Contents

1 SIMLAN, Simulation for Multi-Camera Robotics (4.0.1)

This simulation environment, based on the Ignition Gazebo simulator and ROS 2, resembles a Volvo Trucks warehouse and serves as a playground for rapid prototyping and testing of systems that rely on a multi-camera setup for perception, monitoring, localization or even navigation. This project is inspired by GPSS (Generic photo-based sensor system) that utilizes ceiling-mounted cameras, deep learning and computer vision algorithms, and very simple transport robots. [GPSS demo]

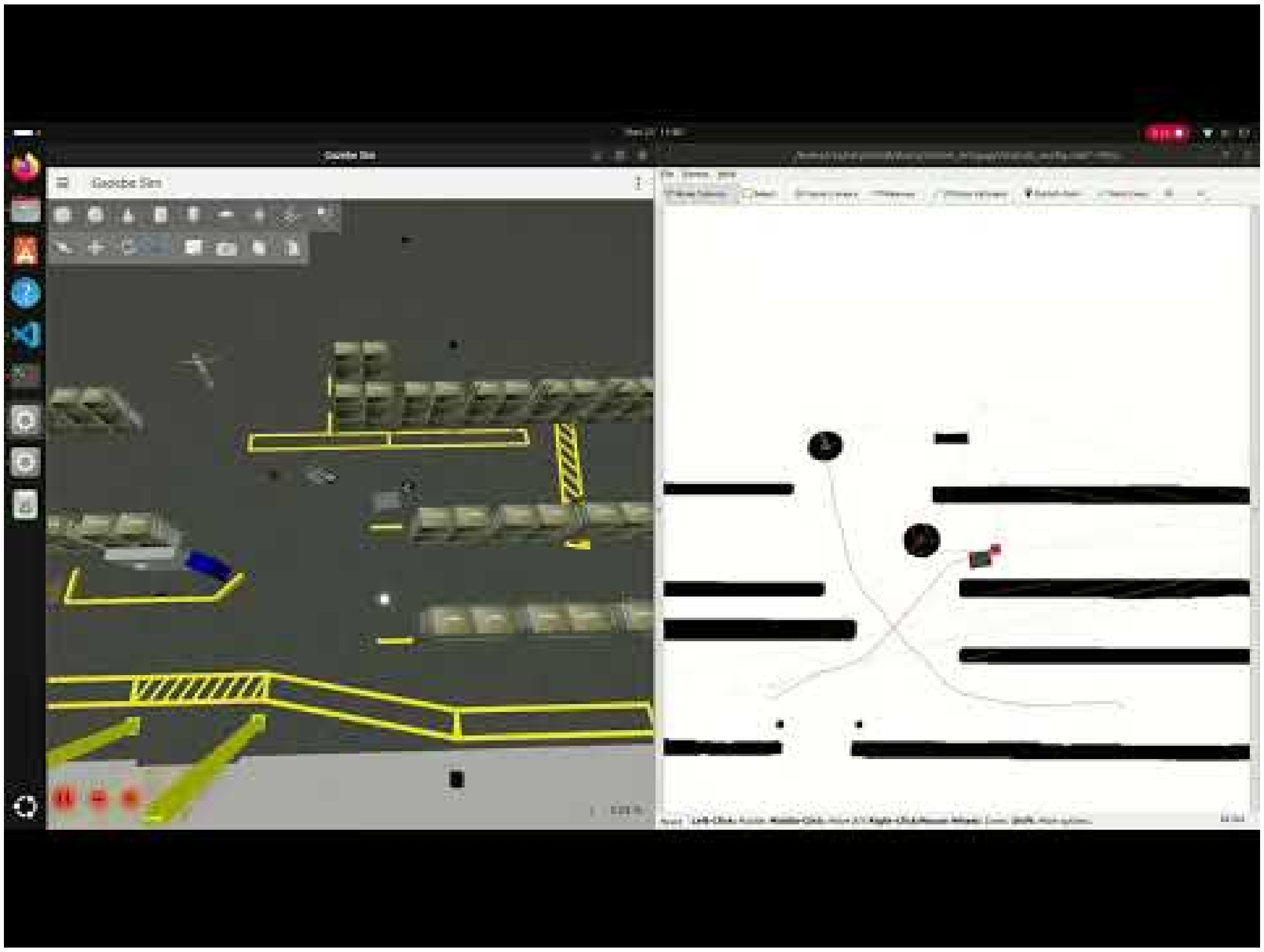
- Online Documentation
- PDF Documentation

1.1 SIMLAN Features

- Ignition Gazebo
- Library of assets
- Real-World environment inspired design (camera position and warehouse layout)
- ROS 2 interfaces (Humble and Jazzy)
- ArUco marker localization

- Simple GPSS (Generic Photo-based Sensor System) navigation
- Multi-Robot localization and navigation using Nav2
- Bird's-Eye view projection
- Multi-Sensor Support (LiDAR, RGB camera, semantic segmentation, depth, etc.)
- Geofencing for safe zones and safe stop on collision
- Motion capture for Human-Robot Collaboration/Interaction (HRC/HRI)

Click the YouTube link below to view the SIMLAN demo video:



1.2 Installation [Demo]

1.2.1 Dependencies

Ubuntu 24.04: use the instructions in dependencies.md#linux-dependencies to install Docker and ensure that your Linux user account has `docker` access. *Attention*: Make sure to restart the computer (for the changes in group membership to take effect) before proceeding to the next step.

Windows 11: use the instructions in dependencies.md#windows-dependencies to install dependencies.

Production environment: follow installation procedure used in .devcontainer/Dockerfile to install dependencies.

Development environment: To improve collaboration we use VS Code and Docker as explained in this instruction and docker files. Install Visual Studio Code (VS Code) and open the project folder. VS Code will prompt you to install the required extension dependencies. Make sure the `Dev containers` extension is installed. Reopen the project in VS Code, and you will be prompted to rebuild the container. Accept the prompt; this process may take a few minutes. Once VS Code is connected to Docker (as shown in the image below), open the terminal and run the following commands:

dev container in vscode

(if you don't see this try to build manually in VS Code by pressing `Ctrl + Shift + P` and select `Dev Containers: Rebuild and Reopen in Container`.)

1.2.2 Quick Start

The best place to learn about the various features, start different components, and understand the project structure is `./control.sh`.

Attention: The following commands (using `./control.sh`) are executed in a separate terminal tab inside VS Code.

To kill all the relevant processes (related to Gazebo and ROS 2), delete build files, delete recorded images and rosbag files using the following command:

```
./control.sh clean
```

To clean up and build the ROS 2 simulation

```
./control.sh build
```

(optionally, in VS Code you can click on Terminal-> Run Task/Run Build Task or use `Ctrl + Shift + B`)

1.3 GPSS controls (pallet trucks, aruco) [Demo]

It is possible for the cameras to detect ArUco markers on the floor and publish their location to TF, both relative to the camera, and the ArUcos transform from origin. The package ./camera_utility/aruco_localization contains the code for handling ArUco detection.

You can also use Nav2 to make a robot_agent (that can be either robot/pallet_truck) navigate by itself to a goal position. You can find the code in simulation/pallet_truck/pallet_truck_navigation

Run these three in separate terminals

```
./control.sh gpss # spawn the simulation, robot_agents and GPSS ArUco detection
./control.sh nav  # spawn map server, and separate nav2 stack in a separate namespace for each robot_agent
./control.sh send_goal # send navigation goals to nav2 stack for each robot_agent
```

If you want to control any robot (pallet truck, humanoid, etc.) manually you can run the following command. Remember to specify what robot you want to control by adding its namespace as argument, i.e. `./control.sh teleop pallet_truck_1`

```
./control.sh teleop ${YOUR_ROBOT_NAMESPACE}
```

If you want to record any of your topics during the tests you can run the following command. Change the topic in the control.sh script: `ros2 bag record /topic` to whatever topic you want to record.

```
./control.sh ros_record
```

To replay your latest recorded rosbag run the following command:

```
./control.sh ros_replay
```

If you want to do a camera dump and save the image from each camera as a .png run the following command. The images will appear at `/src/camera_utility/camera_number`.

```
./control.sh camera_dump
```

If you want to take a screenshot of one of the camera views, run the following command. Replace `###` with the camera you want to take a screenshot of. (163, 164, 165 or 166)

```
./control.sh screenshot ###
```

```
./control.sh birdeye
```

If you want to add the TF links between the cameras and the ArUco markers without running the `gpss` command, you can run the following command. This is primarily useful for debugging, as `gpss` runs this as well.

```
./control.sh aruco_detection
```

Finally, to view the bird's-eye perspective from each camera, run the following command and open `rviz`. Then, navigate to the left panel and under "Camera" change the Topic `/static_agents/camera_XXX/image_projected` to visualize the corresponding camera feed:

1.4 RITA controls (humanoid, robotic arm) [Demo]

```
./control.sh humanoid
```

To move the humanoid around in the simulator

```
./control.sh teleop ${YOUR_HUMANOID_NAMESPACE}
```

1.4.0.1 Arm controls

Spawn the Panda arm inside SIMLAN and instruct it to pick and place a box around with the following commands:

```
./control panda
./control plan_motion
./control pick
```

1.5 Testing

Integration tests can be found inside of the integration_tests/test/ package. Running the tests helps maintain the project's quality. For more information about how the tests are set up, check out the package README. To run all tests, run the following command:

```
./control.sh test
```

1.6 Scenarios

In scenarios.sh you can run predefined scenarios of the project. At the bottom of the file, commands are shown how to run it; otherwise each scenario is referenced by a number. Currently there are 3 scenarios and to run them, run this command in the shell

Before running scenario 1 that uses GPSS cameras, make sure that `CAMERA_ENABLED_IDS` in `config.sh` has the list of GPSS cameras.

```
./control.sh build
./scenarios.sh 1 # navigation using GPSS cameras (real time factor need to be updated)
./scenarios.sh 2 # Humanoid and robotic arm
./scenarios.sh 3 # Humanoid navigation without GPSS cameras (navigate_w_replanning_and_recovery_robot_agent_X.
./scenarios.sh 4 # Record the video demo
```

If you want to see the planning route for all agents, load `scenario_3_planning.rviz` in Rviz.

Keep in mind to change `real_time_factor` in `simulation/simlan_gazebo_environment/worlds/ign_simlan_factory.world.xac` to small values to slow down the simulator (e.g. 0.05-0.1) before building the project. `./control.sh build`

1.7 Advanced options

See resources/ISSUES.md to learn about additional advanced options and to check known issues before reporting any issue or requesting new features. To start the project **without NVIDIA GPU**, please comment out these lines in `docker-compose.yaml` as shown below:

```
#   runtime: nvidia
#
# factory_simulation_nvidia:
#  <<: *research-base
#  container_name: factory_simulation_nvidia
#  runtime: nvidia
#  deploy:
#    resources:
#      reservations:
#        devices:
#          - driver: nvidia
#            count: "all"
#            capabilities: [compute,utility,graphics,display]
```

`camera_enabled_ids` specifies which cameras are enabled in the scene for ArUco code detection and bird's-eye view.

1.7.1 Customized startup

In `config.sh` it is possible to customize your scenarios. From there you can edit what world you want to run, how many cameras you want enabled, and also edit Humanoid-related properties. Modifying these variables is preferred, rather than modifying the `control.sh` file.

1.7.2 World fidelity

in the `config.sh` script, you can adjust the world fidelity

The active worlds are:

arguments	configuration
`default`	Contains the default world with maximum objects
`medium`	Based on default but boxes are removed
`light`	Based on medium but shelves are removed
`empty`	Everything except the ground is removed

1.7.3 Filtering log output

In `config.sh` you can set the level of logs you want outputted into the terminal. By default it is set to "info" to allow all logs. Possible values are: "debug", "info", "warn", and "error". Setting it to "warn" filters out all debug and info messages. Additionally, to filter out specific lines you can add the phrase you want filtered inside `log_blacklist.txt` and setting the `log_level` flag to "warn" or "error" will start filtering out all phrases found in the blacklist.

1.7.4 Older versions

- `gz_classic_humble` branch contains code for **Gazebo Classic (Gazebo11)** that has reached end-of-life (EOL).
- `ign_humble` branch contains code for **ROS 2 Humble & Ignition Gazebo**, an earlier version of this repository.

1.8 Documentation

Learn about the project by reading online documentation page

You can build the online documentation page or a PDF file by running scripts in `resources/build-documentation`.

- `control.sh` script is a shortcut to run different launch scripts, please also see these diagram.
- `config.sh` contains information about which world is loaded, which cameras are active, and what and where the robots are spawned.
- Marp Markdown Presentation
- Configuration Generation
- Bringup and launch files
- Pallet Truck Navigation Documentation
- Camera Utilities and notebooks: (Extrinsic/Intrinsic calibrations and Projection)
- Humanoid Utilities (pose2motion)
- `simulation/`: ROS2 packages
  - Simulation and Warehouse Specification (fidelity)
  - Building Gazebo models (Blender/Phobos)
  - Objects Specifications
  - Warehouse Specification
  - Aruco Localization Documentation
  - humanoid_robot Simulation
  - Geofencing and Collision safe stop
  - Visualize Real Data **requires data from Volvo**
  - Humanoid Control
- `CHANGELOG.md`

- credits.md
- LICENSE (apache 2)
- contributing.md

1.9 Research Funding

This work was carried out within these research projects:

- The SMILE IV project financed by Vinnova, FFI, Fordonsstrategisk forskning och innovation under the grant number 2023-00789.
- The EUREKA ITEA4 ArtWork - The smart and connected worker financed by Vinnova under the grant number 2023-00970.

INFOTIV AB	Dyno-robotics	RISE Research Institutes of Sweden	CHALMERS	Volvo Group

SIMLAN project was started and is currently maintained by Hamid Ebadi. To see a complete list of contributors see the changelog.

2 Dependencies

2.1 Linux Dependencies

2.2 NVIDIA Driver

Use 'nvidia-smi' to ensure that the right NVIDIA driver is installed. If you have not installed **Additional Drivers** when installing Ubuntu, you need to manually install NVIDIA drivers.

2.3 Docker

```
sudo apt install curl git
curl -fsSL https://get.docker.com -o get-docker.sh
sudo sh get-docker.sh
sudo groupadd docker
sudo usermod -aG docker $USER
newgrp docker
```

2.4 nvidia-container-toolkit

To install Docker and `nvidia-container-toolkit`, use the following commands:

```
curl -fsSL https://nvidia.github.io/libnvidia-container/gpgkey | sudo gpg --dearmor -o /usr/share/keyrings/nvi
  && curl -s -L https://nvidia.github.io/libnvidia-container/stable/deb/nvidia-container-toolkit.list | \
    sed 's#deb https://#deb [signed-by=/usr/share/keyrings/nvidia-container-toolkit-keyring.gpg] https://#g' |
    sudo tee /etc/apt/sources.list.d/nvidia-container-toolkit.list

sudo sed -i -e '/experimental/ s/^#//g' /etc/apt/sources.list.d/nvidia-container-toolkit.list

sudo apt-get update

sudo apt-get install -y nvidia-container-toolkit

sudo nvidia-ctk runtime configure --runtime=docker

INFO[0000] Config file does not exist; using empty config
INFO[0000] Wrote updated config to /etc/docker/daemon.json
INFO[0000] It is recommended that docker daemon be restarted.
```

```
sudo systemctl restart docker

sudo nvidia-ctk runtime configure --runtime=containerd

INFO[0000] Using config version 1
INFO[0000] Using CRI runtime plugin name "cri"
WARN[0000] could not infer options from runtimes [runc crun]; using defaults
INFO[0000] Wrote updated config to /etc/containerd/config.toml
INFO[0000] It is recommended that containerd daemon be restarted.
```

Restart the computer to apply the group and user changes.

To check for correct installation of Docker's NVIDIA runtime:

```
docker info|grep -i runtime
 Runtimes: nvidia runc
 Default Runtime: runc
```

Otherwise you get the following error message in VS Code: `Error response from daemon: unknown or invalid runtime name: nvidia`

On a **host** machine's terminal (**not** inside Visual Studio Code terminal): `xhost +local:docker`.

2.5 Windows Dependencies

These instructions are tested on Windows 11, Docker Desktop for Windows ARM64 and VS Code.

To get the Docker container up and running, download and install Docker Desktop for your system: https://www.docker.com/products/ desktop/

When this is done, clone the repo and open the folder in VS Code. Then you should automatically be prompted to download and install VS Code extensions that are needed and recommended for the project.

Once that is completed, make sure Docker Desktop is running and then you should be able to start the container environment in VS Code by pressing Ctrl+Shift+P and searching for Dev Containers: Rebuild and Reopen in Container.

To make Gazebo's and RViz's GUIs visible from the Docker container on your screen, download and install the following program: https://sourceforge.net/projects/vcxsrv/

When this is done, start the XLaunch program and configure it with these settings:

2.5.0.1 Select display settings

```
 Multiple windows
 Fullscreen
 One large window
 One window without titlebar

Display number: -1
```

2.5.0.2 Select how to start clients

```
 Start no client
 Start a program
 Open session via XDMCP
```

2.5.0.3 Extra settings

```
 Clipboard (optional)
 Primary Selection (optional)
 Native OpenGL
 Disable access control
```

Before continuing, make sure Docker Desktop is running.

3 Contributing

Legal Notice: When contributing to this project, you must agree that you have authored 100% of the content, that you have the necessary rights to the content, and that the content you contribute may be provided under the project license.

We try to use:

- Docker, vscode + dev-container extension to develop inside a container
- Markdown for documentation
- Conventional Commits for commit messages
- pre-commit for git pre-commit hooks and basic QA
- Semantic Versioning for versioning
- Draw.io for drawing diagrams for documentation

3.1 Technical

- Avoid optimization unless you have a clear, measurable performance problem, as premature optimization can lead to overly complex and unreadable code.
- PEP 20 – The Zen of Python
  - Simple is better than complex.
  - Readability counts.
  - If the implementation is hard to explain, it's a bad idea.
- Please be generous in giving credit when using an image or a piece of code created by others. Add a link to the original author in credits.md.
- Avoid pushing binary files such as images, models, etc. (use .gitignore), especially if they are constantly changing.

3.1.1 git

- Try to avoid breaking the `main` branch, but don't be obsessed with having a perfect merge request. We can always revert back to the working version or fix the issues. That is why we are using `git`.
- Follow this simple git process:

```
git checkout main
git pull
git checkout -b "branch_name"
# update files
git commit -m "conventional_commit_type: conventional_commit_message"
git push
# Create a pull request.
```

Common commit types:

- feat: Introduces a new feature.
- fix: Patches a bug or issue.
- chore: Routine maintenance or changes that don't affect the app's functionality.
- docs: Documentation changes.

Conventional commit message:

- A conventional commit message uses the imperative mood in the subject line

Example of

- feat(auth): add login functionality
- fix(ui): resolve button color issue

https://www.conventionalcommits.org

3.1.2 Misc

- Don't use absolute paths. Your code should run correctly both inside the VS Code Dev Container and independently outside of it. If you need to modify a configuration file at build time, use relative paths from the project root directory (avoid using ".." as it becomes hard to see which scripts are using a specific directory). For ROS2 resources, use `get_package_share_directory(project_name)` to locate package files.
- When adding a new feature, add it to control.sh. Otherwise, `control.sh` should remain unchanged.
- Any configuration that requires modification should be defined in `config.sh`.

3.1.3 Documentation

Documentation is done in a markdown file. Try to keep our documentation close to your code. Keep the documentation short and clear.

- `#` : Only used once for the title of the project.
- `##` : usually once in each markdown file and usually shares semantics with the markdown filename.
- `###` : Different topics, try to split your documentation into at least 4 topics.
- `####` : subtopics. Try to avoid when you can use simple paragraphs.

Follow this subset of Markdown tags.

4 Config Generation

When building the workspaces, the generation scripts found in the scripts/ directory are run. These automatically generate the `bt.xml`, `control.yaml`, `gz_bridge.yaml`, `nav2_params.yaml`, as well as validate the setup of the `ROBOTS` list in the `config.sh` script. This is done to automatically set up the ROS2 controllers, navigation stack parameters, and other /topics depending on what robots and namespaces are spawned.

Every time the workspace is built, the `generate_XXX` files print that they were generated, and at the top of each generated file, a comment specifies from where the auto-generation occurred. As of now, the parameter files are saved in the package in which they're used instead of directly in the share/ directory to increase code readability for new users.

Explanation of the different generate_XXX files:

- generate_bt_xml.py: The bt_navigator needs the namespace of the robot it is checking conditions for.
- generate_control_yaml.py: The ros2_controller maps namespaced /cmd_vel topics to wheel_joint movements.
- generate_gz_bridge_yaml.py: Some gz_topics need to be mapped to ROS2 topics.
- generate_humanoid_control_yaml.py: The ros2_controller maps namespaced /cmd_vel topics to wheel_joint movements for humanoids.
- generate_humanoid_nav2_params_yaml.py: The individual frame IDs and some ROS2 topics are needed to define humanoids and their corresponding maps.
- generate_nav2_params_yaml.py: The individual frame IDs and some ROS2 topics are needed to define robots and their corresponding maps.
- validate_humanoids.py: A converter from string to list of dicts including a sanity check to throw errors if the HUMANOIDS variable is wrongly configured.
- validate_robots.py: A converter from string to list of dicts including a sanity check to throw errors if the ROBOTS variable is wrongly configured.

5 Simulation

5.1 World Fidelity

It is possible to adjust the level of fidelity for a world in `config.sh`; there the `world_setup` is sent to

- `simulation/simlan_bringup/launch/sim.launch.py`.
- `simulation/simlan_gazebo_environment/launch/simlan_factory.launch.py` (for world generation)
- `simulation/simlan_gazebo_environment/launch/generate_world_file.py` (both `original_world` and the `world_setup`)

5.2 Adding Aruco and camera (static agents)

Aruco codes and cameras are all attached to the same link in Gazebo. To create new static agents, go to `simulation/static_agent_la`
There you only need to add a new line on the form `<xacro:camera number="1" x="3" y="3" z="6" r="0" p="0" w="0"/>`

for a new camera, or a new line on the form `<xacro:aruco number="1" x="3" y="3" z="0.1" r="0" p="0" w="0"/>` for a new aruco. The commands are identical apart from the name.

- `Number` is added to the name to make the static agent unique. For cameras this means they will publish on, e.g., the topic `static_agents/camera_[number]/image_raw`. For aruco the number indicates which aruco png to use.
- `x, y, z` is the coordinate offset from the link the agents are all attached to.
- `r, p, w` are the rotation, pitch, and yaw of the camera, dictating in which direction and at what angle the cameras are looking.

Gazebo supports simulation of camera based on Brown's distortion model. It expects 5 distortion coefficients k1, k2, k3, p1, p2 that you can get from the camera calibration tools. The k coefficients are the radial components of the distortion model, while the p coefficients are the tangential components.

- `<aspect_ratio>` : The ratio of the width and height of the camera.
- `<horizontal_fov>`: The horizontal field of view of the camera in radians.

OpenCV uses five parameters, known as distortion coefficients given by like this: `k1, k2, p1, p2 , k3 # pay attention to the order`.

5.3 Using actors in Gazebo

- Placed in sdf or world file
- Actors use the actor tag (as opposed to the model tag most other objects use).
- actor tags contain links like usual, but also a `<script>` tag.
- The script tag contains:
  - loop: Whether the script should loop on completion
  - delay_start: Time to wait before running the script, also waits between loops
  - auto_start: Whether the script should start automatically when the sim starts
  - A trajectory tag, which has an (unique) id and a type (used to couple it with an animation)
    - Inside the trajectory tags we define waypoint tags which consist of a pose and the time where we are supposed to reach the pose.
    - Note: The trajectory is smoothed as a whole. This means that you'll get a fluid motion, but the exact poses contained in the waypoints might not be reached.

Script structure:

```xml
<script>
  <loop>1</loop>
  <auto_start>1</auto_start>
  <trajectory id='0' type='square'>
    <waypoint>
      <time>0</time>
      <pose>-1 -1 1 0 -0 0</pose>
    </waypoint>
    <waypoint>
      <time>1</time>
      <pose>-1 1 1 0 -0 0</pose>
    </waypoint>
    <waypoint>
      <time>2</time>
      <pose>1 1 1 0 -0 0</pose>
    </waypoint>
    <waypoint>
      <time>3</time>
      <pose>1 -1 1 0 -0 0</pose>
    </waypoint>
    <waypoint>
      <time>4</time>
      <pose>-1 -1 1 0 -0 0</pose>
    </waypoint>
  </trajectory>
</script>
```

- Gazebo supports two different skeleton animation file formats: COLLADA (.dae) and Biovision Hierarchy (.bvh).

  - .bvh: Text-based format with section 'HIERARCHY' and section 'MOTION'.
  - .dae: XML-based format, structured into multiple sections.
- .bvh works for Ignition, while .dae works for both Gazebo classic and Ignition
- Skin is similar, simply import a collada file.
- <interpolate_x>true</interpolate_x> makes the animation match the movement. The animation is done briefly along the x-axis and then it can be interpolated into any direction by gazebo.
- In a top down positive x,y frame of reference: 0 = right, 1.57 = up, -1.57 = down, 3.14 = left
  - Pos/negative matters! 0 to -1.57 does not behave the same as 0 to 4.71!
  - In other words increasing the angle of rotation means rotating left, and decreasing it means rotating right.
- Tension = how strictly it follows the waypoints in a span 0-1 where 0 is not very strict, 1 very strict

5.4 Simulation Specification

Environment (world)

- An in-door factory warehouse
- An out door street/sidewalk (side walk or street) (FUTURE WORK)

Objects

- human actor
- shelves
- standard euro pallets
- pallet racks
- traffic cone
- Boxes
- differential-drive AMR (Autonomous Mobile Robot) (FUTURE WORK)
- forklift (FUTURE WORK)

Textures

- Aruco tag on actor and floor
- objects texture
- Floor texture

Placement of sensors

- camera (depth) on ceiling
- 2D Lidar/camera on AMR (FUTURE WORK)

Physics

- mass
- moment of inertia
- collision
- friction (FUTURE WORK)
- damping

Lighting

Agents movements:

- deterministically following waypoints trajectories (in curve or in form of A-B-C…Z)
- AMR differential-drive using nav2 with obstacle avoidance (FUTURE WORK)
- replaying scenario using rosbag (FUTURE WORK)

Additionally, these are expected from the simulator:

- following realistic and standard measurements and sizes for warehouse, pallettes, shelves, cart, human body, etc
- agent/robot status (position) and control via ROS
- exposing and recording camera images and agent status (e.g. positions) in Python
- tele-operation with keyboard and programmatically from python

5.5 AMR

differential-drive (caster wheel)/Ackermann (car) type of robot

`/cmd_vel` topic accepts `Twist` formatted messages; only non-zero value for linear.x (forward/backward) and non-zero value for angular.z (differential drive, rotating around the z axis).

5.6 Warehouse

The warehouse was created in FreeCAD's BIM Workbench. This workbench isn't available by default, but can easily be acquired by going to `Tools` -> `Addon Manager` -> `BIM` -> `Install/update Selected`.

Using the measurements found in the blueprint in the documentation folder, I drew four lines and turned them into walls (line tool and wall tool respectively). By default the wall will be created around the line, so that the line is in the middle of the wall. This can be changed in the wall properties so that it ends up entirely on one side of the line. As the blueprint lacked walls I used the dimensions specified there as the inner measurements, meaning that the origin point is located at the inner bottom left corner of the wall. All the inner space is in positive x and y coordinates, while the two of the walls are in negative coordinate space. I used the Aligned Dimension tool from the Annotation tools to confirm that the inner measurements matched those of the blueprint.

While the BIM Workbench has support for door objects, they tend to act as solids when imported into Gazebo, so the door is just a hole in the wall. This hole was created by adding a cube object and having it intersect the wall where the door should be located. Then you select the cube and the wall (in that order) in the tree view and press `Remove Component`. This should remove the cube object and create a hole where the cube and wall overlapped. The hole didn't appear in the right place, but it was possible to edit the position of the hole in its properties. I measured the location of the hole using Aligned Dimension to make sure it ended up in the right spot.

Going to the top view, I created a new rectangle object covering the whole warehouse. Then I turned it into a slab with the slab tool to create a floor for the warehouse.

Everything was added to a level object (note: by default this level object is named "Floor", but don't confuse it with the slab that constitutes the physical floor) so that it's grouped together. If we set the wall heights to 0, they will automatically inherit the height of the Level object, so we can change all the walls easily by changing the height of the level.

I added two materials from presets, concrete and wood. I applied the concrete material to all walls and the wood material to the floor. These should be considered to be temporary placeholders.

Finally, I exported the project as a Collada file (.dae) and added it to a simple .world file to see if it loaded properly in Gazebo, and the results seemed correct.

Since the warehouse will be static, we shouldn't need to define any additional parameters like mass or inertia; visuals and collision should be enough. Textures might need some improvement, as currently they're just basic colors, but it should be possible to add those in FreeCAD and have them included in the .dae file so we can load them visually in Gazebo later.

I also added windows for slightly better visibility.

5.7 Add warehouse to world-file

The floor of the warehouse goes below z=0, so the ground plane was lowered by 0.2 so that the warehouse still rests on top of it. As our simulations will mainly take place inside the warehouse, the warehouse floor replaces the ground plane at z=0. The warehouse itself was placed in the models directory and loaded into the world with an `<include>` tag. Additionally, the maze in the stage4 world was moved away from the origin so that it is fully contained inside the warehouse, though in the future it should be removed altogether.

5.8 Textures

Shelf, pallet, warehouse walls are using CC0 textures from https://polyhaven.com/textures Box and warehouse floor are using images from Volvo as a base.

- box: picture from Volvo-group-packaging-specifications_2015.pdf

5.8.1 Conventions

- Keep your links/joints paired, and use the suffix _link and _joint (e.g. arm_link and arm_joint) and maybe follow REP 120 naming conventions.
- Define visual, collision, inertial and Gazebo material (and maybe friction) for all objects

6 SIMLAN bringup

This package is responsible for launching all things related to the simulation. The launch files in the package call launch files in all other packages to start everything up with the following command

```
ros2 launch simlan_bringup full_sim.launch.py
```

This is also called in the `sim` operation in control.sh:

```
./control.sh sim
```

6.1 Launch arguments

There are different launch arguments that can be changed to launch the simulation in different states. These can either be called from the terminal by appending the argument to the launch command above.

```
rviz:='True'
```

Another way to edit what is launched is to change the *default value* in *full_sim.launch.py*.

```
launch_rviz_launch_argument = DeclareLaunchArgument("rviz", default_value="False", description="To
launch rviz")
```

The launch arguments are then either added as a condition straight to a Node:

```
condition=IfCondition(rviz)
```

or passed downward to the launch file being called from the top-level launch file:

```
launch_arguments={"jackal_manual_control":jackal_manual_control,}.items()
```

If you use a specific argument configuration often, it is best to create a new launch file, on top of *full_sim.launch.py*. It can have the correct default values for all arguments and then just call on *full_sim.launch.py*.

6.2 Diagram of launch structure

This diagram is made in DrawIO and the PNG contains the XML code. Drop it into draw.io to make changes and add back to this readme. launch_structure

7 Launching GPSS cameras

This package launches all the static agents, i.e., the cameras in the simulation.

When the workspace is built, the `camera_config.xacro` file is updated with camera_ids and their corresponding intrinsic and extrinsic values. This can be seen in the `build` function in `control.sh`.

The `camera_config.xacro` is then sent to the robot_state_publisher node and published to the /static_agent/robot_description topic. The ros_gz_sim package with the "create" executable then spawns the static_agents by subscribing to the /static_agent/robot_description topic and spawning each agent.

7.0.1 gz_bridge for cameras

Different configurations of the gz_bridge node were tested and evaluated against the simulated RTF (real time factor). In the earlier setups, a gz_bridge node was set up for each camera_stream (image, depth, semantic) for each camera_id. Three configurations were tested where different numbers of nodes were set up and their corresponding RTFs were captured and averaged over five tries.

SN=single node for all cameras, 1NPC=1 node per camera_id, 1NPSPC = 1 node per stream per camera_id

node setup	number of cameras	number of nodes	camera streams	avg RTF
SN	9	1	image	17,9%
1NPC	9	9	image	16,7%
SN	9	1	image depth semantic	5,14%
1NPC	9	9	image depth semantic	5,2%
1NPS	9	81	image depth semantic	4,7%

It doesn't seem like a big difference, so we will stick with one node, i.e., SN.

In the future it can be decided whether to keep the gz_bridge node as is or to generate a gz_bridge_camera.yaml file like the other generation scripts (`config_generation`).

8 Bird Eye View package

As preparation, make sure that the right cameras are selected in the `config.sh`

```
## CAMERA_ENABLED_IDS can be set as a string of camera_ids separated by space ' '. valid camera ids are 160-17
CAMERA_ENABLED_IDS='164 165 166 167 168'
CAMERA_STREAMS='image'
```

8.0.1 Bird eye view

This package features the ability to select areas of your choosing and create a bird-eye view of that area by using available cameras, viewing that area.

Ro run birdeye launch file in separate terminals run the following commands:

```
./control.sh build
./control.sh sim
./control.sh static_agents
./control.sh birdeye
```

bird eye view bird eye view gazebo

8.0.2 Different sensors (Depth, Semantic Segmentation, Color)

The package also includes a camera save node. This node saves raw, depth, and semantic segmentation images for all the enabled camera IDs in config.sh. To take a single image from different cameras, run the following command:

```
CAMERA_ENABLED_IDS='164 165 166 167 168'
## CAMERA_STREAMS can be set as a string of stream options separated by space ' '. valid camera ids are image
CAMERA_STREAMS='image depth semantic'
```

Keep in mind to change `real_time_factor` in `simulation/simlan_gazebo_environment/worlds/ign_simlan_factory.world.xac` to small values to slow down the simulator (e.g. 0.05-0.1).

to record a camera streams as a sequence of images or a video run the following command in separate terminals:

```
./control.sh build
./control.sh sim
./control.sh static_agents
./control.sh save_depth_seg_images
./control.sh save_depth_seg_videos
```

The resulting videos are stored in `/simulation/camera_bird_eye_view/recordings/`

color

semantic

depth

9 Aruco Localization Package

This is the `aruco_localization` package, which runs a ROS2 node that takes images from camera topics `CAMERA_X/camera_info` and `CAMERA_X/image_raw`, processes the ArUco code, and publishes the result in the `/TF` of the detected ArUco marker, mimicking Volvo's logic for tracking robot positions and generating navigation routes. A launch file is provided to run multiple nodes, with each node dedicated and assigned to a single camera. The `camera_enabled_ids` variable found in `config.sh` is used to control which cameras are enabled.

Note: The marker link is named based on the camera. For example, if camera `164` detects an ArUco marker with ID of `12`, the marker link in `/tf` is named `camera_164_marker_12`. This is not the final link that is used. The result from different cameras are accumulated and merged as a single frame with parent as `robot_agent_12/odom` and child `robot_agent_12/baselink`.

Note: The marker ID is tightly linked with the name for the robots, meaning an ArUco with marker_ID=1 will detect the robot with name: {namespace}/{marker_ID}. For example 'robot_agent_1'.

9.1 Important points to pay attention to:

- If two cameras are pointed at the same ArUco code, the system does not alternate between them, but instead shows the midpoint between the two detections.
- Inside the aruco_node, a callback logs the relative position of the newly created `aruco_link` from the `base_link`.
- Multiple cameras can be linked to the same ArUco code simultaneously.

9.2 Important launch files

- `aruco_detection_node.py`: Handles ArUco detection and publishes the TF between the camera and marker. (e.g. `camera_164_marker_12`)
- `aruco_pose_pub.py`: A new node that listens to all transforms of `camera_164_marker_12` as input, and outputs the transform to either TF or ODOM as `robot_agent_12/odom` and child `robot_agent_12/baselink`.
- `multi_detection.launch.py`: Launch file for the new node.

9.3 Odom

These have to be valid for both humanoid and pallet trucks

- world: is the origin (0,0,0)
- odom: is in the initial pose of the robot
- base_link: tracks the movement of the robot relative to the odom

Coordinate

10 Failsafe for the navigation

The SIMLAN systems support a failsafe and geo-fencing mechanism for the pallet trucks' navigation. The main idea is that the pallet trucks should stop the navigation as soon as a dangerous situation below is detected:

In this approach for a new Behavior Tree, the **condition node** named `StopRobotCondition` is defined and can be triggered when a safety issue occurs. We then added a Behavior Tree plugin that calls an action called CancelControl when the given condition is triggered.

Currently, this safety controller triggers `StopRobotCondition`:

10.0.1 Activation of collision sensor

When a collision is detected by the simulator, the collision's physical properties (such as the force and the object that the pallet truck collided with) are published to the pallet trucks' `/contact` topic.

10.0.2 Loss of Observability

To implement geofencing and the safety situation in which a pallet truck is not observable in any camera, the `aruco_localization` pkg under `aruco_localization/aruco_pose_pub.py` continuously publishes the list of pallet trucks that are observable in the `/aruco_marker_seen` topic. Then, if one stops being observable, the BT condition passes.

10.0.3 Behavior tree and direct implementation in aruco_localization pkg

At first, we define a custom `behavior_tree.xml` in `simulation/pallet_truck/pallet_truck_navigation/config/navigate_w_rep` This XML file defines which BT plugins we want to use during the navigation and which run continuously during the navigation. In this XML file, the `StopRobotCondition` and `CancelControl` plugins are defined in a fallback function. A fallback function works as it runs the first stated plugin, and when that plugin fails, it moves over to the second one and executes that plugin. So in this case, we first run the `StopRobotCondition` plugin until the robot is out of bounds. Once the plugin fails, the `CancelControl` plugin executes and stops the pallet truck.

In order to know whether a pallet truck is out of bounds or not, it is determined by looking at the aruco marker on the pallet truck and deciding if it's seen by any of the cameras. This is implemented in the `aruco_localization` pkg under `aruco_localization/aruco_pose_pub.py`. As long as the aruco marker on the pallet truck is seen by any of the cameras, the node publishes the robot namespace in a list to the topic `/aruco_marker_seen`.

The same goes for the collision sensor. It publishes to the topic `/robot_agent_X/contact` when the pallet trucks collide with something.

Step two in the failsafe is to stop the pallet truck when it gets out of bounds or collides with something. This is done by a custom-made behavior tree plugin called StopRobotCondition, which is found in `simulation/bt_failsafe/src/stop_robot.cpp`. This plugin listens to the `/aruco_marker_lost` and `/robot_agent_X/contact` topics. As soon as a contact is detected or a robot's namespace is lost, it fails, and the `CancelControl` behavior tree starts. This, in turn, stops the pallet truck. The `CancelControl` plugin is an existing built-in plugin in Nav2.

10.0.4 Why Nav2 behavior tree plugins were not suitable

We tried to use only built-in plugins, which most likely is the most robust and secure way to do it as the plugins are updated accordingly to ROS2 and Nav2.

We tried to use several plugins to detect if the pallet trucks are out of bounds, but none of them really suited this purpose of this project.

The first one we tried to use is the `TransformAvailable` plugin. This plugin checks if a certain TF exists. If the TF is missing, the plugin returns `FAILURE`. So in this case, we thought we could look at the TFs between the cameras and the aruco marker, and if none of them exists, it should fail and stop the pallet truck. Unfortunately, we couldn't use this plugin because it only looks at the TF from the very beginning of the navigation. And if it exists from the beginning, it will always succeed and will never return `FAILURE`. Therefore, it cannot be used in our case because we have TF from the beginning, and our TF disappears after some time during the navigation. You could probably modify the plugin to work for our case as well, but if modifications are needed, it could be implemented in a better way instead.

The second plugin we tried is the `IsStuckCondition`. This plugin checks if the robot is stuck by calculating the deceleration of the robot. If the deceleration is too big, it returns `FAILURE`. This wasn't anything we could use because the acceleration is set to zero as soon as the robot gets out of bounds. So this is probably not suitable for this case.

11 Pallet Truck bringup

This package handles initialization and status of spawned robots.

- gazebo.launch.py - Spawns robot, handles robot_state_publisher, robot_description
- keyboard_steering.launch.py - keyboard steering.
- multiple_robot_spawn.launch.py - contains configurable list of robots you spawn in the sim.
- sim.launch.py - main launch file for single robot. Make sure Gazebo is running; control, twist_mux, and keyboard_steering are all launched.
- rviz.launch.py - runs RViz

The package focuses on pallet_truck and forklift for now, but could be made modular to spawn other types of robots.

11.1 Configuring and spawning robots inside the sim

Below is the structure which we use to spawn robots inside of the sim. Please read these notes before setting up a new or editing a robot.

Here is the information from the table presented as a markdown list:

11.2 Robot Spawning Attributes

- **`namespace`**
  - **Description:** Used to differentiate between multiple robots. Follows the format `robot_agent_N`, where `N` is the robot ID.
  - **Example:** `"robot_agent_1"`
- **`initial_pose_x`**
  - **Description:** The robot's initial x-coordinate position. **Float value** wrapped as a string.
  - **Example:** `"10.0"`
- **`initial_pose_y`**
  - **Description:** The robot's initial y-coordinate position. **Float value** wrapped as a string.
  - **Example:** `"1.0"`
- **`robot_type`**
  - **Description:** Selects the robot mesh or appearance. Options are **`"pallet_truck"`** or **`"forklift"`**.

  - **Example:** `"pallet_truck"`
- `aruco_id`
  - **Description:** Sets the ID shown on the robot's ArUco marker. Must match the ID in the `namespace`.
  - **Example:** `"1"` if namespace ID is `1`

Would you like me to use these attributes to create a sample robot configuration?

```
Example robot setup:
{
    "namespace": "robot_agent_1",
    "initial_pose_x":"10.0",
    "initial_pose_y":"1.0",
    "robot_type":"pallet_truck",
    "aruco_id":"1"
}
```

11.3 Automatically generated parameter files

Some parameter files are automatically generated for the pallet_truck_bringup package. The generation scripts can be found in the config_generation/ directory. The automatically generated files include the `nav2_params.yaml`, `gz_bridge.yaml` and more. To look at all the automatically generated files, build the workspace and the generated files will be printed in the terminal or look manually in the config_generation/generate.py.

11.4 Pallet Truck Package

Common packages for pallet_truck, including messages and robot description. These are packages relevant to all pallet_truck workspaces, whether simulation, desktop, or on the robot's own headless PC.

Links to read about each individual pkg - pallet_truck_bringup - pallet_truck_control - pallet_truck_description - pallet_truck_navigation

`prefix` is used to publish unique base link names of each robot agent to `/tf`. This way they are all visible in rviz and probably it is used for odometry done by aruco localization.
`namespace` is used to separate nodes and topics related to each robot agent. This way we can control each robot separately and run a separate nav2 stack.

Keep in mind that the value of `namespace` is used for `prefix` but they are different concepts and use cases.

12 Pallet_truck_description

In addition to the change_log, this description is added to give vital information that may help other developers in the future to debug or update features faster.

Here additions to the urdf.xacro files are mentioned

12.0.1 Collision sensor

A collision_sensor was added and linked to the mesh tag of the pallet_trucks. This is integrated in the pallet_truck.urdf.xacro.

To visualize the topic subscribe to: /namespace/collision e.g. /robot_agent_1/contact.

The topic publishes information about the position, torque and which models are in contact with each other.

The topic name is defined by the automatically generated gz_bridge() which can be found in /home/ros/src/simulation/pallet_truck/p

12.0.2 xacro/urdf/sdf files Bugs

Gazebo only reads sdf files and if .xacro or .urdf files are used, they are later parsed to .sdf's before being used by Gazebo.

A "bug" for Gazebo harmonic using ros2 jazzy was that when adding the collision sensor for the pallet_trucks, Gazebo was unable to find the collision tag for the mesh that we were using.

After investigation it was found that when the .xacro file was parsed to an .sdf file, the name of the collision tag was updated by the parse plugin. Therefore this lumped renaming of the collision tag was "hard coded" instead of using its original name which is in the pallet_truck.urdf.xacro.

---

original: “$(arg prefix)/chassis_link::collision”

hard coded: “${prefix}_base_link_fixed_joint_lump__collision_collision”

When adding sensors in the xacro file, make sure the plugins are also loaded. There are different world and robot plugins so if system plugins are used they should most likely be added to the .world file

13 Pallet Truck Control

The `pallet_pallet_truck_control` package includes all nodes necessary for the control of the robots. These include the following:

- **twist_mux**: Takes in velocity topics and orders them in order of relevance. from highest to lowest: /key_vel, /safety_vel, / scenario_vel, /nav_vel, /cmd_vel
- **twist_stamper**: As of ros2 jazzy the ros2_controllers need the twist messages to be stamped, therefore the stamper was introduced
- **ros2_control_node**: Main node which uses the control.yaml file and maps “hardware” to controller actions.
- **spawner-joint_state_broadcaster**: Publishes the joint_state so the robot moves in gazebo and rviz.
- **spawner-velocity_controller**: Accepts and publishes velocity commands

With this setup, the robot_agents can be controlled by running the teleop command in control.sh.

14 Navigation (Robot agent and Humanoid) and Map server

the pallet truck navigation consist of: - map_server.launch.py - starts the map server and creates necessary transforms for all of the robot agents. - nav2.launch.py - starts the navigation stack and the update_map.py for each robot. - cartography.launch.py - starts cartography. - update_map.py - adds the dynamic obstacles (other robot_agents) to each robot_agent’s map.

To run the navigation stack do the following either with the input=ROBOTS or HUMANOIDS

```
./control.sh nav ROBOTS
```

14.1 Map server

Instead of starting with an empty map, to get the general layout of the environment and detect the static object in the environment (walls, cooridors and hallways) you can **optionally** build a map. To start mapping (otherwise known as cartography), you can use `cartography.launch.py`. It requires a lidar to be mounted on the `robot_agent` and `odometry` to exist, which can be activated from the `camera_utility/aruco_localization` package.

Launch the following package, run gazebo, rviz, aruco_localization, and start moving around the simulation with the robot_agent. When you are finished you need to save it using this command: `ros2 run nav2_map_server map_saver_cli -f simulation/pallet_truck/pallet_truck_navigation/maps/YOUR_MAP_NAME`. Keep in mind that this step needs to be done only once and the map assumed to be static.

The navigation stack uses a base map which is updated dynamically with other robot_agents for each robot. This done to add the static environment and update the dynamic moving obstacles later.

```
Node(  # Manually setting the joint between map and odom to initial_pose_x initial_pose_y 0, i.e. the location
    package="tf2_ros",
    executable="static_transform_publisher",
    namespace=robot["namespace"],
    name="static_map_to_odom",
    arguments=[f"{robot["initial_pose_x"]}",
                f"{robot["initial_pose_y"]}",
               "0",
               "0",
               "0",
               "0",
               f"{robot["namespace"]}/map",
            f"{robot["namespace"]}/odom"],
)
```

14.2 Robot_agent navigation

This package covers the functionality of mapping, localizing, and navigation for the robot_agent. Each package is built using code from the nav2 packages, thus you are able to modify the launch files but to change the node's behaviour you need to modify the config files. The code is tailored for the robot_agent and using the `aruco_detection` package that supplies `/TF` chain for the truck.

- The `aruco_localization` package runs the aruco localisation.
- The `map_server.py` creates a map for each agent
- The `nav2.launch.py` uses the localisation (from the previous steps) for the navigation of the robot_agent.

Keep in mind that the same namespace has to be used when launching the navigation stack which is done automatically: `ros2 launch pallet_truck_navigation nav2.launch.py robots:=ROBOTS`

For navigation to be able to automatically find the robot agent, we have to set a namespace that specified when launching the robot_agent. (see above)

To do navigation for both `humanoids and pallet_trucks` at the same time you need to create a merged variable of the 2 and sent that as a input to the map_server.launch.py and nav2.launch.py to make sure the obstacles are detected.

Each robot has it own navigation configuration in `nav2_params.yaml` which is dynamically generated on build based on the `ROBOTS` and `HUMANOIDS` variables in config.sh. the pallet_trucks and forklifts also have an individual `navigate_w_replaning_and_recovery_robot_agent_x.xml` which is supposed to stop the navigation of the agents if their aruco_markers are not detected. To visualize which aruco markers are seen, first run the gpss and nav operations and then echo the `/aruco_marker_seen` topic.

We use following `/TF` structure:

view_frames.png

Note that `world` and `map` are static at the same position. `robot_agent_X/odom` is static at the position from where the robots are spawned and the new position of the robots are then determined by `robot_agent_X/base_link` which is a dynamic transform from `robot_agent_X/odom`.

Good video to watch for multi agent navigation: https://www.youtube.com/watch?v=cGUueuIAFgw

14.3 Humanoid_navigation

The humanoid navigation is implemented in the same way as for the robot_agents with the difference of calling the function `./control.sh nav HUMANOIDS`. Then the same principles apply but with a different nav2_parameter file being used as an argument to nav2 nodes.

There is one major difference between the navigation of the humanoids and the robot_agents. The robot _agents get their `odom` frame from the aruco_localization pkg which find aruco_markers and publishes their orientation and position. The humanoids on the other hand get their `odom` frame from the humanoid_odom_pub node which subscribes to the ground truth of the humanoids pose from gazebo and creates a dynamic transform between the `namespace/odom` -> `namespace/base_link` frames. This means the robot_agents get their odom frame from what the cameras can see and the humanoids get their odom frame from the actual pose in the simulation.

14.4 Obstacles detection (update_map_node)

The obstacle detection is based on a static map which is dynamically updated with all other robot agents except itself. Therefore every robot_agent has its own map to not map itself. To not depend on the orientation of the robots, the obstacle is depicted as a circle. With this setup the map_server is gets an action call that updates the moving obstacles

---

The `/map_updater/update_map_node.py` node works as following:

First it makes a copy of the `warehouse.pgm` map which is a long array of pixel values ranging from (0-255). Since the map nav2 needs has a different setup than the `.pgm` file, a conversion is needed.

The .pgm file has these pixels ranges

value	meaning
0	black obstacle
255	White Free space
1-254	ranges of gray, not defined at the moment

The map nav2 needs is set up with pixels ranging from (-1-100) where

value	meaning
-1	Unknown
0	Free space
100	obstacle
1-99	probabilistic occupancy

therefore this has to be parsed to match by doing the following:

```
current_array = self.original_array.copy()
occupancy_array = np.zeros(current_array.shape, dtype=np.uint8)
occupancy_array[current_array.copy()<100] = 100
```

The map is a 2D array, typically stored row-major from bottom-left, but many implementations flip it vertically ([::-1]) to match how image viewers treat top-left as (0,0). and therefore the following is done

```
occupancy_array = occupancy_array[::-1, :]
```

The position of all other robot_agents are found by their transforms between robot_agent_x/base_link and world and an obstacle is set at that position which is updated with 1 Hz to continuously update their positions. This can be tweaked but to limit CPU usage it was set to 1 Hz.

This concept is repeated for all namespaces sent through the ./control.sh nav function.

14.5 Future improvements

How often the new path the pallet_trucks can take be update Add a speed limiter instead of obstacle, in those cases the robot will slow down instead of replanning the paths. Could probably mix with the inflation radius's of the costmaps to make robot take wider turns

Collision Two robots that are approaching each other (from left and right) don't know where each one is planning to go. Therefore they might both plan to the same path and both trie to out turn each other resulting in a race condition.

Possible solution: block future path on every other robots map!

15 Humanoid Robot

There needs to be better documentation of the difference between this pkg and the humanoid_support_moveit_config pkg. Is this supposed to be the bring-up of that pkg? Why is the humanoid model in this pkg and the URDF that spawns the robot in humanoid_support_moveit_config? Are the humanSubject01-08 used, or is only humanSubjectWithMeshes used? Clarification is needed. All in all, a restructure of the entire humanoid_robot and humanoid_support_moveit_config should be done.

Package created based on this tutorial
Note that only the human with mesh model is installed in the package; see /model/CMakeLists.txt for how to install the non-mesh models.

In the URDF, the fixed links and joints are (mostly) named [muscle name]_[location], for example, BicBrac_RUA = biceps brachii_right upper arm. In humanSubjectWithMesh_simplified.urdf, all fixed joints have been removed; only the skeleton joints remain.

To open Rviz with the human model:

```
ros2 launch urdf_tutorial display.launch.py model:=/home/ros/src/simulation/humanoid_robot/model/human-gazebo/
```

16 Humanoid

In this file, the basics of the humanoid structure will be described. In the future, all nodes will be namespaced to add the functionality of spawning and controlling multiple humanoids.

16.1 Moveit

For MoveIt to work, 3 components must have correct information:

- URDF link names
- SRDF joint definitions
- TF frames published by robot_state_publisher

There was an issue when trying to add namespaces to the humanoid project. Since the humanoid is made of 40+ links, it was decided to add the `"frame_prefix": f"{namespace}"` in the robot_state_publisher node. This, however, made a conflict since the URDF, SRDF, and TF frames no longer matched. This was solved by adding `base_link` in between the world and namespace/base_link frames and creating a static transform between them. **This means that all further humanoids will be spawned under the `base_link` frame.** (otherwise, the warning below again appears)

16.2 Dynamically updated URDFs

To make the multiple_humanoid_spawn work, we need to be able to launch different namespaces in the robot_description. This is done by running:

```
moveit_config = (
    MoveItConfigsBuilder("human_support", package_name="humanoid_support_moveit_config")
    .robot_description(
        file_path="config/human_support.urdf.xacro", mappings={"namespace": namespace}
    )
    .to_moveit_configs()
)
```

This updates the `namespace` argument inside the .xacro files.

16.3 Planning scene monitor

When working with the MoveIt package, some bugs or undesired features were found. When MoveIt is launched, the PlanningSceneMonitor and PlanningFrame subscribe to all TF frames that exist and assume they can transform any known object or sensor data into its planning frame, which is `base_link` in this case. If this cannot be done, it will throw a warning saying the following:

```
[WARN] [humanoid.moveit.moveit.ros.planning_scene_monitor] [id]: Unable to transform object from frame 'unconn
```

To limit this, a wait function was added: `ld.add_action(TimerAction(period=5.0, actions=[rsp_node]))` to make sure the transform is published before the rsp_node starts.

16.4 Figures

My Robot Diagram *Figure 1: Humanoid frames visualization*

16.5 Rviz2 visualization

If you want to visualize the movement in Rviz, you need to configure the `config/moveit.rviz` file and change all `/humanoid_X` instances to the namespace you want to visualize.

16.6 Bugs: TF visualization and Gazebo simulation not matching

When visualizing and comparing the actual TF data and simulated locations of the humanoids when doing navigation and teleop, it can be seen that these do not match. This is a major issue since that means the location of where the humanoids think they are in the map and the location of where the humanoids actually are in Gazebo will be different. This can be visualized by turning the humanoid 360 degrees in Gazebo with the `./control.sh teleop humanoid_1` command and comparing it to Rviz. There, the humanoid has turned closer to 300 degrees.

This problem probably comes from some URDF descriptions not matching the actual geometry or specifications the humanoid has in `/home/ros/src/simulation/humanoid_support_moveit_config/config/human_support_wheels.urdf.xacro`. By tuning the `mass`, `mu`, and `inertial` values in the `<xacro:macro name="wheel" params="wheel_prefix *joint_pose">` xacro tag, it was possible to tune the degree mismatch.

For proper navigation and control, this needs to be fixed!

17 Humanoid Motion Capture

This project develops a system for translating human pose detection to humanoid robot motion in simulation environments. Using Google MediaPipe for pose landmark detection from camera input, the system maps detected human poses to corresponding joint movements executed by a humanoid robot in the Gazebo simulator. The implementation leverages ROS2 Ignition and MoveIt2 for motion planning and control, with a data generation pipeline that creates training pairs of pose landmarks and robot joint configurations. This approach provides a foundation for safety monitoring applications in industrial simulation (SIMLAN), where human pose analysis can be integrated for workplace incident detection. The work is based on a master's thesis "Human Motion Replay on a Simulated Humanoid Robot Using Pose Estimation" by Tove Casparsson and Siyu Yi, supervised by Hamid Ebadi, June 2025.

17.1 Pose to humanoid motion

This project employs a deep neural network to learn the mapping between human pose and humanoid robot motion. The model takes 33 MediaPipe pose landmarks as input and predicts corresponding robot joint positions.

17.2 Terminology:

- **Forward Kinematics (FK)**: The process of calculating the position and orientation of a robot's links given the values of its joint parameters (e.g., angles or displacements). In other words, FK answers the question: "Where is the robot's hand if I know all the joint values? (usually have one answer)"
- **Inverse Kinematics (IK)**: The process of determining the joint parameters (e.g., angles or displacements) required to achieve a desired position and orientation of the robot's links. In other words, IK answers the question: "What joint values will place the robot's hand here? (usually have many answers)"
- **Pose** : 3D pose landmarks (MediaPipe) extracted by MediaPipe from a 2D image of human posture
- **Motion** : A kinematic instruction (joint parameters) sent to the robot (via MoveIt) for execution. While "kinematic instruction" would be a more accurate term, we continue to use "motion" for historical reasons (used in the word motion-capture), even though "motion" often refers to the difference/movement between two postures (e.g., posture2 - posture1).
- **Motion Capture**: Here it means using 2D images to find the motion (kinematic instruction/joint parameters) to instruct a humanoid robot to mimic the human posture.

17.3 Dataset generation

To build the synthetic dataset, first ensure that the cameras pointing to the humanoids are active in the `./config.sh`.

```
CAMERA_ENABLED_IDS='500 501 502 503'
```

and rebuild the project with these commands:

```
./control.sh build
```

The image below describes how the dataset generation system works.

Dataset generation overview

To create a humanoid dataset (paired pose data, motion data and reference images) in the `DATASET/TRAIN/` directory:

```
./control.sh dataset TRAIN/
./control.sh dataset EVAL/
```

To find the implementation of how the dataset is created, go to pre_processing/ directory.

When the data is transformed into numpy/tabular format, each row in the CSV file represents **multiple pose** samples **from each camera** and its corresponding robot motion. The number of columns depends on the number of cameras, and it's possible to select what camera inputs you want to use; i.e., just one camera input or up to four camera inputs.

- The first columns are the 3D coordinates (x, y, z) for each of the 33 MediaPipe pose landmarks and **for each camera**, named like `cam500_0_x, cam501_0_x, cam502_0_x, cam503_0_x, cam500_0_y, cam501_0_y, cam502_0_y, cam503_0_y, cam500_0_z, cam501_0_z, cam502_0_z, cam503_0_z....`

- The remaining columns are the robot joint positions (motion targets) for that sample, with names like jRightShoulder_rotx, jLeftElbow_roty, etc.

17.4 Prebuild datasets

We use `20251028-DATASET-TRAINONLY-MULTI.zip` (place it in `DATASET` for training) and `20251028-DATASET-EVAL1000-MULTI.zip` (place it in `input/` directory for prediction) that are available at SharePoint as our common datasets.

17.5 Raw dataset shape

```
DATASET:
  TRAIN
      camera_500
         pose_data : 1.json , 2.json
         pose_images : 1.png , 2.png
      camera_501
         pose_data : 1.json , 1.json
         pose_images : 1.png, 2.png
      motion_data: 1.json, 2.json
```

17.6 Loading Dataset

For training and prediction, we have a helper library that loads the data into these two primitive numpy arrays. All training must use these two data arrays with the following shape:

```
motion_NP
-------------
scenario_id, motion_joint[1-47]
1, [JDATA_a]
2, [JDATA_b]

pose_NP
------------
scenario_id, cam_id, mp_landmark[1-99]
1, 500, [LDATA_a]
1, 501, [LDATA_b]
2, 500, [LDATA_c]
2, 501, [LDATA_d]
```

17.7 Models

17.7.1 PyTorch

We have more control over the PyTorch model. It has its own implementation of dataset and model definition, located inside utils.py. What is specific about using PyTorch is the possibility to use Optuna as a hyperparameter optimization tool. This tool helps find the best suitable set of hyperparameters given its training data. AutoGluon has its own internal optimization; thus we only use Optuna for PyTorch.

- Input Layer: 33 MediaPipe pose landmarks for each camera (x, y, z coordinates).
- Hidden Layers: Multi-layer perceptron with pose normalization preprocessing
- Output Layer: Robot joint position sequences for humanoid motion control, this results in a total of 47 joints to be outputted.
- Training: Supervised learning on pose-motion paired datasets.

17.7.2 AutoGluon

We use Autogluon tabular predictor. This model is trained as an ensemble (collection) of models where each separate model aims to predict a single joint given the complete input poses. The result from each model is then merged together and becomes a predicted list for each target joint.

17.8 Model Training

To train a model run the command below. The input size depends on the number of cameras you plan on using, meaning that if you only select one camera the input shape will take **1** camera into account. If you select **4** the model will have a **4 times** larger input size. To set it up in the way you want, go to `config.sh`. The variables are:

- `$model_instance`: If you want to reuse a model, specify its name here. Keep blank if you don't want to save.
- `$dataset_cameras`: The cameras you want to train on. This can be a list of cameras i.e. "500 501 502". For single training set this to one ""
- `$model_type`: Possible selections: pytorch, autogluon

Finally you specify what JSON directory you want to use as training data which becomes the second argument, below is an example of running train using the TRAIN/ dataset.

After a training run is complete, the model is saved inside of `pose_to_motion/{$model_type}/output/saved_model_states`. A summary report of the session is also generated and will be saved inside `pose_to_motion/{$model_type}/output/train_report.cs`
Ensure that the `./config.sh` is properly configured.

```
# The cameras you want to train on. This can be a list of cameras i.e. "500 501 502". For single training set
dataset_cameras='500'

# Possible selections: pytorch, autogluon
model_type="pytorch"

# If you want to reuse a model, specify its name here. Keep blank if you don't want to save.
model_instance="pytorch_test_pred_500"
```

> Keep in mind that we use a separate virtual environment to install machine learning related pip packages called `mlenv` with a separate `requirements.txt`. Build the environment first using `./control.sh mlenv`

```
./control.sh mlenv
./control.sh train TRAIN/
```

17.9 Model Evaluation

Running `control.sh eval` in the terminal runs the evaluation pipeline. This will trigger the selected `model_type`'s evaluation pipeline. To set it up in the way you want, go to `config.sh`. The variables are:

- `$model_instance`: If you want to reuse a model, specify its name here. Keep blank if you don't want to save.
- `$dataset_cameras`: The cameras you want to eval on. This can be a list of cameras i.e. "500 501 502". For single training set this to one ""
- `$model_type`: Possible selections: pytorch, autogluon

The evaluation will run metrics defined in metrics.py; currently MSE and MAE are calculated. The results are then saved in a summary report located in `pose_to_motion/{$model_type}/outputs/eval_report.csv`.

To evaluate a trained model, there is a command that evaluates based on a set of metric. In the example below it evaluates on the EVAL/ JSON-dataset:

```
./control.sh eval EVAL/
```

As the result MSE and MAE values are printed.

17.10 Model Prediction

The input format needs to be structured as the template below. Please note the "dataset_name" can be whatever you want to describe the content of the data. The camera names are connected with the variable `dataset_cameras` inside config.sh. Lastly, this folder needs to be placed inside the input/ directory.

```
DATASET_NAME_TEMPLATE_SOME_DESCRIPTOR/
    camera_500/
        file_1.(JSON, PNG, JPG, MP4)
        file_2.(JSON, PNG, JPG, MP4)
    camera_501/
        file_1.(JSON, PNG, JPG, MP4)
        file_2.(JSON, PNG, JPG, MP4)
    camera_502/
        file_1.(JSON, PNG, JPG, MP4)
        file_2.(JSON, PNG, JPG, MP4)
      camera_503/
        file_1.(JSON, PNG, JPG, MP4)
        file_2.(JSON, PNG, JPG, MP4)
```

The prediction pipeline uses these variables in the `config.sh`:

- `humanoid_input_dir`: Input directory. Images or videos in this dir can be predicted.
- `humanoid_output_dir`: Output directory. pose files and predicted motions are placed here
- `dataset_cameras`: What cameras to take into concern for prediction.
- `model_instance`: What trained model instance to use.
- `replay_motion_namespace`: What humanoid you want to replay motions on.

The prediction pipeline is implemented in a way where it expects the input dataset to follow the exact format as the model was trained on; see the note above for reference. It is possible to predict on three different types of data: `images`, `videos`, and raw `JSON` data. A good folder to put the data is inside the `input/` directory.

The input dataset gets processed into JSON format and is saved inside the `output/` directory, using `processes_input_data.py`. It creates the pose json files and MediaPipe images. `predict.py` inputs the pose json files and generates the predicted motions by saving them into a directory called `$humanoid_output_dir/dataset_name/predicted_motions/`. Running `control.sh predict $mode $dataset_name` inputs all poses for each camera in JSON format and for each pose_data outputs a predicted motion, which is replayable in RViz along with a ground truth pose_image. Finally, all motions are inputted sequentially into the motion viewer and visualized inside of Gazebo.

To run predict successfully, ensure that the variables in `config.sh` are set and place the data you want to predict inside `$humanoid_input_dir`. Then use the folder name of the dataset you want to predict as an argument along with the format of the data.

For testing make sure that

- `input/TEST` (with DATASET_NAME_TEMPLATE_SOME_DESCRIPTOR explained above) is present (**`input/` and not `DATASET/`**)
- `input/VIDEOS/camera_500/20250611video.mp4`

To apply the predicted motion to your preferred humanoid adjust the `config.sh`:

```
replay_motion_namespace="humanoid_2"
dataset_cameras='500'
```

Finally run the command below:

```
./control.sh predict TEST/
./control.sh predict VIDEOS/
./control.sh predict IMAGES/
```

17.11 Project Structure:

- `humanoid_ml_requirements.txt`: Contains the humanoid machine learning requirements
- `input/` : This folder has the pose data to be predicted
- `output/` : This folder saves the predicted motion data and intermediate results
- `/DATASET/TRAIN`: Contains motion, pose and image files generated by `./control.sh dataset TRAIN`
  - `motion_data` : Corresponding random motion (request)
  - `camera/pose_data` : Mediapipe pose (result) for each camera
  - `camera/pose_images`: Mediapipe annotated images and original images for each camera
- `/DATASET/EVAL`: Contains motion, pose and image files generated by `./control.sh dataset EVAL`
  - `motion_data` : Corresponding random motion (request)
  - `camera/pose_data` : Mediapipe pose (result) for each camera
  - `camera/pose_images`: Mediapipe annotated images and original images for each camera

- pre_processing/: contains all pre-processing files, used to convert the JSON data into tabular data. This will be the input of the model.
  - process_input_data.py: Handles pose detection from actual images or videos using MediaPipe
  - dataframe_converter.py: Contains helper functions to convert json files into dataframe/numpy format and is used by the models. Running its main will generate csv files that contains both pose and motion data from the json files. Inputs the cameras you want to use, the json dir, the csv filename to generate.
- pose_to_motion/: Contains the ML training pipeline for pose-to-motion prediction
  - model.py: The main file containing implementations for training, evaluating, and predicting motions. This file acts as an orchestrator and inputs the data, what model to use, and what operation should be done. It is responsible for handling commands.
  - autogluon/: directory containing model implementation for autogluon model. Handles training, evaluation, and predicting motions.
    - framework.py: Framework file, contains the implementation for train, evaluation, and prediction.
    - utils.py: Utils file, contains helper methods, and model definition. The `model.py` use this.
    - output/: Contain reports of ran sessions and saved model states. Saved models will be found here
      - saved_model_states/: Saved autogluon models are stored here
      - reports/: Manually generated reports file containing training/evaluation run information. Good for reproducibility. The reports are generated by `generate_report.py` and the new report row is saved in `train_report.csv` or `eval_report.csv`
  - pytorch/: directory containing model implementation for pytorch model. Handles training, evaluation, and predicting motions.
    - framework.py: Framework file, contains the implementation for train, evaluation, and prediction.
    - utils.py: Utils file, contains helper methods, and model definition. The `model.py` use this.
    - output/: Contain reports of ran sessions and saved model states. Saved models will be found here
      - saved_model_states/: Saved autogluon models are stored here
      - reports/: Manually generated reports file containing training/evaluation run information. Good for reproducibility. The reports are generated by `generate_report.py` and the new report row is saved in `train_report.csv` or `eval_report.csv`
      - optuna_studies/: Specific for pytorch. Optuna studies are stored at this folder. See optuna section for more information about Optuna.
  - generate_report.py: Contains methods to generate report files, used after a training or evaluation session has been done for any selected model. These generated reports contain information for reproducing runs and how a model performed against metrics. A new report is generated every time we run train or eval on a model and every report is saved in a single CSV file where every row is a report. The files are named: `train_report.csv` or `eval_report.csv`.
  - metrics.py: Contains all functions for calculating metrics for models for any selected model.
  - humanoid_config.py: it contains the humanoid configuration like the number of joints and the number of pose landmarks.

17.12 Send motion commands

The package simulation/random_motion_planner/ allows us to execute motions to any humanoid robot using the MoveIt2 framework. It is set up in config.sh where you set the variable below, which is the robot namespace you want to control.

```
replay_motion_namespace="humanoid_1"
```

This package has two features. You can run it to execute a series of random motions which is useful when you want to create synthetic data. The other feature is that when the package is running for a given humanoid robot, you can send motions you want to execute directly. When the package runs it subscribes to a topic: `humanoid_X/execute_motion` which inputs a stringified message of a dict (joint_name, value).

The easiest way to use it is in python using a dictionary, stringifying it with json, and then publishing it to the topic:

```
motion_dict = {JOINT_DATA}
json_str = json.dumps(motion_dict)
cmd = [
    "ros2",
    "topic",
    "pub",
    "--once",
    f"{humanoid_namespace}/execute_motion",
    "std_msgs/msg/String",
    f"{{data: '{json_str}'}}",
]
subprocess.run(cmd, check=True)
```

17.13 Model structure

`model.py` is responsible for routing the training and prediction to the right model. When running `model.py train modelX=pytorch_model cams=[500,501]` from `./control.sh`, the pseudocode below in `model.py` loads the right model and passes the needed information for training and prediction to the model.

```
import modelX
modelX.train(pose_NP, motion_NP, cams)
motion = modelX.train(pose_NP, cams)
pred= modelX.train(pose_NP, cams)
```

modelX can now have all information that it needs for training or prediction. Each model has to define at least the following interfaces:

```
modelX.py:

def modelX.train():
    FL_POSE  = FLATTEN_POSE_BY_SCENARIO(pose_NP, cams)
    // 1, [LDATA_a], [LDATA_b]
    x = join(FL_POSE, motion_NP)
    // [LDATA_a], [LDATA_b], [JDATA_a]
    train with the data above
    SAVE_MODEL_AS(str(cams))

def modelX.predict(pose_NP, cams):
    FL_POSE  = FLATTEN_POSE_BY_SCENARIO(pose_NP, cams)
    LOAD_MODEL(str(cams), FL_POSE)
    WHATEVER
    return motion_np
```

17.14 The Theory

- `I` : an image. (`I_s`: from simulator, `I_r` from real world)
- `P` : a pose (mediapipe output)
- `M` : a motion (moveit2)

then

- `SIM(M) -> I` : Simulator(gazebo) using inverse kinematics(moveit2) to convert the motion `M` to create image `I`
- `PE(I) -> P` : Pose estimator(mediapipe) takes the image `I`, to find human pose `P`
- `Q(P) -> M` : machine learning model `Q`, takes pose `P` and tries to replicate that pose estimator(mediapipe) pose using motion (moveit) `M`

Assumption: Pose estimator(mediapipe) performs good enough that it can detect human poses from both simulator and real-world domains:

- `PE(I_r) -> P_r`
- `PE(I_s) -> P_s`

Our goal is to

- pass random M to build the dataset of pairs : `<M,PE(SIM(M))> = <M,P>`
- Use the dataset above to find `Q()` which is the inverse of this `PE(SIM())`

Now we can do motion capture (replicate real human movements) by `Q(PE(I_r))`

17.15 Autogluon Model configuration

This section describes how AutoGluon is configured for a multi-label regression problem in our current setup.

To use AutoGluon for a multi-label regression problem, first we need to create a MultilabelPredictor by setting

- **labels**: the labels that we want to predict.
- **problem_types**: the problem type for each TabularPredictor.
- **path**: path to the directory where models and intermediate outputs should be saved.
- **consider_labels_correlation**: Whether the predictions of multiple labels should account for label correlations or predict each label independently of the others.

For **consider_labels_correlation**, we set it to FALSE in order to disable using one label as a feature for another. The reasons for this are:

- Each joint has its own degree of freedom and control, and it is independent from the rest of the joints.
- Training stability without error propagation.
- When set to False, the training will be faster.
- More general and simple model.

For training the model, we should take into consideration the following hyperparameters related to:

- Stacking: which is an ensemble technique where multiple models are trained and then a “meta-model” learns how to combine their predictions. If we use dynamic Stacking, it will automatically determine the optimal number of stacking layers and which models to include.
- Bagging: multiple training versions of the same model on different subsets of data and combining their predictions.
- preset: which condenses the complex hyperparameter setups. For example, we can use a small model size by using medium_quality, which will lead to faster training but with less prediction quality. Or we can use a large model by setting preset to best_quality, which will lead to better performance but much longer training time.

The current AutoGluon uses the following hyperparameters:

- **dynamic_stacking=False**: Disables the automatic stacking optimization.
- **num_stack_levels=0**: no stacking level.
- **auto_stack=False**: Disables automatic ensemble stacking.
- **num_bag_folds=0** and **num_bag_sets=1**, which means no bagging.

So here is the current training flow for each joint:

- Train NN_TORCH, GBM, and XGB models.
- No bagging: Each model trains on the full dataset once, no multiple training versions of the same model on different subsets of data and combining their predictions.
- No stacking: No meta-models combining predictions.
- Best model selection: Choose the best performing model per joint.

Eliminating dynamic stacking and multi-level stacking will lead to 50% to 70% time savings.

Overall, this configuration prioritizes training speed and stability over ensemble complexity, making it suitable for fast iteration and independent joint predictions.

17.16 Hyperparameter Tuning

In the current AutoGluon model, we are training NN_TORCH, GBM, and XGB models by using their default built-in hyperparameter settings. The next step for performance improvement is to add the **hyperparameter_tune_kwargs argument** to enable AutoGluon’s internal hyperparameter optimization. This will allow AutoGluon to automatically search for the best hyperparameters for each model type, balancing training time and prediction quality.

18 Moveit Panda robot

This package contains code for a robot called “Panda” which includes a description/URDFs (see `panda_description/`). Inside the `panda_moveit_config/` exist all code and config files that allow you to spawn a panda robot in Gazebo and be able to plan and execute motions. Inside `config/` there are several `*.yaml` files which act as configurations, and many `*.srdf, *.urdf, *.xacro` files. These are all related to the same panda robot. The main description file is `panda.urdf.xacro`.

To run our package, make sure that the Gazebo simulator is running. Then we can run the `panda_moveit_config/launch/demo.launc` which will do the following:

- Spawn a panda robot in Gazebo and publish its `robot_description`.
- Start RViz with a preset MoveIt config.
- Start the `move_group_node`, which handles all planning and executing of motions.
- Load all controllers for panda so that it can be controlled in Gazebo.

18.1 Moveit motion planner using Python API

The folder `custom_motion_planning_python_api` has scripting files that plan and execute motions for a panda robot. It uses the official MoveIt2 Python API to achieve this. In this package's `scripts/` folder exist these Python files. Look at the demo scripts for inspiration. To run any of the scripts, you run the launch file:

- `motion_planning_python_api_planning_scene.py` is an original demo from MoveIt2.
- `motion_planning_python_api_tutorial.py` is an original demo from MoveIt2.
- `demo_pick_and_place.py` is custom made.

The `moveit2_py` package is not currently available for Humble, but Jazzy supports it. The prerequisites to run a motion planner script are to run these in parallel terminals: `gazebo sim & panda_moveit_config/launch/demo.launch.py`. When these two are finished initializing, run the motion script with: `motion_planning_python_api_tutorial.launch.py`

19 Gazebo Worlds

```
<physics type="ode">
  <ode>
    <solver>
      <type>world</type>
    </solver>
    <constraints>
      <contact_max_correcting_vel>0.1</contact_max_correcting_vel>
      <contact_surface_layer>0.0001</contact_surface_layer>
    </constraints>
  </ode>
  <max_step_size>0.001</max_step_size>
</physics>
```

Two solvers: world step gives an accurate solution if it is able to solve the problem, while quick step depends on the number of iterations to reach an accurate enough solution.

19.0.1 contact/collision parameters

dampingFactor double Exponential velocity decay of the link velocity - takes the value and multiplies the previous link velocity by (1-dampingFactor). maxVel double maximum contact correction velocity truncation term. minDepth double minimum allowable depth before contact correction impulse is applied maxContacts int Maximum number of contacts allowed between two entities. This value overrides the max_contacts element defined in physics.

contact_max_correcting_vel : contact_max_correcting_vel This is the same parameter as the max_vel under collision->surface->contact. contact_max_correcting_vel sets max_vel globally. contact_surface_layer : contact_surface_layer This is the same parameter as the min_depth under collision->surface->contact.

Note: We had issue getting global settings (`contact_max_correcting_vel` and `contact_surface_layer`) working. We therefore define object level `minDepth` and `maxVel` for each object

"quick" solver parameters min_step_size The minimum time duration which advances with each time step of a variable time step solver. iters The number of iterations for the solver to run for each time step.

19.0.2 objects:

As of now there are many objects in the world file nad if you want to move them all it's quite cumbersome. Therefore the move_objects.py was added to be able to increment the poses by x amount if needed. See move_objects.py

19.0.3 reference:

- Physics Parameters
- SDF format

20 Converting meshes to SDF models

Here are the steps to convert FreeCAD meshes to SDF models (for Gazebo):

20.1 Script step-by-step

- Install FreeCAD `sudo apt-get install freecad=0.19.2+dfsg1-3ubuntu1` (Tested with version 0.19)
- Install Blender v3.3 LTS
- Download phobos.zip (tested with phobos 2.0 "Perilled Pangolin".)
- Install phobos by following the guide on their GitHub
- In your phobos settings (easiest accessed through the Blender GUI), change the models folder to where you want the output to be. You can add a relative path here so that the output folder depends on where you run the script, e.g., setting the model folder to `./phobos_out/` would create a `./phobos_out/` directory as a sub-directory to wherever you run the script and save the exported SDFs there.
- Enter FreeCAD and load your `.FCstd`, then select the body in the tree view and select `File->Export` and choose to export it as a Collada (`.dae`) file.
- Run `./build.sh`

You should now be able to load the model/directory created by the reformat script directly through Gazebo.

```
wget https://download.blender.org/release/Blender3.3/blender-3.3.0-linux-x64.tar.xz
tar -xf blender-3.3.0-linux-x64.tar.xz
mv blender-3.3.0-linux-x64 blender
alias blender="`pwd`/blender/blender"
blender --version
git clone git@github.com:dfki-ric/phobos.git
cd phobos
git checkout 2.0.0
blender -b --python install_requirements.py
cd ..
./build.sh
```

20.2 In GUI

- Import `.dae` file
- Set phobostype to `visual`
- Select the object and set the `geometry type`
- Guide says to `Object->Apply->Rotation & Scale` here, but I'm not sure what it actually accomplishes
  - This will apply the scaling of objects to their mesh information (vertices) and applies the current orientation to the mesh as well. After this option, all objects have zero orientation. If this is not desired, one can also only apply the `Scale`.
- Create a "visual" collection and put the model into there; this lets us create a tree structure. (unsure of necessity when we only have one model)
- Select the object and press `Create Link(s)`, make sure `selected objects` mode is chosen.
- Select the visual and then `Create Collision`.
- Select the object and then `Create Inertials`.
  - Here we need to manually enter our mass; we should check if that can be adjusted somehow.
- Save and export.
- Then we need to move around the resulting files so they're structured in the way Gazebo wants them, i.e.:

```
model
  meshes
     filename.dae
  model.config
  model.sdf
```

In order to accomplish this, we'll have to create the model.config file; however, this is relatively simple as it doesn't contain any unique data between different models aside from the model name.

Example of a successful build:

```
simulation/raw_models$ ./build_models.sh
```

```
../simlan_gazebo_environment/models/aruco/materials/textures/0.png
../simlan_gazebo_environment/models/aruco/materials/textures/1.png
../simlan_gazebo_environment/models/aruco/materials/textures/2.png
../simlan_gazebo_environment/models/aruco/materials/textures/3.png
```

```
Color management: using fallback mode for management
Color management: Error could not find role data role.
Blender 3.0.1
Read prefs: [HOME]/.config/blender/3.0/config/userpref.blend
Color management: scene view "Filmic" not found, setting default "Standard".
/usr/lib/python3/dist-packages/scipy/__init__.py:146: UserWarning: A NumPy version >=1.17.3 and <1.25.0 is req
  warnings.warn(f"A NumPy version >={np_minversion} and <{np_maxversion}"
Checking requirements:
ensurepip is disabled in Debian/Ubuntu for the system python.

Python modules for the system python are usually handled by dpkg and apt-get.

    apt install python3-<module name>

Install the python3-pip package to use pip itself.  Using pip together
with the system python might have unexpected results for any system-installed
module, so use it at your own risk, or make sure to only use it in virtual
environments.

WARNING: We couldn't do ensurepip; we try to continue anyway
  Checking yaml
  Checking numpy
  Checking scipy
  Checking collada
  Checking pydot
  Checking lxml
  Checking networkx
  Checking trimesh
  Checking PIL
Importing phobos
IMPORT:  phobos.blender.defs
Parsing definitions from: [HOME]/.config/blender/3.0/scripts/addons/phobos/data/blender/definitions
  defaultControllers.yml
  defaultSensors.yml
  defaultMaterials.yml
  defaultJoints.yml
  defaultSubmechanisms.yml
  defaultMotors.yml
Creating new definition type: joints
Creating new definition type: motors
IMPORT:  phobos.blender.display
IMPORT:  phobos.blender.io
RELOAD:  phobos.blender.model
IMPORT:  phobos.blender.operators
RELOAD:  phobos.blender.phobosgui
RELOAD:  phobos.blender.phoboslog
RELOAD:  phobos.blender.phobossystem
RELOAD:  phobos.blender.reserved_keys
RELOAD:  phobos.blender.utils
Registering operators.selection...
Registering operators.io...
Registering operators.editing...
TypeError: EnumProperty(..., default='mechanism'): not found in enum members
```

```
ValueError: bpy_struct "PHOBOS_OT_define_submodel" registration error: 'submodeltype' EnumProperty could not r
Registering operators.naming...
Registering operators.misc...

Registering phobosgui...
  ... successful.
+-- Collada Import parameters------
| input file      : ./objects/meshes/eur-pallet.dae
| use units       : no
| autoconnect     : yes
+-- Armature Import parameters ----
| find bone chains: yes
| min chain len   : 0
| fix orientation : yes
| keep bind info  : no
| IOR of negative value is not allowed for materials (using Blender default value instead)+-- Import Scene ---
| NODE  id='node0', name='node0'
+----------------------------------
| Collada Import : OK
+----------------------------------
Error in sys.excepthook:

Original exception was:
File "[HOME]/.config/blender/3.0/scripts/addons/phobos/blender/operators/editing.py", line 1048, in toggleVisu
Error in sys.excepthook:

Original exception was:
File "[HOME]/.config/blender/3.0/scripts/addons/phobos/blender/operators/editing.py", line 1051, in toggleColl
[20231211_08:52:53] WARNING No text file README.md found. (phobos/blender/utils/blender.py - readTextFile (125
Collada export to: [PATH]simulation/raw_models/phobos_out/unnamed/meshes/dae/Body.dae
Info: Exported 1 Object
Info: Exported 1 Object
Collada export to: [PATH]simulation/raw_models/phobos_out/unnamed/meshes/dae/Body.dae
Info: Exported 1 Object
Info: Exported 1 Object
Info: Phobos exported to: phobos_out/unnamed
Info: Export successful.
Info: Phobos exported to: phobos_out/unnamed
Info: Export successful.

----------------------------------------------------------------------------------------------------
Unregistering Phobos...
Unregistering phobosgui...
Unregistering display...
Unregistering icons...
Unregistering classes...
Unregistering manuals...
  ... successful.

Blender quit
Error: Not freed memory blocks: 14, total unfreed memory 0.002426 MB
```

20.3 Blueprint

This document contains explanations and motivations for the measurements as well as names for variables that should be used when designing the objects. The categories are mostly self-explanatory; the one that might need more of an explanation is the color red. This refers to loose objects such as pallets, barrels, boxes, and things that will be placed directly on the ground.

Measurements:

Category	Part / Object	Variable Name	Value [unit]
Shelf	Load Beam	beam_length	2.7 [m]
Shelf	Load Beam	beam_depth	0.05 [m]
Shelf	Load Beam	beam_height	0.14 [m]
Shelf	Load Beam	beam_weight	15.2 [kg]
Shelf	Footplate	footplate_length	0.111 [m]
Shelf	Footplate	footplate_width	0.1 [m]
Shelf	Footplate	footplate_height	0.004 [m]
Shelf	Stand / Pole	stand_length	0.07 [m]
Shelf	Stand / Pole	stand_width	0.08 [m]
Shelf	Stand / Pole	stand_height	3 [m]
Shelf	Stand / Pole	stand_weight	8.5 [kg]
Shelf	Cross Brace	brace_length	0.96 [m]
Shelf	Cross Brace	brace_width	0.01 [m]
Shelf	Cross Brace	brace_height	0.02 [m]
Shelf	Cross Brace	brace_weight	1 [kg]
Loose Object	EUR-Pallet	EUR_pallet_length	1.2 [m]
Loose Object	EUR-Pallet	EUR_pallet_width	0.8 [m]
Loose Object	EUR-Pallet	EUR_pallet_height	0.144 [m]
Loose Object	EUR-Pallet	EUR_pallet_weight	25 [kg]
Loose Object	Steel Drum	steel_drum_radius	0.3 [m]
Loose Object	Steel Drum	steel_drum_height	0.9 [m]
Loose Object	Steel Drum	full_steel_drum_weight	188 [kg]
Loose Object	Steel Drum	empty_steel_drum_weight	15 [kg]
Loose Object	Traffic Cone	traffic_cone_radius	0.22 [m]
Loose Object	Traffic Cone	traffic_cone_angle	-10 [degrees]
Loose Object	Traffic Cone	traffic_cone_total_height	1 [m]
Loose Object	Traffic Cone	traffic_cone_base_length	0.52 [m]
Loose Object	Traffic Cone	traffic_cone_base_width	0.52 [m]
Loose Object	Traffic Cone	traffic_cone_base_height	0.03 [m]
Loose Object	Traffic Cone	traffic_cone_base_angle	-20 [degrees]
Loose Object	Traffic Cone	traffic_cone_weight	6.5 [kg]
Loose Object	Box	box_length	Varies [m]
Loose Object	Box	box_width	Varies [m]
Loose Object	Box	box_height	Varies [m]
Loose Object	Box	box_weight	Varies [m]
Warehouse	Support Pole	pole_length	0.3 [m]
Warehouse	Support Pole	pole_width	0.3 [m]
Warehouse	Support Pole	pole_hole_side_length	0.27 [m]
Warehouse	Support Pole	pole_hole_side_width	0.13 [m]
Warehouse	Support Pole	pole_height	6 [m]
Warehouse	Support Pole	pole_weight	Undefined [kg]

EUR-pallet
EUR-pallet Measurements

20.4 AMR

AMR

It is defined in the XY plane, and the center of the AMR is on x,y=0,0, and the bottom of the robot is on z=0.

Define **scale**, **pos**, and collision in these files:

We write our AMR description in `simlan_gazebo_environment/urdf` and use `xacro` to create a `urdf` file with gazebo tags (so not a pure urdf file) that can be used both by `state_publisher` and `gazebo` (unlike 2 separate files: `model.sdf` that is used for Gazebo and `turtlebot.urf` that is used for state_publisher in the original Turtlebot_simulation git project).

There are two actual diffdrive wheels that are named `left` and `right` and four supporting wheels to balance the robot that are named `front_left`, `front_right`, `back_left`, and `back_right` (they have no friction and can be moved to different positions) with the radius of 0.98 of the main wheels.

Note: The reason for the shake is the difference between the radius of the real wheels and caster wheels.

Orientation:

- x: forward
- y: left
- z: up

20.5 Objects Modeled in FreeCAD

All the objects are located in `objects`. Below follows a list of the objects currently available, models created in FreeCAD.

- EUR-pallet
- Shelf
- Modular Shelf - a stackable shelf part that can create shelves of varying sizes
- Steel Drum
- Traffic Cone
- Support Pole
- Boxes with measurements in mm [Box-Length x Width x Height]
  - Box-160x130x70
  - Box-185x185x75
  - Box-185x185x185
  - Box-210x180x130
  - Box-230x160x85
  - Box-250x195x160
  - Box-430x250x260
  - Box-440x320x175
  - Box-570x380x380
  - Box-1185x785x1010
  - Box-600x800x400 (2 per level on EUR-Pallet)
  - Box-600x400x400 (4 per level on EUR-Pallet)
  - Box-300x400x400 (8 per level on EUR-pallet)

20.6 AMR camera

Camera

Inspired by CCTV Camera 3G-SDI

In the model: camera base height (70mm) + camera lens height (50mm) = 120mm

20.7 Resources

- Pallet Rack Specification & Configuration Guide
- Pallet Rack Estimator
- Pallet rack configurator
- pallet
- Shelf typical measurements: Shelf Measurements
- Forklift height which influences Shelf height: Forklift Height
- EUR-pallet: EUR-pallet
- Shelf design: Shelf design
- Shelf exact measurements: Shelf exact measurement
- Examples of boxes: Boxes Examples
- Measurement for steel drum: Steel Drum
- Full Steel drum weight is based on the oil density of 825kg/m^3 and the drum fits 210 liters.
- Some values helping in constructing the traffic cone: Traffic Cone

20.8 Blueprint

This document contains explanations and motivations for the measurements, as well as names for variables that should be used when designing the objects. The categories are mostly self-explanatory; the one that might need more of an explanation is the color red. This refers to loose objects such as pallets, barrels, boxes, and things that will be placed directly on the ground.

Measurements:

Category	Part / Object	Variable Name	Value [unit]
Warehouse	Walls	warehouse_length	23.2 [m]
Warehouse	Walls	warehouse_width	18 [m]
Warehouse	Walls	warehouse_height	6 [m]
Warehouse	Aisle	aisle_width	3.7 [m]
Warehouse	Door	door_height	4 [m]
Warehouse	Door	door_width	3 [m]
Warehouse	Warehouse	door_distance_to_wall	2 [m]
Warehouse	Shelf	shelf_distance_side_side	0 [m]
Warehouse	Shelf	single_shelf_wall_distance	0.1 [m]
Warehouse	Shelf	double_shelf_depth_distance	0.2 [m]
Warehouse	Shelf	shelf_total_length	2.88 [m]
Warehouse	Shelf	shelf_total_depth	1.1 [m]
Warehouse	Shelf	shelf_total_height	3 [m]

blueprint
The storage blueprint

20.9 Install instruction for BIM and Arch workbench

The warehouse was created in FreeCAD's BIM Workbench. This workbench isn't available by default but can easily be acquired by:

- Installing `pip3 install gitpython`
- Activating `Tools` -> `Addon Manager` -> `BIM` -> `Install/update Selected`

Using the measurements found in the blueprint in the documentation folder, I drew four lines and turned them into walls (line tool and wall tool respectively). By default, the wall will be created around the line, so that the line is in the middle of the wall. This can be changed in the wall properties so that it ends up entirely on one side of the line. As the blueprint lacked walls, I used the dimensions specified there as the inner measurements, meaning that the origin point is located at the inner bottom left corner of the wall. All the inner space is in positive x and y coordinates, while two of the walls are in negative coordinate space. I used the Aligned Dimension tool from the Annotation tools to confirm that the inner measurements matched those of the blueprint.

While the BIM Workbench has support for door objects, they tend to act as solids when imported into Gazebo, so the door is just a hole in the wall. This hole was created by adding a cube object (`3D/BIM -> Cube`) and having it intersect the wall where the door should be located. Then you select the cube and the wall (in that order) in the tree view and press `Modify -> Remove Component`. This should remove the cube object and create a hole where the cube and wall overlapped. The hole didn't appear in the right place, but it was possible to edit the position of the hole in its properties. I measured the location of the hole using Aligned Dimension to make sure it ended up in the right spot.

Going to the top view, I created a new rectangle object covering the whole warehouse. Then I turned it into a slab with the slab tool to create a floor for the warehouse.

Everything was added to a level object (note: by default, this level object is named "Floor," but don't confuse it with the slab that constitutes the physical floor) so that it's grouped together. If we set the wall heights to 0, they will automatically inherit the height of the Level object, so we can change all the walls easily by changing the height of the level. Textures for the floor and walls can later be added in Blender.

Finally, I exported the project as a Collada file (.dae) and added it into a simple .world file to see if it loaded properly in Gazebo, and the results seemed correct.

Since the warehouse will be static, we shouldn't need to define any additional parameters like mass or inertia; visuals and collision should be enough. Textures might need some improvement, as currently they're just basic colors, but it should be possible to add those in FreeCAD and have them included in the .dae file so we can load them visually in Gazebo later.

I also added windows for slightly better visibility.

20.10 Add warehouse to world-file

The floor of the warehouse goes below z=0, so the ground plane was lowered by 0.2 so that the warehouse still rests on top of it. As our simulations will mainly take place inside the warehouse, the warehouse floor replaces the ground plane at z=0. The warehouse itself was placed in the models directory and loaded into the world with an `<include>` tag. Additionally, the maze in the stage4 world was moved away from the origin so that it is fully contained inside the warehouse, though in the future it should be removed altogether.

20.11 Resources

Warehouse Aisle width: Aisle Width

21 Object Mover package

Dyno-robotics/Infotiv delivery

Setting `self.MODE = "abnormal/normal"` in the main script causes the object movement to deviate from the normal distribution specified above.

To reproduce

```
git checkout ....
# In three separate terminals:
./control.sh clean ; ./control.sh build ; ./control.sh sim
./control.sh move_object
./control.sh camera_dump
```

21.1 Random, In Distribution Objects Movement (normal mode)

Data collection from cameras according to requirements

- Random self.objects are placed within the position range self.grid_x, self.grid_y: REQ.ID.1
- Random rotation along the z axis.
- You can use `camera_config ID.xacro` for placement of several cameras in the intersection. You can alternatively use `camera_config ID_noise.xacro` to slightly changes the camera settings (REQ.ID.2) as below:
  - at most +-10 degree (+-0.1) rotation around one of the axis
  - at most +-10cm (+-0.1) change in x,y,z coordinates

21.2 Random, Out of Distribution Objects Movement (abnormal mode)

Data collection from cameras according to requirements.

- Addition of other objects (spotlight, support pole, traffic cone): REQ.OD.1, REQ.OD.2.
- With some probability, the objects don't leave the scene and may coexist and collide with new objects: REQ.OD.3.
- Objects are at least rotated by /4 (45 degrees, upside-down object): REQ.OD.3, REQ.OD.4.
- The spotlight is tilted by /6 (30 degrees): REQ.OD.1.
- Objects are not placed on the ground and instead are dropped from a height of 2-3 meters: REQ.OD.4.

21.2.1 Deterministic, Disentanglement One-parameter Object Movements (one_object_deterministic mode)

- Object: forklift
- Camera: 1
- (#Parameter: orientation of forklift)
- Annotation: Position and rotation of the forklift. Annotation in filename.
- 256x256 RGB non compression

./control.sh sim ./control.sh move_object ./control.sh camera_dump

21.3 Licenses and Credits

The majority of code for the pallet truck comes from these turtlebot3 and turtlebot3_simulations repositories (in `humble-devel` branch) with *Apache-2.0 license* and we continue using the same license: The Turtlebot3 robot project belongs to robotis.com and accessible in git repository. The authors and maintainers of the original packages that all credits go to are:

- Darby Lim
- Pyo
- Will Son
- Ryan Shim

In October 2023, Hamid Ebadi renamed the package owner information and name for turtlebot3 projects to avoid dependency issues any naming confusion with the original packages and created independent packages for activities within the research project. We also got inspired and used the skeleton code from these open source project and courses:

- https://github.com/ros-controls/gazebo_ros2_control
- https://github.com/renan028/forklift_robot
- https://github.com/ROBOTIS-GIT/
- http://turtlebot3.robotis.com
- Articulated Robotics
- "ROS2 for Beginners Level 2 - TF | URDF | RViz | Gazebo" Udemy course
- "ROS2 Nav2 [Navigation 2 Stack] - with SLAM and Navigation" Udemy course
- Visual Servoing in Gazebo grobot
- The modification of Human-Gazebo in /src/humanoid_robot/model/human-gazebo is licensed under LGPL-2.1.
- Moveit2 is licensed under BSD-3-Clause license.

21.4 Resources

- Gazebo official video playlist
- Getting Ready to Build Robots with ROS
- Udemy course on ROS2 and Gazebo
- classcentral course
- theconstructsim course
- Articulated Robotics
- ROS URDF
- FoxGlove studio (rviz alternative)
- ROS2 Tutorials
- aws-robomaker
- model for factory
- Tugbot in Warehouse
- logistics in warehouse
- aws-robomaker
- gazebo_worlds
- warehouse
- forklift
- Docker install guide
- Hazard stripes taken from Tugbot warehouse
- Deadlock image

Other solutions:

- Kollmorgen: How does an AGV navigate?
- SwissLog CarryPick
- Toyota forklifts
- ILIAD Project
- GoPal
- navigation_oru navigation stack by Örebro University
- https://alignproductionsystems.com/equipment-category/navigation/#

Specification of items:

- SLAM Navigation Compact Pallet Mover Nature Navigation Mini Forklift with Payload 1000KG
- Driverless Lifting System: Single & Double Scissor Lift | AGILOX

- Wholesale Pallet Agv Trucks Jack Automated Autonomous Forklift
- Volvo modular containers
- Volvo Emballage Specifications Volvo Group Packaging System
- Freecad
- Blender
- example worlds
- SDF format
- SDF tutorial
- FreeCAD RobotCreator Workbench

Coordinates:

- odom

Camera projection:

- https://github.com/polygon-software/python-visual-odometry/blob/master/Chapter%203%20-%20Camera%20Projection.ipyn
- https://classic.gazebosim.org/tutorials?tut=camera_distortion
- https://learnopencv.com/rotation-matrix-to-euler-angles/
- https://www.geeksforgeeks.org/calibratecamera-opencv-in-python/
- https://docs.opencv.org/4.x/dc/dbb/tutorial_py_calibration.html
- http://sdformat.org/tutorials?tut=specify_pose

Jackal Robot: The Clearpath Jackal Robot code is forked by Dyno Robotics from Clearpaths jackal Github. Branch foxy-devel is ported to dyno_humble, where changes for namespacing and a bringup package is added.

- To improve collaboration in development environment we use vscode and docker as explained in this instruction using these docker files. For production environment follow installation procedure used in .devcontainer/Dockerfile to install dependencies.
- The Docker setup is added by Christoffer Johanesson (Dyno Robotics), based on Dyno experience working with Docker.

Panda Arm: - https://github.com/cpezzato/panda_simulation - https://www.researchgate.net/figure/Panda-robot-with-labeled-joints_fig1_361659008 - https://www.kuka.com/it-it/settori/banca-dati-di-soluzioni/2025/05/5-kuka-robots-optimize-pet-food-handling-in-74-sqm - https://arxiv.org/pdf/2506.12089

AI/ML: - https://www.calibraint.com/blog/beginners-guide-to-diffusion-models

Humanoid: - https://gazebosim.org/docs/latest/actors/ - https://research.google/blog/on-device-real-time-body-pose-tracking-with-mediapipe-blazepose/

21.5 Licenses

Imu_tools is the one with several licenses, using BSD-3, GPLv3, GNU v3, but as we don't change the code so we believe there is no license conflict with the current project license. The dependency wireless (https://github.com/clearpathrobotics/wireless.git) from Clearpath robotics is the without a separate license file. There are license names within files, referencing BSD so we believe there is no license conflict with the current project license.

21.6 Docker and VS Code setup

The Docker setup added by Christoffer Johanesson (Dyno Robotics), based on Dyno experience working with Docker.

21.7 Project maintainer

This project is currently maintained by Hamid Ebadi.

22 Changelog

22.0.1 5.0.0: Refactor MoCap pipeline and sensors (Dec 2025) - Team/Tech Lead: Hamid Ebadi

- Working demo video scenario: Hamid Ebadi, Pär Aronsson
- Refactor and improve prediction pipeline: Pär Aronsson
- Review and update the mocap architecture: Pär Aronsson, Hamid Ebadi
- Pipeline for Prediction on real data, images, and videos: Pär Aronsson
- Improve motion player: Pär Aronsson
- New demo scenario: Pär Aronsson

- Unify “single” and “multi” pipeline: Pär Aronsson, Hamid Ebadi
- Documentation and presentation (mkdocs, pandas): Hamid Ebadi
-

22.0.2 4.0.0: Improvements (Nov 2025) - Team/Tech Lead: Hamid Ebadi

- System integration tests: Pär Aronsson
- Overall system diagrams (launch files and bringups): Anton Stigemyr Hill
- PyTorch model for humanoid MoCap: Pär Aronsson
- Improved documentation and AutoGluon configuration: Marwa Naili
- Demo scenario: Anton Stigemyr Hill
- Semantic segmentation, depth, color camera sensors: Anton Stigemyr Hill, Pär Aronsson
- Humanoid navigation: Anton Stigemyr Hill
- Rework GPSS/ArUco navigation: Anton Stigemyr Hill

22.0.3 3.0.0: Jazzy, humanoid (Oct 2025) - Team/Tech Lead: Hamid Ebadi

- Collision sensor: Anton Stigemyr Hill
- Automatically generated parameter-files for robot_agents and world fidelity: Anton Stigemyr Hill
- Shared map and obstacle generation for robot_agents: Anton Stigemyr Hill
- Geofencing and safety stop: David Espedalen
- Humanoid motion capture: Casparsson and Siyu Yi, Hamid Ebadi
- Humanoid integration (pymoveit2 to moveit_py): Siyu Yi, Pär Aronsson
- Mocap (humanoid) with multi-camera training pipeline: Siyu Yi, Pär Aronsson
- Panda arm integration: Pär Aronsson
- Dyno-robotics trajectory replay integration, scenario: Sebastian Olsson

22.0.4 2.1.0: Namespace cleanup and multi-agent navigation (Aug 2025) - Team/Tech Lead: Hamid Ebadi

- Implemented generic robot-agent control supporting multiple meshes (e.g., pallet truck, forklift) and distinct ArUco IDs: Pär Aronsson
- Robot-agent and navigation2 (nav2) namespace: Pär Aronsson
- Multi-agent navigation capabilities using GPSS: Pär Aronsson

22.0.5 2.0.0: GPSS (May 2025) - Team/Tech Lead: Hamid Ebadi

- Upgrade Gazebo from classic to ignition, adapting new sensors: Nazeeh Alhosary
- ArUco_localization, added localization and nav2 to new robot: Pär Aronsson
- Navigation: Pär Aronsson
- Bird’s Eye View ros package: Converting projection.ipynb to camera_bird_eye_view: Hamid Ebadi, Pär Aronsson
- Deprecated: InfoBot, replaced with Pallet_truck (based on Jackal by Clearpath Robotics)

22.0.6 1.0.6: Collision (Feb 2025)

- Add scenario execution library for ros2
- Add action servers for set_speed, teleport_robot and collision
- Add Node for TTC calculation
- Add package for custom messages

22.0.7 1.0.5: Delivery 4 (Dec 2024)

- D3.5 Disentanglement: One-parameter Object Movements.
- Trajectory visualization: Hamid Ebadi
- feat: added DynoWaitFor to make sure Jackal is launched last (synchronisation issue)
- simlan_bringup pkg created: Christoffer Johannesson

22.0.8 1.0.4: Delivery 3 (Oct 2024)

- SMILE-IV and ArtWork projects contributions
- Updated camera extrinsic, fixing OpenCV camera calibration to Gazebo simulator conversion: Erik Brorsson, Volvo
- D3.4 Out distribution data collection: Hamid Ebadi
- D3.3 In distribution data collection: Hamid Ebadi

- D3.2 Images for stitching: Hamid Ebadi
- D3.1 Dataset draft for HH: Hamid Ebadi
- CI/CD and Kubernetes integration by Filip Melberg, Vasiliki Kostara
- Jackal integration: Christoffer Johannesson, Hjalmar Ruscck
- Docker/vscode
- Disable GPU support by default
- POC for camera projection to pixel coordinates (Jupyter notebook): Hamid Ebadi
- Camera image dump and image stitching: Hamid Ebadi

22.0.9 1.0.3: Jackal Robot (Mar 2024) - Christoffer Johannesson

- Added dyno fork of jackal repo from Clearpath Robotics.
- Updated to Humble, added bringup and support for namespacing. Jackal can be spawned in Gazebo and controlled through the keyboard.
- Added .devcontainer folder with Dockerfile and devcontainer.json to set up project container in VS Code.
- Added docker-compose to link all needed files and set environment variables.
- Added .vscode folder with settings and tasks for easy building of the project.
- Updated README with info on how to use Docker setup in VS Code, and some features to make it easy to share the same setup with others.
- Features include: python3 dependency install with pip, cloning of other git repositories and how to make changes to those repositories.

22.0.10 1.0.2: Delivery 2 (Feb 2024)

- Volvo warehouse 0.0.1: Hamid Ebadi
- Volvo camera calibration in Gazebo 0.0.1: Hamid Ebadi
- Integrate Infobot_agent 0.0.2: InfoBot differential-drive AMR (Autonomous Mobile Robot) URDF and ROS launcher (GOPAL and forklift): Hamid Ebadi
- Integrate Infobot_cartographer 2.1.5: cartographer for creating PGM maps
- Integrate nav2_commander 0.0.2: ROS package to command Infobot where the destination is: Hamid Ebadi
- Integrate Infobot_navigation2 2.1.5: Standard Nav2 stack launcher: Hamid Ebadi
- Integrate Infobot_teleop 0.0.2: Teleoperation for InfoBot

22.0.11 1.0.1: Delivery 1 (Dec 2023) - Team/Tech Lead: Hamid Ebadi

- Basic warehouse model 1.0.0: Anders Bäckelie
- CAD modelling (eur-pallet, boxes, shelf, support_pole, traffic-cone, steel_drum) 1.0.0: Jacob Rohdin
- Physics (collision, inertia), visuals and Gazebo compatible mesh creation 1.0.0: Anders Bäckelie
- Walking actor using scripted trajectories 1.0.0: Anders Bäckelie
- Infobot_Gazebo_environment 1.0.0: ROS2 launcher to start Gazebo world: Hamid Ebadi
- static_agent_launcher 1.0.0: Camera and ArUco tags: Hamid Ebadi
- camera-viewer 1.0.0: Python code to get Gazebo camera feed: Hamid Ebadi

22.0.12 high-level_diagram_SIMLAN.drawio

high-level_diagram_SIMLAN.drawio.png

22.0.13 Humanoid_mocap_flow.drawio

Humanoid_mocap_flow.drawio.png

22.0.14 humanoid_mocap_pipeline.drawio

humanoid_mocap_pipeline.drawio.png

22.0.15 launch_bringup.drawio

launch_bringup.drawio.png

22.0.16 SIMLAN_DIAGRAM.drawio

SIMLAN_DIAGRAM.drawio.png