. Used to read LIDAR, IMU, camera topics.
`<depend>nav_msgs</depend>`	Contains `OccupancyGrid` (map), `Odometry`. Used by gmapping and navigation.
`<depend>actionlib</depend>`	Action client/server library. Used by go_to_point.py and multi_point_nav.py to send goals to move_base.
`<depend>move_base_msgs</depend>`	Contains `MoveBaseAction` and `MoveBaseGoal` message types used in navigation scripts.
`<depend>gmapping</depend>`	SLAM package dependency — needed for slam_mapping.launch.
`<depend>map_server</depend>`	Map loading package dependency — needed for navigation.launch.
`<depend>amcl</depend>`	Localisation package dependency — needed for navigation.launch.
`<depend>move_base</depend>`	Navigation stack core — needed for navigation.launch.

FILE 2 — CMakeLists.txt

Purpose: Build instructions for catkin.

Line	Explanation
`cmake_minimum_required(VERSION 3.0.2)`	Minimum CMake version. ROS Noetic requires 3.0.2 or higher.
`project(arjuna_gazebo)`	Project name — must match `package.xml` exactly.
`find_package(catkin REQUIRED COMPONENTS ...)`	Finds catkin and all listed ROS component packages before building.
`catkin_package()`	Declares this as a catkin package so others can depend on it.

Line	Explanation
`catkin_install_python(PROGRAMS ...)`	Registers all four Python scripts as ROS executables so `rosrun` can find and run them.
`install(DIRECTORY launch urdf worlds config maps ...)`	Copies all resource folders into the install space. `$(find arjuna_gazebo)` in launch files depends on this.

FILE 3 — urdf/arjuna_sim.urdf

Purpose: Complete physical description of the robot — shape, joints, sensor plugins.

A — Base Body

URDF Element	Explanation
`<link name="base_link">`	Root link of the robot. Everything attaches to this. `base_link` is the standard name in ROS.
`<box size="0.45 0.30 0.10"/>`	Robot body dimensions: 45 cm long, 30 cm wide, 10 cm tall.
`<color rgba="0.2 0.4 0.8 1.0"/>`	Blue colour. RGBA values 0.0–1.0. Last value 1.0 = fully opaque.
`<collision><geometry>`	Physics collision shape used by Gazebo. Matches visual shape so hits look correct.
`<mass value="5.0"/>`	Body mass in kg. Affects how quickly the robot accelerates and decelerates.
`<inertia ixx="0.05" iyy="0.10" izz="0.15"/>`	Rotational inertia tensor. Required for stable Gazebo physics. Values calculated from mass × dimensions.

B — Wheels (same structure for all 4)

URDF Element	Explanation
`<cylinder radius="0.05" length="0.04"/>`	Wheel shape: 5 cm radius (10 cm diameter), 4 cm wide.
`<joint type="continuous">`	Joint with no angle limit — wheel can spin indefinitely. Correct type for drive wheels.
`<parent link="base_link"/>`	Wheel is a child of the body — moves with the body.

URDF Element	Explanation
`<origin xyz="0.15 0.17 -0.03" rpy="-1.5708 0 0"/>`	Front-left wheel position: 15 cm forward, 17 cm left, 3 cm below body centre. `rpy="-1.5708 0 0"` rotates the cylinder 90° so it stands upright sideways instead of lying flat.
`<axis xyz="0 0 1"/>`	Wheel spins around its local Z axis (after the rotation this is the correct lateral axis).
Front-right origin	`xyz="0.15 -0.17 -0.03"` — same front, opposite side (negative Y).
Rear-left origin	`xyz="-0.15 0.17 -0.03"` — negative X for rear.
Rear-right origin	`xyz="-0.15 -0.17 -0.03"` — rear and right.

C — LIDAR Link

URDF Element	Explanation
`<cylinder radius="0.04" length="0.05"/>`	Red cylinder representing the RPLIDAR body.
`<joint type="fixed">`	LIDAR does not rotate relative to the body. Fixed joint.
`<origin xyz="0.0 0.0 0.20"/>`	LIDAR is placed 20 cm above the body centre. **Raised from 0.10 to 0.20 to sit above the camera block** — prevents the camera from appearing as an obstacle in the scan.

D — Camera Link

URDF Element	Explanation
`<box size="0.04 0.08 0.04"/>`	Grey box representing the camera — 4 cm deep, 8 cm wide, 4 cm tall.
`<joint type="fixed">`	Camera does not move. Fixed joint.
`<origin xyz="0.20 0.0 0.08"/>`	Placed 20 cm forward and 8 cm above body centre — front face of the robot.

E — Gazebo Plugins

Plugin	Library File	Topics	Explanation
Skid Steer Drive	`libgazebo_ros_skid_steer_drive.so`	Subscribes `/cmd_vel`, Publishes `/odom`	Controls all wheels from velocity commands

Plugin	Library File	Topics	Explanation
			Acts as the motor drive
LIDAR	`libgazebo_ros_laser.so`	Publishes `/scan`	Fires 360 simulated laser pulses per rotation Output is identical to real RPLIDA
Camera	`libgazebo_ros_camera.so`	Publishes `/camera/image_raw`	Renders a virtual cam from the camera_link position.
IMU	`libgazebo_ros_imu_sensor.so`	Publishes `/imu/data`	Simulates accelerome gyroscope, and orientation 50 Hz.

Plugin Parameter	Value	Explanation
`<leftFrontJoint>`	`front_left_joint`	Maps the drive plugin to the correct wheel joint names in URDF.
`<wheelSeparation>`	`0.34`	Distance between left and right wheels in metres. Used for differential kinematics.
`<wheelDiameter>`	`0.10`	Wheel diameter in metres (radius 0.05 × 2). Converts RPM to m/s.
`<commandTopic>`	`cmd_vel`	Topic to receive drive commands from. Teleop and all navigation nodes publish here.
`<broadcastTF>`	`1`	Enables automatic odom → base_link TF publishing. Required for gmapping and navigation.
`<samples>`	`360`	One laser pulse per degree — full 360° scan.
`<min_angle>` / `<max_angle>`	`-π / +π`	Scan covers complete circle.
`<mu1> / <mu2>`	`1.5`	Wheel friction coefficients. 1.5 gives good grip without sliding.

Plugin Parameter	Value	Explanation
`<kp>`	`1e6`	Ground contact stiffness. Prevents wheels sinking into floor.

FILE 4 — worlds/arjuna_world.world

Purpose: Defines the Gazebo simulation environment — the room and all obstacles.

Element	Value / Setting	Explanation
`<include><uri>model://sun</uri>`	Built-in sun model	Adds directional lighting. Without this the world is completely dark.
`<include> <uri>model://ground_plane</uri>`	Built-in ground	Flat grey floor. Without this the robot falls through empty space.
`<static>true</static>`	All walls and obstacles	Makes objects immovable with infinite mass. Robot cannot push them.
Wall north pose	`0 5 0.5 0 0 0`	Centred at x=0, y=5 m (north edge), z=0.5 m (half the 1 m wall height).
Wall dimensions	`10.2 × 0.2 × 1.0 m`	10.2 m long to cover full room width with corner overlap. 0.2 m thick. 1 m tall.
`obstacle_red` pose	`2.5 0 0.3`	Directly in front of the robot at spawn — first thing the robot encounters. Red colour.
`obstacle_green` pose	`-1.5 3.0 0.3`	Left-rear area. Green colour.
`obstacle_blue` pose	`-2.0 -2.5 0.3`	Rear-right area. Blue colour.
`obstacle_yellow` pose	`3.0 -3.0 0.3`	Front-right area. Yellow colour.
Obstacle dimensions	`0.5 × 0.5 × 0.6 m`	50 cm wide. 60 cm tall — taller than the LIDAR height at 20 cm. LIDAR will always detect them.

FILE 5 — launch/arjuna_sim.launch

Purpose: Launches Gazebo with the world, spawns the robot, starts TF publishers.

Line	Explanation
`<param name="robot_description" textfile="...urdf"/>`	Loads the URDF file as a text string into the ROS parameter server. All other nodes read the robot description from here.
`$(find arjuna_gazebo)`	ROS macro that resolves to the full filesystem path of the package. Works from any directory.
`<include file="$(find gazebo_ros)/launch/empty_world.launch">`	Reuses the standard Gazebo launch file and overrides its arguments.
`<arg name="world_name" value="...arjuna_world.world"/>`	Passes our custom world file to Gazebo instead of the default empty world.
`<arg name="use_sim_time" value="true"/>`	All ROS nodes use Gazebo's simulated clock. Required for correct timing in simulation.
`<arg name="gui" value="true"/>`	Opens the Gazebo graphical window. Set to `false` for headless/server use.
`spawn_model args="-urdf -model arjuna_sim -param robot_description -x 0 -y 0 -z 0.1"`	Reads URDF from parameter server and places the robot at coordinates (0, 0, 0.1). The 0.1 m Z offset prevents clipping into the ground.
`-R 0.0 -P 0.0 -Y 0.0`	Initial orientation: roll, pitch, yaw all zero. Robot faces the positive X direction (toward the red obstacle).
`joint_state_publisher`	Reads all joint angles and publishes on `/joint_states`. robot_state_publisher reads this to compute transforms.
`robot_state_publisher`	Reads URDF + joint_states and publishes TF for every link. Required by gmapping, amcl, move_base, and rviz.

FILE 6 — launch/slam_mapping.launch

Purpose: Starts the gmapping SLAM node to build a map while driving.

Parameter	Value	Explanation
`base_frame`	`base_link`	TF frame of the robot centre. gmapping uses this as the robot pose reference.
`odom_frame`	`odom`	TF frame of odometry. gmapping uses odom → base_link to track relative movement between scans.
`map_frame`	`map`	Output frame. gmapping publishes the map → odom transform as it builds the map.
`scan_topic`	`/scan`	LIDAR input topic. gmapping reads this to update the map.
`maxUrange`	`10.0`	Maximum range used for map building in metres. Readings beyond this are ignored to reduce noise.
`maxRange`	`12.0`	Maximum physical range of the sensor.
`minimumScore`	`50`	Quality threshold for accepting a scan match. Low = accept more scans (faster but noisier). High = stricter quality.
`map_update_interval`	`1.0`	Seconds between map grid updates. Lower = smoother map updates but more CPU.
`linearUpdate`	`0.2`	Robot must move 20 cm before a new scan is processed. Prevents over-processing when stationary.
`angularUpdate`	`0.3`	Robot must rotate ~17° before a new scan is processed.
`delta`	`0.05`	Map resolution: 5 cm per cell. Each cell is 5×5 cm in the real world.
`xmin/ymin/xmax/ymax`	`-10 to +10`	Map boundaries in metres. Set to match the 10×10 m world size.
`particles`	`50`	Number of particle filter particles. More particles = more accurate but slower. 50 is good for simulation.

FILE 7 — launch/navigation.launch

Purpose: Loads saved map, starts AMCL localisation, and starts move_base for autonomous navigation.

Node	Package	Explanation
`map_server`	`map_server`	Reads `maps/my_map.yaml`, loads the `.pgm` image file, and publishes the map on `/map` as a `nav_msgs/OccupancyGrid`.
`amcl`	`amcl`	Adaptive Monte Carlo Localisation. Reads `/scan` and `/odom`, compares real sensor readings to the saved map, and estimates where the robot actually is on the map. Publishes `map→odom` TF.
`move_base`	`move_base`	The navigation brain. Receives a goal pose, runs the global planner to find a route, runs the local planner (DWA) to follow it in real time, and publishes velocity commands to `/cmd_vel`.

`<rosparam file=...` line	What it loads
`move_base_params.yaml`	Planner selection (navfn global, DWA local), update frequencies, recovery behaviour.
`costmap_common_params.yaml` into `global_costmap` namespace	Applies robot_radius, obstacle layer, inflation layer settings to the global costmap.
`costmap_common_params.yaml` into `local_costmap` namespace	Same settings applied to the local costmap.
`global_costmap_params.yaml`	Sets global costmap reference frame (`map`), update rate, static map usage.
`local_costmap_params.yaml`	Sets local costmap reference frame (`odom`), rolling window size (4×4 m), update rate.
`base_local_planner_params.yaml`	DWA planner velocity limits, acceleration limits, goal tolerance, scoring weights.

FILE 8 — config/costmap_common_params.yaml

Purpose: Shared settings for both global and local costmaps — robot size and obstacle detection.

Parameter	Value	Explanation
`robot_radius`	`0.25`	25 cm circular footprint. The planner will not route the robot through gaps smaller than 2 × 0.25 = 50 cm.
`observation_sources`	`laser_scan_sensor`	Name of the sensor configuration used for obstacle detection.
`sensor_frame`	`lidar_link`	TF frame the LIDAR data comes from. Must match `frameName` in URDF LIDAR plugin.
`data_type`	`LaserScan`	Message type on the LIDAR topic.
`topic`	`/scan`	Subscribes to the LIDAR topic.
`marking: true`	—	When a reading detects an obstacle — mark that cell as occupied in the costmap.
`clearing: true`	—	When a reading shows free space — clear that cell in the costmap.
`inflation_radius`	`0.35`	Adds a 35 cm safety buffer around every obstacle. Planner routes around this buffer. Adjust to make the robot more or less cautious.
`cost_scaling_factor`	`5.0`	How quickly the inflation cost drops off from the obstacle edge. Higher = steeper drop = narrower buffer effect.
`resolution`	`0.05`	5 cm per costmap cell. Must match gmapping `delta` setting.

FILE 9 — config/global_costmap_params.yaml

Purpose: Settings for the costmap used by the global path planner.

Parameter	Value	Explanation
`global_frame`	`map`	Global costmap is in map coordinates — uses the saved/built map as reference.
`robot_base_frame`	`base_link`	Robot position is tracked relative to this frame.
`update_frequency`	`2.0 Hz`	Updates costmap 2 times per second. Slower than local costmap — global plan does not need to react in real time.

Parameter	Value	Explanation
`static_map`	`true`	Uses the map loaded by map_server as its base layer. Static obstacles from the map are included.
`plugins`	static + obstacle + inflation	Three layers: the saved map, live sensor obstacles, and inflation buffer.

FILE 10 — config/local_costmap_params.yaml

Purpose: Settings for the costmap used by the local planner (real-time obstacle avoidance).

Parameter	Value	Explanation
`global_frame`	`odom`	Local costmap is in odometry coordinates — follows the robot as it moves.
`update_frequency`	`5.0 Hz`	Updates 5 times per second — fast enough for real-time avoidance.
`static_map`	`false`	Does NOT use the saved map. Only uses live sensor data.
`rolling_window`	`true`	The 4×4 m window moves with the robot — always centred on the robot.
`width / height`	`4.0 m`	Size of the local costmap window.
`resolution`	`0.05`	5 cm per cell — same as global costmap.
`plugins`	obstacle + inflation	Only two layers: live sensor obstacles and inflation. No static map layer needed.

FILE 11 — config/base_local_planner_params.yaml

Purpose: DWA (Dynamic Window Approach) local planner tuning — velocity limits and goal tolerance.

Parameter	Value	Explanation
`max_vel_x`	`0.4 m/s`	Maximum forward speed during navigation. Lower than obstacle avoidance to be safer.
`min_vel_x`	`0.0 m/s`	Allows the robot to stop.
`max_vel_y / min_vel_y`	`0.0`	No sideways movement. Arjuna is a differential drive robot — it cannot slide sideways.

Parameter	Value	Explanation
max_vel_trans	0.4 m/s	Maximum combined translational speed.
min_vel_trans	0.1 m/s	Minimum speed while moving. Prevents very slow creeping.
max_vel_theta	0.8 rad/s	Maximum rotation speed (~46°/s).
min_vel_theta	0.3 rad/s	Minimum rotation speed to ensure turns complete.
acc_lim_x	1.0 m/s²	Forward acceleration limit — how quickly the robot can speed up or slow down.
acc_lim_theta	1.5 rad/s²	Rotational acceleration limit.
xy_goal_tolerance	0.20 m	Robot considers itself at the goal when within 20 cm.
yaw_goal_tolerance	0.15 rad	Acceptable heading error at goal (~8.6°).
sim_time	1.5 s	DWA simulates 1.5 seconds ahead when sampling velocity commands.
vx_samples	10	Number of different forward speeds to try when looking for the best velocity.
vth_samples	20	Number of different angular speeds to try. More samples = better path but more CPU.
path_distance_bias	32.0	How strongly the robot follows the planned global path. Higher = follows path more strictly.
goal_distance_bias	20.0	How strongly the robot heads toward the goal.
occdist_scale	0.02	How strongly the robot avoids obstacles in the costmap.

FILE 12 — config/move_base_params.yaml

Purpose: General move_base settings — planner selection, timing, recovery.

Parameter	Value	Explanation
base_global_planner	navfn/NavfnROS	Dijkstra-based global planner. Finds shortest path on the costmap

Parameter	Value	Explanation
		from current position to goal.
`base_local_planner`	`dwa_local_planner/DWAPlannerROS`	DWA local planner. Converts global path into velocity commands for real-time driving.
`planner_frequency`	`1.0 Hz`	Global plan is recalculated once per second.
`controller_frequency`	`5.0 Hz`	Local planner runs 5 times per second — fast enough for smooth motion.
`planner_patience`	`5.0 s`	If no global plan found in 5 seconds — attempt recovery behaviour.
`controller_patience`	`15.0 s`	If robot makes no progress in 15 seconds — attempt recovery.
`recovery_behavior_enabled`	`true`	Enables automatic recovery when stuck.
`clearing_rotation_allowed`	`true`	Recovery behaviour: spin in place to clear the costmap and detect new obstacles.

Parameter	Value	Explanation
`oscillation_timeout`	`10.0 s`	If robot oscillates (goes back and forth) for 10 seconds — trigger recovery.
`oscillation_distance`	`0.2 m`	Robot must move 20 cm between checks to not be considered oscillating.

FILE 13 — config/amcl_params.yaml

Purpose: AMCL particle filter localisation settings.

Parameter	Value	Explanation
`min_particles`	`500`	Minimum number of particles. More particles = more accurate but slower.
`max_particles`	`2000`	Maximum particles. AMCL adapts between min and max based on uncertainty.
`alpha1-alpha4`	`0.2, 0.2, 0.8, 0.2`	Odometry noise model. How much the real wheel motion differs from what odometry reports. Higher values = trust odometry less.
`laser_model_type`	`likelihood_field`	Likelihood field model — faster and works better in simulation than beam model.
`laser_max_beams`	`60`	Number of LIDAR beams used per update (subset of 360 for speed).
`update_min_d`	`0.1 m`	Resample particles after robot moves 10 cm.
`update_min_a`	`0.2 rad`	Resample after rotating ~11°.
`odom_frame_id`	`odom`	Must match the odometry frame from the drive plugin.

Parameter	Value	Explanation
`base_frame_id`	`base_link`	Must match the robot base frame in URDF.
`global_frame_id`	`map`	The map coordinate frame.
`initial_pose_x/y/a`	`0.0, 0.0, 0.0`	Robot assumed to start at map origin facing east. Change if robot spawns elsewhere.

FILE 14 — scripts/teleop_key.py

Purpose: Custom keyboard teleop using w/a/s/d keys. No extra packages needed.

Code Line	Explanation
`#!/usr/bin/env python3`	Shebang line — tells the OS to use Python 3. Required for `rosrun` to execute the file directly.
`from geometry_msgs.msg import Twist`	Imports the `Twist` message type containing `linear.x` and `angular.z` fields for velocity control.
`import tty, termios`	Terminal control libraries built into Python. Used to capture single keypresses without pressing Enter.
`FORWARD_SPEED = 0.3`	Forward speed in m/s. Edit this to make the robot faster or slower.
`TURN_SPEED = 0.5`	Turning speed in rad/s. Edit this for sharper or gentler turns.
`def get_key():`	Reads exactly one character from keyboard immediately on press — no Enter needed.
`fd = sys.stdin.fileno()`	Gets the file descriptor number for standard input (the keyboard).
`old = termios.tcgetattr(fd)`	Saves the current terminal settings so they can be restored after reading.
`tty.setraw(fd)`	Puts the terminal in raw mode — characters sent immediately on press, no line buffering.
`ch = sys.stdin.read(1)`	Reads exactly one character.

Code Line	Explanation
`termios.tcsetattr(fd, termios.TCSADRAIN, old)`	Restores the original terminal settings. In `finally` block so it always runs even if an error occurs.
`rospy.init_node('teleop_key')`	Registers this script as a ROS node. Must be called before any ROS communication.
`pub = rospy.Publisher('/cmd_vel', Twist, queue_size=10)`	Creates a publisher on `/cmd_vel`. The Gazebo drive plugin subscribes here and moves the wheels.
`twist = Twist()`	Creates a Twist message. All fields start at 0.0 by default.
`while not rospy.is_shutdown():`	Keeps running until Ctrl+C or roscore stops. Cleaner than `while True`.
`key == 'w': twist.linear.x = FORWARD_SPEED`	W key: set forward speed, no rotation.
`key == 's': twist.linear.x = -FORWARD_SPEED`	S key: negative linear.x = move backward.
`key == 'a': twist.angular.z = TURN_SPEED`	A key: positive angular.z = turn left (counter-clockwise). linear.x = 0 = turn in place.
`key == 'd': twist.angular.z = -TURN_SPEED`	D key: negative angular.z = turn right (clockwise).
`key == 'x': twist = zero`	X key: zero all fields = full stop.
`key == 'q': pub.publish(twist); break`	Q key: publish a final stop command then exit cleanly.
`else: twist = zero`	Any unknown key: stop. Prevents robot from continuing a command accidentally.
`pub.publish(twist)`	Sends the Twist message to `/cmd_vel`. Drive plugin receives it and spins wheels.

FILE 15 — scripts/obstacle_avoidance.py

Purpose: Reads LIDAR, decides direction, publishes velocity commands automatically.

Code Line	Explanation
`OBSTACLE_DIST = 0.7`	React when an obstacle is within 0.7 m in front. Increase to react earlier.

Code Line	Explanation
`FORWARD_SPEED = 0.3`	Default forward speed when path is clear.
`TURN_SPEED = 0.5`	Speed when turning to avoid an obstacle.
`cmd_pub = None`	Global publisher variable — declared here so both `main()` and `scan_callback()` can use it.
`def get_min_dist(ranges, angle_min, angle_inc, from_deg, to_deg):`	Helper function. Finds the closest valid distance in a defined angular region of the LIDAR scan. Angle-based so it works with any number of scan samples (360 in Gazebo, 850 on real RPLIDAR).
`from_rad = math.radians(from_deg)`	Converts the region boundary angles to radians (LaserScan stores angles in radians).
`for i, dist in enumerate(ranges):`	Loops through every LIDAR pulse. `i` = pulse index, `dist` = measured distance.
`if math.isinf(dist) or math.isnan(dist): continue`	Skips invalid readings. `inf` means nothing detected in that direction. `nan` means sensor error. Both break `min()`.
`if dist <= 0.01: continue`	Skips readings under 1 cm — sensor self-noise or the robot detecting its own body.
`angle = angle_min + i * angle_inc`	Calculates the actual angle this pulse was fired at in radians.
`if from_rad <= angle <= to_rad: distances.append(dist)`	Collects only the readings inside the region we are checking.
`return min(distances) if distances else 10.0`	Returns the closest obstacle in the region. Returns 10.0 (effectively clear) if no valid readings found.
`front = get_min_dist(..., -15, 15)`	Closest obstacle in the 30° cone ahead (±15° around 0°).
`left = get_min_dist(..., 15, 90)`	Closest obstacle in the 75° arc to the left.
`right = get_min_dist(..., -90, -15)`	Closest obstacle in the 75° arc to the right.
`if front > OBSTACLE_DIST:`	Front clear — move forward.
`elif left >= right:`	Front blocked. Left has more space — turn left toward the opening.
`twist.angular.z = TURN_SPEED`	Positive angular.z = turn left. linear.x = 0 = turn in place.

Code Line	Explanation
`else:`	Front blocked, right side has more space — turn right.
`twist.angular.z = -TURN_SPEED`	Negative angular.z = turn right.
`cmd_pub.publish(twist)`	Sends the decision to `/cmd_vel` . Robot moves accordingly.
`rospy.loginfo(...)`	Prints state and distances every scan — students can watch the decisions live.
`rospy.Subscriber('/scan', LaserScan, scan_callback)`	Subscribes to LIDAR. `scan_callback` fires automatically at 10 Hz.
`rospy.spin()`	Keeps the node alive waiting for LIDAR messages. Without this the program exits immediately.

FILE 16 — scripts/go_to_point.py

Purpose: Sends one navigation goal to move_base. Robot plans a path and drives there autonomously.

Code Line	Explanation
`GOAL_X = 2.0`	Target X coordinate in the map frame (metres). Change to any valid map position.
`GOAL_Y = 2.0`	Target Y coordinate in the map frame (metres).
`GOAL_YAW = 0.0`	Final heading at the goal in radians. 0 = face east. 1.57 = face north. 3.14 = face west.
`def yaw_to_quaternion(yaw):`	Converts a yaw angle in radians to a quaternion (x, y, z, w). move_base requires orientation as a quaternion not degrees.
`q.z = math.sin(yaw / 2.0)`	Z component of quaternion from yaw angle.
`q.w = math.cos(yaw / 2.0)`	W component of quaternion from yaw angle. X and Y are 0 for pure yaw rotation.
`client = actionlib.SimpleActionClient('move_base',`	Creates an action client that connects to the `move_base`

Code Line	Explanation
`MoveBaseAction)`	action server. Actions are like service calls but long-running with feedback.
`client.wait_for_server()`	Blocks until move_base is fully started. Must be called before sending any goal.
`goal = MoveBaseGoal()`	Creates a goal message containing the target position and orientation.
`goal.target_pose.header.frame_id = "map"`	Goal coordinates are in the map frame. AMCL and move_base both work in this frame.
`goal.target_pose.header.stamp = rospy.Time.now()`	Timestamp for the goal. Required for TF to work correctly.
`goal.target_pose.pose.position.x = x`	Sets the target X position.
`goal.target_pose.pose.orientation = yaw_to_quaternion(yaw)`	Sets the target heading as a quaternion.
`client.send_goal(goal)`	Sends the goal to move_base. move_base immediately starts planning and driving.
`client.wait_for_result()`	Blocks until the robot reaches the goal or gives up.
`result = client.get_state()`	Gets the final status code: 3 = SUCCEEDED, 4 = ABORTED, 5 = REJECTED.
`if result == actionlib.GoalStatus.SUCCEEDED:`	Checks if the robot actually reached the goal.

FILE 17 — scripts/multi_point_nav.py

Purpose: Sends multiple goals to move_base in sequence. Robot visits each waypoint in order.

Code Line	Explanation
`WAYPOINTS = [(x, y, yaw), ...]`	List of waypoints to visit. Each is a tuple of (X metres, Y metres, yaw radians). Robot visits them in list order.

Code Line	Explanation
`( 0.0, 0.0, 0.0)`	Last waypoint returns robot to map origin (home position).
`def send_goal(client, x, y, yaw, waypoint_num):`	Sends one goal and waits for result. Returns `True` if reached, `False` if failed. Same logic as go_to_point.py.
`for i, (x, y, yaw) in enumerate(WAYPOINTS):`	Loops through every waypoint. `enumerate` gives both the index `i` and the values.
`if rospy.is_shutdown(): break`	Stops the loop cleanly if Ctrl+C is pressed mid-navigation.
`success = send_goal(client, x, y, yaw, i + 1)`	Send this waypoint as a goal. Waits for completion before continuing.
`rospy.logwarn("Skipping to next waypoint...")`	If a waypoint fails, the script continues to the next one rather than stopping entirely.
`rospy.sleep(1.0)`	1 second pause between waypoints. Gives the robot time to stabilise before the next goal.
`rospy.loginfo("NAVIGATION COMPLETE")`	Summary printed when all waypoints are processed. Shows how many were reached and how many failed.

PART 3 — TOPICS REFERENCE

Topic	Message Type	Direction	Pub
`/cmd_vel`	`geometry_msgs/Twist`	← input	tele obs mo
`/scan`	`sensor_msgs/LaserScan`	→ output	Gaz
`/camera/image_raw`	`sensor_msgs/Image`	→ output	Gaz plu
`/imu/data`	`sensor_msgs/Imu`	→ output	Gaz
`/odom`	`nav_msgs/Odometry`	→ output	Gaz

Topic	Message Type	Direction	Pul
`/map`	`nav_msgs/OccupancyGrid`	→ output	gm SL (na
`/amcl_pose`	`geometry_msgs/PoseWithCovarianceStamped`	→ output	am
`/move_base/goal`	`move_base_msgs/MoveBaseActionGoal`	← input	go_ mu rviz
`/tf`	TF transforms	↔ both	rob / d

PART 4 — HOW SLAM AND NAVIGATION WORK

SLAM (Simultaneous Localisation and Mapping)

```
Robot moves around the room
      ↓
LIDAR publishes /scan (360 distance readings every 0.1 sec)
      ↓
Drive plugin publishes /odom (estimated position from wheel rotations)
      ↓
gmapping reads both — matches each new scan to previously seen features
      ↓
Builds an occupancy grid map:
  White cells  = free space (robot can drive here)
  Black cells  = wall or obstacle detected
  Grey cells   = unknown (not yet scanned)
      ↓
Publishes /map — visible in rviz as the map grows
      ↓
When room is fully mapped → save with map_saver
```

Navigation (After Map is Saved)

```
map_server loads my_map.yaml → publishes /map
      ↓
amcl compares live /scan to the saved map
  → estimates robot's current position → publishes map→odom TF
```

```
      ↓
move_base receives a goal (x, y) in map coordinates
      ↓
Global planner (navfn) — finds shortest path on the global costmap
      ↓
Local planner (DWA) — follows path while avoiding live obstacles
→ publishes /cmd_vel with safe velocity commands
      ↓
Gazebo drive plugin receives /cmd_vel → wheels turn → robot moves
      ↓
Process repeats at 5 Hz until robot reaches the goal
```

PART 5 — EXPERIMENTS WITH COMMANDS

> Source the workspace in every new terminal before running anything:
>
> ```
> source ~/ros_sim/devel/setup.bash
> ```

EXPERIMENT 1 — Load Robot Only (Empty World)

What you see: Gazebo opens. Blue robot with red LIDAR cylinder and grey camera box appears in an empty world. No walls. No obstacles.

Purpose: Verify the robot URDF loads correctly and all sensor topics appear.

```
# Terminal 1 — launch simulation
roslaunch arjuna_gazebo arjuna_sim.launch

# Terminal 2 — verify all topics appeared
rostopic list
```

Expected topics:

```
/scan
/camera/image_raw
/imu/data
/odom
```

```
/cmd_vel
/tf
```

Verify sensors are publishing:

```
rostopic hz /scan             # → ~10 Hz
rostopic hz /imu/data         # → ~50 Hz
rostopic hz /camera/image_raw # → ~30 Hz
```

EXPERIMENT 2 — Robot with Environment

What you see: Full 10×10 m room with 4 grey walls. Four coloured obstacle boxes inside — red (front), green (left), blue (rear), yellow (right). Robot spawns at centre facing the red box.

Purpose: Verify the world loads and LIDAR detects the walls and obstacles.

```
# Terminal 1
roslaunch arjuna_gazebo arjuna_sim.launch

# Terminal 2 — view LIDAR readings
rostopic echo /scan | grep -A5 "ranges"

# Terminal 3 — visualise in rviz
rviz
```

rviz setup:

1. Fixed Frame → `odom`
2. Add → **RobotModel** (see 3D robot)
3. Add → **LaserScan** → Topic: `/scan` (see laser dots on walls)

EXPERIMENT 3 — Robot with Environment and Teleop

What you see: Robot in the room. You drive it manually using keyboard.

Option A — Custom teleop (w/a/s/d, no extra window):

```
# Terminal 1
roslaunch arjuna_gazebo arjuna_sim.launch
```

```
# Terminal 2 — drive with keyboard
rosrun arjuna_gazebo teleop_key.py
```

Key	Action
w	Forward
s	Backward
a	Turn Left
d	Turn Right
x	Stop
q	Quit

Option B — Standard teleop (opens in separate xterm window):

```
# Terminal 1
roslaunch arjuna_gazebo arjuna_sim.launch

# Terminal 2
roslaunch arjuna_gazebo teleop.launch
```

Key	Action
i	Forward
,	Backward
j	Turn Left
l	Turn Right
k	Stop
q / z	Speed up / slow down

EXPERIMENT 4 — Robot with Obstacle Avoidance

What you see: Robot starts moving forward automatically. Detects the red box → turns toward more open space → continues forward → repeats for every obstacle it encounters.

Purpose: Demonstrate reactive navigation using LIDAR.

```
# Terminal 1
roslaunch arjuna_gazebo arjuna_sim.launch
```

```
# Terminal 2
roslaunch arjuna_gazebo obstacle_avoidance.launch
```

Expected Terminal 2 output:

```
[FORWARD  ]  Front: 3.45m  Left: 4.20m  Right: 3.80m
[FORWARD  ]  Front: 1.23m  Left: 3.90m  Right: 4.10m
[TURN RIGHT]  Front: 0.55m  Left: 1.20m  Right: 3.40m
[TURN RIGHT]  Front: 0.71m  Left: 1.50m  Right: 3.60m
[FORWARD  ]  Front: 2.10m  Left: 3.20m  Right: 3.80m
```

Tune the behaviour by editing `scripts/obstacle_avoidance.py` :

```
OBSTACLE_DIST = 0.7   # increase to react earlier
FORWARD_SPEED = 0.3   # increase for faster movement
TURN_SPEED    = 0.5   # increase for sharper turns
```

After editing: `cd ~/ros_sim && catkin_make && source devel/setup.bash`

EXPERIMENT 5 — SLAM Mapping

What you see: Robot in the room. A 2D map grows in rviz in real time as you drive. White = explored free space. Black = walls and obstacles. Grey = unexplored.

Purpose: Build a map of the room that will be used for autonomous navigation.

Step A — Start simulation and mapper:

```
# Terminal 1 — simulation
roslaunch arjuna_gazebo arjuna_sim.launch

# Terminal 2 — start SLAM mapper
roslaunch arjuna_gazebo slam_mapping.launch

# Terminal 3 — drive around to build the map
rosrun arjuna_gazebo teleop_key.py
```

Step B — Watch the map build in rviz:

```
# Terminal 4
rviz
```

rviz setup:

1. Fixed Frame → `map`
2. Add → **RobotModel**
3. Add → **Map** → Topic: `/map` (watch the map grow as you drive)
4. Add → **LaserScan** → Topic: `/scan`

Step C — Drive the robot around the entire room:

- Drive along all walls
- Drive past all 4 obstacles
- Return to starting position
- The map should show all walls and obstacles as black cells

Step D — Save the map when complete:

```
# Terminal 5 — save the map
rosrun map_server map_saver -f ~/ros_sim/src/arjuna_gazebo/maps/my_map
```

This creates two files:

```
maps/my_map.pgm     ← map image (open in any image viewer)
maps/my_map.yaml    ← metadata (resolution, origin)
```

Do not close Gazebo after saving — use the same session for Experiment 6.

EXPERIMENT 6 — Single Point Navigation (go_to_point)

What you see: Robot automatically plans a path to a single target coordinate and drives there. move_base handles all path planning and obstacle avoidance.

Prerequisite: Experiment 5 completed — `maps/my_map.yaml` must exist.

Step A — If starting fresh (Gazebo not running):

```
# Terminal 1
roslaunch arjuna_gazebo arjuna_sim.launch

# Terminal 2
roslaunch arjuna_gazebo navigation.launch
```

If continuing from Experiment 5 (Gazebo still running):

```
# Stop slam_mapping (Ctrl+C in Terminal 2)
# Then start navigation instead:
roslaunch arjuna_gazebo navigation.launch
```

Step B — Set your target in go_to_point.py:

```
gedit ~/ros_sim/src/arjuna_gazebo/scripts/go_to_point.py
```

Edit these lines:

```
GOAL_X   =  2.0    # target X in metres
GOAL_Y   =  2.0    # target Y in metres
GOAL_YAW =  0.0    # final heading (0 = face east)
```

Save the file. Rebuild:

```
cd ~/ros_sim && catkin_make && source devel/setup.bash
```

Step C — Run navigation:

```
# Terminal 3
rosrun arjuna_gazebo go_to_point.py
```

Step D — Watch in rviz:

```
# Terminal 4
rviz
```

rviz setup:

1. Fixed Frame → `map`

2. Add → **RobotModel**
3. Add → **Map** → Topic: `/map`
4. Add → **Path** → Topic: `/move_base/NavfnROS/plan` (shows planned path)
5. Add → **LaserScan** → Topic: `/scan`

You can also send goals from rviz directly:

- Click **2D Nav Goal** button in the toolbar
- Click a location on the map
- Robot drives there automatically

Expected terminal output:

```
[INFO] go_to_point started
[INFO] Target: X=2.00  Y=2.00  Yaw=0.00
[INFO] Sending goal → X: 2.00  Y: 2.00
[INFO] GOAL REACHED — arrived at (2.00, 2.00)
```

EXPERIMENT 7 — Multi-Point Navigation (Waypoints)

What you see: Robot visits multiple points in sequence automatically. Each waypoint is reached before the robot moves to the next one. Final waypoint returns to home.

Prerequisite: Experiment 5 completed. `navigation.launch` running.

Step A — Edit the waypoints in multi_point_nav.py:

```
gedit ~/ros_sim/src/arjuna_gazebo/scripts/multi_point_nav.py
```

Find and edit the WAYPOINTS list:

```
WAYPOINTS = [
    ( 2.0,  2.0,  0.0),    # Waypoint 1
    ( 2.0, -2.0,  0.0),    # Waypoint 2
    (-2.0, -2.0,  3.14),   # Waypoint 3
    (-2.0,  2.0,  3.14),   # Waypoint 4
    ( 0.0,  0.0,  0.0),    # Waypoint 5 — home
]
```

Waypoint coordinates must be inside the room (within ±4.5 m) and not inside an obstacle.

Save and rebuild:

```
cd ~/ros_sim && catkin_make && source devel/setup.bash
```

Step B — Make sure navigation is running:

```
# Terminal 1 — if not already running
roslaunch arjuna_gazebo arjuna_sim.launch

# Terminal 2 — if not already running
roslaunch arjuna_gazebo navigation.launch
```

Step C — Run the waypoint navigator:

```
# Terminal 3
rosrun arjuna_gazebo multi_point_nav.py
```

Expected terminal output:

```
[INFO] Waiting for move_base action server...
[INFO] move_base is ready.
[INFO] Starting waypoint navigation — 5 waypoints
[INFO] ──────────────────────────────────────────────────
[INFO] Waypoint 1 / 5  →  X: 2.00  Y: 2.00  Yaw: 0.00
[INFO] ✓ Waypoint 1 REACHED
[INFO] ──────────────────────────────────────────────────
[INFO] Waypoint 2 / 5  →  X: 2.00  Y: -2.00  Yaw: 0.00
[INFO] ✓ Waypoint 2 REACHED
...
[INFO] ══════════════════════════════════════════════════
[INFO] NAVIGATION COMPLETE
[INFO] Waypoints reached : 5
[INFO] Waypoints failed  : 0
[INFO] Total waypoints   : 5
```

PART 6 — QUICK COMMAND REFERENCE

Action	Command
Build workspace	`cd ~/ros_sim && catkin_make`
Source workspace	`source ~/ros_sim/devel/setup.bash`
Simulation	
Launch Gazebo + robot	`roslaunch arjuna_gazebo arjuna_sim.launch`
Custom teleop (w/a/s/d)	`rosrun arjuna_gazebo teleop_key.py`
Standard teleop (i/j/k/l)	`roslaunch arjuna_gazebo teleop.launch`
Obstacle avoidance	`roslaunch arjuna_gazebo obstacle_avoidance.launch`
SLAM	
Start SLAM mapping	`roslaunch arjuna_gazebo slam_mapping.launch`
Save completed map	`rosrun map_server map_saver -f ~/ros_sim/src/arjuna_gazebo/maps/my_map`
Navigation	
Start navigation stack	`roslaunch arjuna_gazebo navigation.launch`
Navigate to one point	`rosrun arjuna_gazebo go_to_point.py`
Navigate multiple points	`rosrun arjuna_gazebo multi_point_nav.py`
Diagnostics	
List all topics	`rostopic list`
Check LIDAR rate	`rostopic hz /scan`
Check IMU rate	`rostopic hz /imu/data`
Watch cmd_vel live	`rostopic echo /cmd_vel`
Watch robot pose	`rostopic echo /amcl_pose`
List running nodes	`rosnode list`
Kill Gazebo cleanly	`killall gzserver gzclient`
Kill all ROS nodes	`rosnode kill -a`

PART 7 — COMMON ERRORS AND FIXES

Error	Cause	Fix
`Couldn't find library libgazebo_ros_skid_steer_drive.so`	gazebo-ros-pkgs not installed	`sudo apt install ros-noetic-gazebo-ros-pkgs`
`/scan not in rostopic list`	LIDAR plugin failed	Check Terminal 1 for `[Err]`. Verify `libgazebo_ros_laser.so` in URDF.
`robot visible but does not move`	Teleop terminal not focused	Click inside the teleop window before pressing keys.
`map not building in rviz`	Fixed Frame wrong	Set Fixed Frame to `map` (not `odom`) in rviz during SLAM.
`navigation.launch fails: map file not found`	Map not saved yet	Run Experiment 5 first and save the map.
`GOAL FAILED` during navigation	Goal inside obstacle or wall	Choose coordinates not blocked. Check costmap in rviz.
`AMCL: no laser scan received`	`/scan` not publishing	Verify LIDAR plugin loaded: `rostopic hz /scan`.
`Robot spins but does not localise`	AMCL initial pose wrong	In rviz use **2D Pose Estimate** button to set the robot's starting position on the map.
`catkin_make fails`	Missing dependency	`rosdep install --from-paths src --ignore-src -r -y` then retry.
`Gazebo very slow`	Not enough GPU/RAM	Close other apps. Lower Gazebo rendering in View menu.
`Robot sinks into ground`	Wheel friction/contact wrong	Ensure `<kp>1e6</kp>` in URDF wheel friction settings.
`Obstacle avoidance: robot spins in circles`	LIDAR angle convention mismatch	Swap left and right angle values in `obstacle_avoidance.py`.