Lab03: Software In The Loop (SITL) & Auto Modes

Table Of Contents:

1. Introduction to Software In The Loop (SITL)

1.1 What is SITL?

Software In The Loop (SITL) is a **simulation environment** that **allows ArduPilot firmware to run on a desktop computer instead of physical hardware**. The autopilot code executes in a **simulated environment** where sensor inputs, actuator outputs, and environmental conditions are mathematically modeled. This creates a **virtual vehicle** that behaves identically to real hardware from the firmware's perspective.

SITL operates by creating a complete simulation loop where:

- **Sensor data** (GPS, IMU, compass, etc.) is generated mathematically based on the vehicle's simulated state
- **ArduPilot firmware** processes this data using the same algorithms as real hardware
- **Control outputs** are fed back into the simulation to update the vehicle's virtual position and orientation
- **MAVLink telemetry** is transmitted exactly as it would be from physical hardware

1.2 Hardware Context

In SmartMove Lab, we utilize several autopilot configurations:

- **Primary training units** (**Used in your KITs**): Pixhawk 2.4.8 PX4 32 Bits Flight Controller
- **Aerosea vessel**: CUAV X7+ PRO Flight Controller
- **C.U.A.K. vessel**: CUAV V5+ Autopilot

All units run **ArduPilot Rover firmware** (latest stable: version 4.6.0 as of June 2025) configured for maritime operations through the `FRAME_CLASS` parameter set to 2, which optimizes the control algorithms for waterborne vehicles.

1.3 Key Benefits of SITL

Development and Testing Efficiency SITL enables rapid development cycles without requiring physical hardware access. Algorithm modifications, parameter adjustments, and mission planning can be tested immediately without hardware setup time or environmental constraints.

Risk-Free Experimentation Complex autonomous behaviors, aggressive maneuvers, and failure scenarios can be tested safely without risk to expensive hardware or operational safety. This is particularly valuable for maritime applications where vehicle recovery can be challenging.

Reproducible Testing Environment Unlike real-world conditions, SITL provides completely controllable and repeatable test scenarios. Weather conditions, sensor noise levels, and environmental factors can be precisely controlled or systematically varied.

Mission Development and Validation Waypoint missions, autonomous behaviors, and complex navigation scenarios can be developed and validated before deployment to physical hardware, significantly reducing sea trial time and costs.

1.4 SITL vs. Alternative Testing Approaches

Hardware In The Loop (HITL) HITL combines real autopilot hardware with simulated sensors and actuators. While providing more accurate hardware timing and processor loading, HITL requires physical hardware and is more complex to configure than SITL.

Real Hardware Testing Physical testing provides the ultimate validation but is resource-intensive, weather-dependent, and carries operational risks. Real testing is essential for final validation but inefficient for initial development phases.

Alternative Simulation Platforms Other simulation environments (Gazebo, AirSim, X-Plane) offer enhanced visual rendering and physics modeling but typically require more complex setup procedures. ArduPilot's SITL provides an optimal balance between capability and accessibility for autopilot-focused development.

1.5 System Architecture Integration

SITL extends beyond basic autopilot simulation to enable testing of complete autonomous maritime systems. In operational deployments, vessels utilize **Raspberry Pi 5 companion computers** connected to autopilots, running Python scripts that:

- **Receive telemetry data** from the autopilot via MAVLink protocol
- **Send mission commands** and mode changes to the autopilot
- **Communicate with shore stations** and other vessels via MQTT protocol
- **Access internet connectivity** through onboard 4G dongles for real-time coordination

The power of SITL lies in its ability to simulate this entire architecture on a single development machine. Python scripts that would normally run on the Raspberry Pi companion computer can execute alongside the SITL simulation, creating a comprehensive testing environment for:

- **Multi-vessel coordination protocols** using MQTT messaging
- **Autonomous mission execution** and adaptation algorithms

- **Shore-to-vessel communication** scenarios including network interruptions
- **Data collection and telemetry streaming** workflows

This integrated simulation approach enables thorough validation of the complete system stack—from low-level autopilot behaviors to high-level fleet coordination—before expensive sea trials or hardware deployment.

1.6 Applications in Maritime Robotics

SITL proves particularly valuable for maritime applications where:

- **Sea trials are expensive** and weather-dependent
- **Vehicle recovery** can be challenging in open water
- **Environmental conditions** (currents, waves, wind) significantly affect vehicle behavior
- **Multi-vessel coordination** requires extensive testing before deployment
- **Communication protocols** between vessels and shore stations need validation
- **Companion computer integration** requires debugging without physical hardware risks

The simulation environment allows comprehensive testing of autonomous behaviors, collision avoidance algorithms, MQTT-based communication protocols, and multi-vessel coordination scenarios that would be impractical to test extensively with physical hardware.

1.7 Computer Vision and Object Recognition Integration

Autonomous maritime systems require **computer vision capabilities** for collision avoidance, vessel recognition, and situational awareness. In operational deployments, vessels utilize **Raspberry Pi Camera Module 3** with object detection models such as **YOLO** to identify and track other vessels, obstacles, and navigation markers.

SITL Simulation Limitations Basic SITL environments cannot simulate camera feeds or visual object recognition algorithms. The simulation focuses on autopilot firmware behavior and does not include computer vision processing or sensor data beyond standard navigation sensors (GPS, IMU, compass).

Practical Testing Approaches Despite these limitations, computer vision integration can be tested through modular approaches:

- **MQTT message simulation**: Python scripts can generate simulated object detection results (vessel positions, classifications, confidence levels) and transmit them via MQTT, allowing testing of decision-making algorithms based on detection data

- **Collision avoidance logic**: Autonomous behavior algorithms can be developed and validated using simulated detection inputs before integration with actual computer vision systems

- **Multi-vessel scenarios**: Fleet coordination behaviors can be tested by simulating detection of other vessels through MQTT messaging

Advanced Simulation Alternatives More sophisticated simulation environments such as **Gazebo** or **AirSim** can provide visual simulation capabilities including camera feeds and object recognition testing, though they require significantly more complex setup procedures.

Future Sensor Integration The modular system architecture supports expansion with additional sensors including **LIDAR** for enhanced obstacle detection and **AIS receivers** for tracking larger commercial vessels. These sensors follow similar integration patterns where sensor data processing can be developed separately and integrated through standardized communication protocols.

2. SITL Installation and Setup

2.1 System Requirements and Platform Considerations

SITL supports multiple operating systems with varying installation complexity:

Linux (Ubuntu/Debian): Native support with straightforward installation. Recommended for development work due to direct compatibility with ArduPilot build tools.

Windows 10/11: Requires Windows Subsystem for Linux 2 (WSL2) to run Ubuntu environment. Additional display server configuration needed for graphical applications.

macOS: Supported but requires Homebrew package manager and additional dependencies. Installation complexity varies by macOS version.

Minimum Requirements:

- 4GB RAM (8GB recommended)
- 20GB available disk space
- Intel/AMD x64 processor
- Internet connection for package downloads

2.2 Installation Process Overview

The SITL installation involves several key steps:

1. **Environment Setup**: Configure operating system and dependencies
2. **ArduPilot Repository**: Clone source code and select appropriate firmware version
3. **Build Environment**: Compile SITL binaries and configure tools
4. **Display Configuration**: Enable graphical output (Windows/WSL2 specific)
5. **Parameter Customization**: Configure simulation parameters for maritime operations

2.3 Windows Setup with WSL2

Full notes on this procedure:
https://ardupilot.org/dev/docs/building-setup-windows.html

TL;DR: In recent versions of Windows 11, the single command `wsl --install` mostly works as an adequate first initial step.

WSL2 Installation Windows users must install WSL2 (Windows Subsystem for Linux 2) to run the Ubuntu environment required for ArduPilot development.

```
# Enable WSL and Virtual Machine features (works on modern Windows 11 PCs)
# MUST be ran from a Powershell (or Windows Terminal) elevated (administrator) window

wsl --install
```

Note: **Only if non-admin**, run these first in an elevated windows terminal prompt (admin):

```
# Enable WSL and Virtual Machine features
dism.exe /online /enable-feature /featurename:Microsoft-Windows-Subsystem-Linux /all /norestart
dism.exe /online /enable-feature /featurename:VirtualMachinePlatform /all /norestart
```

Then **reboot your machine**, login as standard user, open windows terminal and issue the following command (will ask for admin permission):

```
wsl --install -d Ubuntu
```

Follow prompts and Ubuntu will be installed and usable by your 'regular' (non-admin) user.

Procedure: Verifying GUI Application Support (WSLg)

To confirm that graphical Linux applications can run on Windows 11 via the integrated WSLg feature. This method does not require a third-party X Server (e.g., VcXsrv) or manual `DISPLAY` variable configuration.

Steps:

Launch the WSL Terminal: Open the installed Linux distribution (e.g., "Ubuntu") from the Windows Start Menu.

Update Package Lists & Install Test Utilities: Run the following command to refresh the system's package index and install the `x11-apps` package, which contains simple graphical programs for testing.

```
sudo apt update && sudo apt install x11-apps
```

Note: Enter the UNIX password when prompted. Press `Y` to approve the installation.

Execute a Test Application: Launch a sample graphical program from the package.

```
xeyes
```

Verify the Outcome: The graphical window for the `xeyes` application will appear directly on the Windows desktop. This confirms that the WSLg feature is functioning correctly, and no further GUI configuration is necessary.

[Note: following step (VcXsrv) NOT NEEDED in Windows 11]

Display Server Configuration WSL2 requires an X server to display graphical applications. Install **VcXsrv Windows X Server**:

1. Download and install VcXsrv from official sources
2. Launch XLaunch with configuration:
    - **Multiple windows**
    - **Display number: -1**
    - **Start no client**
    - **Disable access control** (checked)

WSL2 Display Configuration Configure Ubuntu environment to use Windows X server:

```
# Add to ~/.bashrc
export DISPLAY=$(grep -m 1 nameserver /etc/resolv.conf | awk '{print $2}'):0
```

2.4 Ubuntu/Linux Environment Setup

Some notes on the install procedure:

The official guide mentions some '*Recommended GUI Tools*'. I don't use them, so I did not install them. You might install them if you want, but they are not necessary.

Also, in 'Clone ArduPilot repository' the instructions mention two choices, either cloning main ardupilot repo - this is what I recommend, because this is what we want to download and compile, or forking the repo for possibly making changes and submitting back. I don't think this is relevant right now. So use https://github.com/ArduPilot/ardupilot.git URL in that section.

System Dependencies Install required packages and (absolutely essential) development tools (*):

(*) - Rest will be handled by dedicated script later in the installation process.

```
# Update package manager
sudo apt update && sudo apt upgrade -y

# Install essential tool only (git)
sudo apt install git
```

ArduPilot Prerequisites The ArduPilot repository includes an automated script for installing development dependencies:

```
# Clone ArduPilot repository
git clone --recurse-submodules https://github.com/ArduPilot/ardupilot.git
cd ardupilot
```

2.5 Run prerequisite installation script

```
# Run prerequisite installation script
# From here: https://ardupilot.org/dev/docs/building-setup-linux.html#install-some-required-packages

# Note: This will take some time and stress the computer..
Tools/environment_install/install-prereqs-ubuntu.sh -y

# Reload environment variables
source ~/.profile
```

2.6 Use latest stable version of ArduPilot

By default, cloning the ArduPilot repository places you on the `master` branch (at least this is what happened in my case). This branch contains the latest, cutting-edge development code, which may be unstable. For production use or to follow specific tutorials, it is highly recommended to use an official, stable version tag.

The following steps will guide you through switching from the development branch to a specific stable release (e.g., `Rover-4.6.0`).

Step 1: Navigate to the ArduPilot Directory

First, ensure your terminal is operating inside the cloned repository folder.

```
cd ~/ardupilot
```

Step 2: Verify Your Current (Development) Version

Before making changes, let's confirm you are on the development branch. The `git describe` command will show your position relative to the last official release.

```
git describe --tags
```

The output will be a long string, indicating you are many commits ahead of the last tag. It will look something like this: `ArduPilot-4.6.0-beta1-3024-ge1d097a639`

Step 3: Switch to the Stable Version Tag

Now, check out the specific stable version tag. This command switches your project's code to the exact state of that release.

```
git checkout Rover-4.6.0
```

Note: After running this command, Git will report that you are in a "detached HEAD" state. This is normal and expected, as it simply means you are on a fixed release tag instead of a development branch.

Step 4: Align Submodules with the Stable Version

This is a critical step.
After switching the main repository, **you must update all the sub-component libraries to match the version they had at the time of the stable release**.

```
git submodule update --init --recursive
```

(Above command updated all external libraries/dependencies to the versions they had when Rover-4.6.0 was released)

Step 5: Verify the New (Stable) Version

Finally, run the `describe` command again to confirm you have successfully switched.

```
git describe --tags
```

The output should now be a simple, clean tag name associated with the release, confirming you are no longer on a development branch:

You should see something like: `APMrover2-stable`

Build Configuration Configure and compile SITL for Rover applications:

```
# Configure build environment for SITL
./waf configure --board sitl

# Build Rover firmware
./waf rover
```

The build process creates SITL executables and configures the development environment for simulation use.

2.7 Environment Customization

Maritime Parameter Configuration Create parameter files optimized for boat simulation:

```
# Create parameter directory
mkdir -p ~/custom-parms
cd ~/custom-parms

# Create boat parameter file
nano boat.parm
```

Parameter File Contents:

```
# Maritime vessel configuration
FRAME_CLASS 2
BATT_CAPACITY 30000
# Additional maritime-specific parameters can be added here

# CTRL-O to save, CTRL-X to leave
# (***) We 're using nano as editor, you can use whatever you want,
# or even remote login with VS Code perhaps as a more user-friendly
# option
```

Custom Simulation Locations Add specific geographic locations for simulation scenarios:

```
# Edit locations file
nano ~/ardupilot/Tools/autotest/locations.txt

# Add custom locations (append to file)
# Syros - Nisaki area
Syros=37.439322,24.945616,0,180

# Piraeus (port of athens):
Piraeus=37.9435,23.6472,0,90

# Add more here if needed
# Here, I add some more, each one 20m south and 20m west
# of previous one (we 'll use them later)

Syros2=37.439142,24.945389,0,180
Syros3=37.438962,24.945162,0,180
Syros4=37.438782,24.944935,0,180
```

Location format: `name=latitude,longitude,altitude,heading`

2.8 First SITL Run (attention: this will probably fail !)

```
# Launch SITL with environmental conditions
sim_vehicle.py -v Rover -L syros --add-param-file=/home/<username>/custom-parms/boat.parm
```

2.8.1 - CRITICAL - Fix OpenCV/NumPy Compatibility Issues (Critical for Map Display)

*** [As of time of writing - June 2025] ***

Problem: The ArduPilot prerequisites script installs OpenCV 4.11.0+ and NumPy 2.x, which have a known compatibility bug that prevents MAVProxy's map module from loading.

Symptoms:

- Console works fine, shows `MANUAL>` prompt
- Map fails to load with errors like:
  - `cv2.error: (-5:Bad argument) in function 'imdecode'`
  - `A module that was compiled using NumPy 1.x cannot be run in NumPy 2.3.0`
  - `AttributeError: _ARRAY_API not found`

Solution (required for map functionality):

(run below commands while `(venv-ardupilot)` is activated !!!

```
# Downgrade OpenCV to last stable version before the bug
pip uninstall opencv-python
pip install opencv-python==4.9.0.80

# Downgrade NumPy to 1.x for compatibility with OpenCV 4.9.0.80
pip install "numpy<2"
```

Verification: Run SITL with map:

```
sim_vehicle.py -v Rover -L Syros --console --map
```

Both console and map should now work without errors.

Note for Future:

This compatibility issue affects ArduPilot installations from mid-2024 onwards due to newer OpenCV/NumPy versions. The fix above uses the last known working combination of OpenCV 4.9.0.80 + NumPy 1.x. This should be resolved sometime in the future.

For context, some info was found here:
- https://github.com/opencv/opencv/issues/25918
- https://stackoverflow.com/questions/38190679/trying-to-read-numpy-array-into-opencv-cv2-imdecode-returns-empty-argument
- https://answers.opencv.org/question/17829/error-during-image-decoding-imdecode-using-python/

2.9 Comprehensive Environmental Factors Simulation - Cycladic Meltemi Winds

Environmental Simulation Parameters SITL supports comprehensive environmental simulation to test vehicle behavior under realistic maritime conditions. For the Cyclades region where the summer school takes place, the **Meltemi wind** presents particular challenges with strong northerly winds, significant waves, and surface currents.

Create an environmental parameter file to simulate these challenging conditions:

```
# Create Meltemi wind parameter file
nano ~/custom-parms/meltemi.parm
```

Meltemi Parameter File Contents:

```
# A strong, gusty wind from the north.
# Significant waves with occasional larger swells.
# A considerable surface current.
SIM_WIND_SPD 12       # Wind speed of 12 m/s (about 23 knots)
SIM_WIND_DIR 0        # Wind from the north (typical for Meltemi)
SIM_WIND_TURB 0.5     # Moderate turbulence
SIM_WAVE_ENABLE 1     # Enable wave simulation
```

```
SIM_WAVE_LENGTH 20    # 20-meter wavelength
SIM_WAVE_AMP 1.5      # 1.5-meter wave height (3-meter
peak-to-trough)
SIM_WAVE_DIR 0        # Waves coming from the north
SIM_WAVE_SPEED 6      # Wave speed of 6 m/s
SIM_DRIFT_SPEED 1     # Surface current of 1 m/s (about 2 knots)
SIM_DRIFT_TIME 5      # Drift changes direction every 5 seconds
```

Parameter Explanations:

- **Wind Simulation**: `SIM_WIND_SPD` and `SIM_WIND_DIR` create consistent northerly winds typical of Meltemi conditions
- **Wind Turbulence**: `SIM_WIND_TURB` adds realistic gustiness affecting vehicle stability
- **Wave Effects**: `SIM_WAVE_*` parameters simulate sea state impacts on vehicle dynamics
- **Surface Current**: `SIM_DRIFT_*` parameters model water movement affecting navigation accuracy

Loading Environmental Parameters:

```
# Launch SITL with environmental conditions
sim_vehicle.py -v Rover -L syros \
--add-param-file=/home/<username>/custom-parms/boat.parm \
--add-param-file=/home/<username>/custom-parms/meltemi.parm \
--console --map
```

Testing Applications: Environmental simulation enables realistic evaluation of:

- **LOITER mode performance** under strong current and wind conditions
- **AUTO mode navigation** accuracy in challenging weather
- **RTL mode reliability** when returning against environmental forces

- **Parameter tuning** for real-world deployment conditions

This environmental simulation closely replicates conditions that might be encountered during actual sea trials in the Cyclades, providing invaluable experience before hardware deployment.

2.9 Installation Verification

SITL Startup Test Verify successful installation by running a basic simulation:

```
# Create test directory
mkdir -p ~/sitl-test
cd ~/sitl-test

# Launch SITL with basic configuration
sim_vehicle.py -v Rover -L syros --console --map
```

Or you can run a more "complete" simulation by issuing the following:

```
sim_vehicle.py -v Rover -L Syros
--add-param-file=/home/thomas/custom-parms/boat.parm
--add-param-file=/home/thomas/custom-parms/meltemi.parm --console
--map
```

(*) *- Replace 'thomas' directory with your own wsl username*

Expected Output Verification:

- MAVProxy console launches successfully
- Map window displays vehicle location
- No critical error messages in terminal output
- Vehicle responds to basic commands (`arm throttle`, `mode GUIDED`)

Troubleshooting Common Issues:

- **Display errors**: Ensure VcXsrv is running and DISPLAY variable is correctly set **(For Windows 10)**
- **Permission errors**: Verify user has access to required directories and dialout group
- **Build failures**: Check that all prerequisites were installed successfully
- **Network timeouts**: Verify internet connectivity for package downloads

The installation is complete when SITL launches successfully with console and map interfaces operational.

3. Running SITL and Basic Operations

3.1 SITL Startup Procedures

Working Directory Setup Create a dedicated directory for SITL operations to organize simulation files and logs:

```
# Create and navigate to test directory
mkdir -p ~/sitl-test
cd ~/sitl-test
```

Standard SITL Launch Command The complete command structure for maritime SITL simulation:

```
sim_vehicle.py -v Rover -L Syros --out=udp:127.0.0.1:14550 \
--add-param-file=/home/<username>/custom-parms/boat.parm \
--map --console
```

3.2 Command Structure and Parameters

Command Component Breakdown:

`sim_vehicle.py:` Primary SITL execution script, accessible globally after installation

`-v Rover:` Specifies vehicle type as Rover (which includes boats when properly configured)

`-L syros:` Sets initial simulation location using custom location definition

`--out=udp:127.0.0.1:14550:` Configures UDP output for MAVLink telemetry communication. This creates a bidirectional connection point that Python scripts can use to communicate with the simulated vehicle

`--add-param-file=<path>:` Loads custom parameter file containing maritime-specific configurations including `FRAME_CLASS=2`

`--map:` Opens live map visualization showing vehicle position, path, and waypoints

`--console:` Launches MAVProxy console for real-time command interface

3.3 Interface Overview

MAVProxy Console The primary command interface provides real-time interaction with the simulated vehicle. Essential commands include:

```
# Vehicle arming/disarming
arm throttle
disarm

# Mode changes
mode MANUAL
mode GUIDED
mode AUTO
```

```
# Basic movement (in GUIDED mode)
# Right-click on map to command movement
```

Map Interface The map window displays:

- Vehicle current position and orientation
- Flight path history

- Planned waypoints and missions
- Real-time position updates during simulation

Connection Architecture The `127.0.0.1:14550` UDP endpoint serves as the primary connection point for:

- Ground Control Station software (Mission Planner)
- Custom Python scripts using DroneKit or PyMavlink
- Additional monitoring or control applications

3.4 ArduPilot Rover Modes for Maritime Applications

Maritime Configuration Requirements All maritime mode behaviors require `FRAME_CLASS=2` parameter to enable boat-specific features. This configuration fundamentally changes vehicle behavior in several modes compared to land-based rovers.

3.4.1 Manual Control Modes

MANUAL Mode Direct transmitter control of throttle and steering without autopilot intervention. Essential for:

- Initial vehicle testing and calibration
- Emergency manual override capability
- Basic operator familiarization

```
mode MANUAL
```

ACRO Mode Rate-controlled mode where steering input controls turn rate rather than absolute steering angle. Provides:

- Precise maneuvering control for advanced operators
- **Heading hold** when steering returns to neutral
- Speed control based on `CRUISE_THROTTLE` and `CRUISE_SPEED` parameters

```
mode ACRO
```

3.4.2 Position-Holding Modes: Critical Maritime Differences

HOLD Mode - Limited Maritime Utility In HOLD mode "the vehicle should stop and for regular steering-throttle rovers, the steering will point straight ahead". **For maritime applications, this creates significant operational problems**:

- **Motor shutdown**: Vehicle becomes completely passive
- **Environmental susceptibility**: Boat drifts uncontrollably with wind, waves, and current
- **No position correction**: Unlike land rovers that remain stationary, boats continue moving with environmental forces

```
mode HOLD    # Generally not recommended for boats
```

LOITER Mode - Maritime Position Hold Solution LOITER mode "allows boats to hold position in a strong current" through active position control:

Operation Mechanism:

- Vehicle drifts within `LOIT_RADIUS` of target position
- When exceeding radius: "rotates to point either directly towards the target or directly away from it (whichever results in less rotation)"
- Applies forward/reverse thrust "at 0.5 m/s * the distance to the edge of the circle around the target"

Key Parameters:

- `LOIT_RADIUS:` Distance from target before active correction begins
- `LOIT_SPEED:` Aggressiveness of position correction
- `WP_SPEED:` Maximum correction speed

```
mode LOITER
param set LOIT_RADIUS 5    # 5-meter radius before correction
```

3.4.3 Autonomous Navigation Modes

AUTO Mode - Mission Execution Executes pre-programmed missions consisting of "navigation commands (waypoints) and 'do' commands". Maritime-specific behaviors:

- **Position maintenance**: "In Auto, Guided, RTL and SmartRTL modes the vehicle will attempt to maintain its position even after it reaches its destination"
- **Environmental compensation**: Actively corrects for current and wind drift
- **Waypoint approach**: Uses maritime-optimized navigation algorithms

```
mode AUTO
```

GUIDED Mode - Real-time Navigation Accepts navigation commands from ground control stations or companion computers. Essential for:

- Dynamic mission modification
- Real-time obstacle avoidance
- Programmatic vehicle control via MAVLink

```
mode GUIDED
# Then right-click on map or send MAVLink commands
```

RTL Mode - Return to Launch Returns to launch position but with maritime-specific behavior: "hold position if a surface vehicle, or loiter/circle around the destination if it is a boat".

- **Navigation**: Direct path to launch location at `RTL_SPEED`
- **Arrival behavior**: Automatic LOITER mode around launch position

- **Safety function**: Reliable emergency return capability

```
mode RTL
param set RTL_SPEED 3    # 3 m/s return speed
```

Smart RTL Mode - Intelligent Path Return Smart RTL provides obstacle-aware return navigation by retracing the actual path traveled rather than taking a direct route home. This mode is particularly valuable for maritime operations in harbors, marinas, and areas with navigational obstacles.

How Smart RTL Works:

- **Path recording**: "The path used to return home is captured in a buffer as the vehicle flies around in any other mode"
- **Path optimization**: Route is "simplified" (curved paths become straight lines) and "pruned" (loops are removed)
- **Safe navigation**: Vehicle retraces the known-safe outbound path instead of attempting direct return

Maritime Applications:

- **Harbor navigation**: Avoids direct paths that would cross piers, breakwaters, or shallow areas
- **Marina operations**: Retraces the safe channel used during departure
- **Obstacle-rich environments**: Navigates around anchored vessels, buoys, or temporary obstacles encountered during outbound journey
- **Emergency return**: Provides reliable return path when direct RTL would be unsafe

Smart RTL vs. Regular RTL Comparison:

```
# Regular RTL: Direct line to home (may cross obstacles)
mode RTL

# Smart RTL: Retrace safe outbound path
```

```
mode SMART_RTL
```

Key Parameters:

- `SRTL_ACCURACY:` Path simplification tolerance in meters (default balances accuracy vs. storage)
- `SRTL_POINTS:` Maximum stored path points (each 100 points requires ~3KB RAM)
- `RTL_SPEED:` Return speed for both RTL and Smart RTL modes

Operational Considerations:

- **GPS requirement**: Vehicle must have "good position estimate" when armed or Smart RTL will be disabled
- **Buffer limitations**: "SmartRTL deactivated: buffer full" warning when path storage exceeds capacity
- **Path quality**: Longer, more complex routes may exceed buffer capacity in extended missions
- **Maritime behavior**: Upon reaching home, boat will "loiter/circle around the destination" rather than simply stopping

Usage Strategy: Use Smart RTL for missions involving:

- **Complex harbor departures** requiring navigation around fixed obstacles
- **Marina or port operations** with narrow channels and restricted areas
- **Survey patterns** near coastlines with shallow water or obstacles
- **Any scenario** where the outbound path was specifically planned to avoid hazards

Use regular RTL when:

- **Open water operations** with clear direct path to home
- **Emergency situations** requiring fastest possible return

- **Simple missions** where outbound path offers no advantages over direct return

```
# Enable Smart RTL for harbor operations
mode SMART_RTL
# Vehicle will retrace safe departure path through harbor
```

3.4.4 Specialized Modes

STEERING Mode (*) Provides heading control while maintaining manual speed control:

- User controls throttle directly
- Autopilot maintains desired heading
- Useful for assisted navigation in challenging conditions

CIRCLE Mode Executes circular patterns around a specified point:

- Useful for survey applications
- Maintains constant radius from center point
- Maritime applications include area monitoring and station-keeping

3.4.5 Mode Selection Strategy for Maritime Operations

Recommended Mode Assignments for 3-Position Switch:

1. **MANUAL**: Direct control for launch, recovery, and emergency situations
2. **LOITER**: Position holding for station-keeping and operational pauses
3. **AUTO**: Mission execution for autonomous operations

Alternative Configuration:

1. **MANUAL**: Emergency control
2. **GUIDED**: Operator-directed navigation
3. **RTL**: Emergency return function

Mode Transition Considerations:

- Always test mode changes in controlled conditions
- Verify GPS lock before engaging autonomous modes
- Understand environmental conditions affecting each mode's performance
- **Maintain manual override capability** in all operational scenarios

3.5 Basic Vehicle Control and Testing

Initial Mode Testing Sequence After successful SITL startup, verify mode functionality:

```
# 1. Test manual mode (should respond immediately)
# We will use this in Sea Trials with RC Control
mode MANUAL
arm throttle

# 2. Test guided mode navigation
mode GUIDED
# Right-click on map to command movement

# 3. Test loiter capability
mode LOITER
# Vehicle should maintain current position

# 4. Test return functionality
mode RTL
# Vehicle should return to launch point
```

Connection Verification Confirm UDP connection is operational for future Python integration:

```
# Check MAVProxy outputs
output
# Should show: 127.0.0.1:14550 among listed connections
```

Parameter Validation Verify maritime configuration is properly loaded:

```
param show FRAME_CLASS    # Should return: 2
param show BATT_CAPACITY  # Should return: 30000
```

This establishes the foundation for both immediate SITL operation and future integration with companion computer systems in Lab04.

4. Mission Planning and Auto Mode Operations

4.1 Mission Planning Fundamentals

Mission Structure and Concepts ArduPilot missions consist of sequential commands executed automatically when the vehicle operates in AUTO mode. Navigation commands "control the movement of the vehicle, including takeoff, moving to and around waypoints" while maintaining maritime-specific behaviors when `FRAME_CLASS=2` is configured.

Mission Command Types For maritime applications, the primary mission commands include:

- **NAV_WAYPOINT**: Navigate to specified position coordinates
- **NAV_LOITER_UNLIM**: Loiter indefinitely at specified location (maritime position holding)
- **NAV_LOITER_TIME**: Loiter for specified duration then continue mission
- **NAV_RETURN_TO_LAUNCH**: Return to launch position and maintain station

Coordinate System Rover missions use "MAV_FRAME_GLOBAL_RELATIVE_ALT frame, which uses the same latitude and longitude, but sets altitude as relative to the home position (home altitude = 0)". For maritime operations, altitude typically remains at 0 meters relative to launch position.

Mission Execution Behavior When missions complete, "Rover will hold" at the final waypoint. For boats with `FRAME_CLASS=2,` this engages position-holding behavior similar to LOITER mode. Missions "reset to the beginning of the mission list every disarm", ensuring consistent starting conditions.

4.2 Mission Planning with MAVProxy

Interactive Mission Creation MAVProxy provides intuitive map-based mission planning through its integrated map interface:

```
# Right-click on map, select: Mission > Draw
# Left-click on map to place waypoints sequentially
# Waypoints appear as numbered markers with connecting lines
```

Essential MAVProxy Mission Commands

```
# View currently loaded mission
wp list

# Save current mission to file
wp save mission_name.txt

# Load mission from file
wp load mission_name.txt

# Clear all waypoints (use with caution)
wp clear

# Execute mission
mode AUTO
arm throttle
```

Basic Waypoint Mission Example The most fundamental autonomous mission pattern involves navigating from home position through a series of waypoints and returning to home. This pattern forms the foundation for maritime autonomous operations.

Create directory missions (in your home directory or elsewhere, if you prefer). Then, inside it:

Sample Simple Mission: Create a mission file `simple_nisaki.txt` with the following content:

QGC WPL 110

0 1 0 16 0 0 0 0 37.438788 24.945544 0 1

1 0 3 16 0 0 0 0 37.436093 24.945544 0 1

2 0 3 16 0 0 0 0 37.438788 24.945544 0 1

3 0 3 16 0 0 180 0 37.438000 24.945544 0 1

Sample Longer Mission: Create a mission file `nisaki_long.txt` with the following content:

QGC WPL 110

0 1 0 16 0 0 0 0 37.438788 24.945544 0 1

1 0 3 16 0 0 0 0 37.438500 24.945556 0 1

2 0 3 16 0 0 0 0 37.436694 24.943688 0 1

3 0 3 16 0 0 0 0 37.434360 24.943989 0 1

4 0 3 16 0 0 0 0 37.434326 24.950666 0 1

5 0 3 16 0 0 0 0 37.443268 24.954376 0 1

6 0 3 16 0 0 0 0 37.443978 24.948330 0 1

7 0 3 16 0 0 0 0 37.434744 24.948587 0 1

8 0 3 16 0 0 0 0 37.435255 24.944293 0 1

9 0 3 16 0 0 0 0 37.439242 24.942790 0 1

10 0 3 16 0 0 180 0 37.438000 24.945544 0 1

Mission File Format Explanation:

- **Line 1**: `QGC WPL 110` - File format header

- **Waypoint 0**: Home position (37.438788 24.945544) - launch point in Syros 'Nisaki' area
- **Waypoints 1-10**: Sequential navigation coordinates forming patrol pattern
- **Key Parameters**:
    - `0` (column 2): Current waypoint flag (0 = not current, 1 = current)
    - `3` (column 3): Coordinate frame (MAV_FRAME_GLOBAL_RELATIVE_ALT)
    - `16` (column 4): Command type (MAV_CMD_NAV_WAYPOINT)
    - `0.000000` (column 11): Altitude relative to home (0 meters for maritime operations)
    - `1` (column 12): Autocontinue to next waypoint

Mission Execution Process:

```
# Load the mission file
wp load ~/missions/nisaki_long.txt

# Verify mission loaded correctly
wp list
# Should display 10 waypoints (0-9)

# Start mission execution
arm throttle
mode AUTO

# Vehicle will automatically:
# 1. Navigate to waypoint 1
# 2. Proceed through waypoints 2-10 sequentially
# 3. Hold position at final waypoint (maritime LOITER behavior)
```

Maritime Mission Considerations:

- **Zero altitude**: All waypoints use 0-meter altitude for surface vessel operations

- **Position holding**: Upon mission completion, boat maintains station at final waypoint
- **Environmental adaptation**: Mission execution automatically compensates for wind and current
- **Emergency procedures**: Mode changes (MANUAL, RTL) interrupt mission for operator control

This basic pattern serves as the foundation for more complex autonomous behaviors and provides essential experience with waypoint navigation before advancing to companion computer integration.

Mission Validation and Testing Before execution, verify mission parameters:

```
# Check mission waypoint count and sequence
wp list

# Verify vehicle configuration
param show FRAME_CLASS    # Should be 2 for boats
param show WP_SPEED       # Mission execution speed
```

4.3 Mission Planning with Mission Planner

Flight Plan Interface Mission Planner provides comprehensive graphical mission planning through the **FLIGHT PLAN** tab. The interface offers intuitive point-and-click waypoint creation with real-time mission visualization.

Basic Mission Creation Workflow:

1. **Select FLIGHT PLAN tab** in Mission Planner interface
2. **Left-click on map** to add waypoints sequentially
3. **Configure waypoint parameters** in the mission list panel
4. **Write mission to autopilot** using the "Write" button
5. **Verify mission upload** using the "Read" button

Mission Management Functions:

- **Save WP File**: Export missions to local storage for reuse
- **Load WP File**: Import previously created mission files
- **Right-click options**: Insert waypoints, modify existing points, measure distances
- **Mission commands dropdown**: Select appropriate command types for each waypoint

Mission Planner Advantages:

- **Visual mission preview** with distance and time estimates
- **Terrain awareness** with optional altitude verification
- **Grid pattern generation** for systematic area coverage
- **Mission validation** with pre-flight checks

4.4 Mission File Formats

File Structure Considerations ArduPilot supports various mission file formats including `.waypoints` and `.txt` formats. Mission files are "whitespace-delimited" with "tab-delimited" fields "formatted in the same order as the fields for a MISSION_ITEM MAVLink message".

Important Compatibility Note Mission file format specifications remain somewhat underdocumented and may vary between software versions. When working with mission files, verify compatibility through testing with small missions before deploying complex operations. Always validate imported missions using `wp list` command verification.

Recommended Approach For laboratory work, prioritize interactive mission creation through MAVProxy or Mission Planner interfaces rather than manual file editing. This ensures proper command formatting and parameter validation.

4.5 Programmatic Mission Control *(Lab04 will be dedicated to this)*

Python Integration Overview Future companion computer integration will enable programmatic mission control through Python libraries. **DroneKit** provides simplified mission management, while **PyMavlink** offers lower-level MAVLink protocol access.

DroneKit Mission Capabilities (Lab04 Preview):

- Mission upload and download via MAVLink connection
- Real-time mission modification during execution
- Mission item creation with coordinate and command specification
- Integration with telemetry data for adaptive mission planning

MQTT Communication Integration Maritime operations utilize **MQTT messaging** for shore-to-vessel and inter-vessel coordination. Mission data and waypoint updates can be transmitted via MQTT protocols, enabling:

- **Remote mission modification** from shore-based control stations
- **Coordinated multi-vessel operations** with shared waypoint data
- **Dynamic mission adaptation** based on environmental conditions or operational requirements

4.6 AUTO Mode Detailed Operations

Mission Initialization and Execution AUTO mode execution begins immediately upon mode selection if the vehicle is armed and a valid mission exists:

```
# Mission execution sequence
mode AUTO
arm throttle
# Vehicle immediately begins navigation to waypoint 1
```

4.7 Environmental Mission Testing

Meltemi Conditions Testing Using the previously configured `meltemi.parm` environmental parameters, missions can be tested under realistic Cyclades maritime conditions. Launch SITL with environmental simulation:

```
sim_vehicle.py -v Rover -L syros \
--add-param-file=/home/<username>/custom-parms/boat.parm \
--add-param-file=/home/<username>/custom-parms/meltemi.parm \
```

```
--console --map
```

Expected Environmental Effects Under simulated Meltemi conditions, observe realistic maritime challenges:

- **Reduced ground speed** when navigating against 12 m/s northerly winds
- **Increased battery consumption** due to additional thrust requirements for position maintenance
- **Extended mission duration** as vehicles compensate for environmental forces
- **Enhanced loiter performance** demonstrating active position-holding capabilities

Mission Performance Validation Environmental testing validates mission planning effectiveness and reveals parameter optimization opportunities for real-world deployment. Monitor mission execution for:

- Waypoint arrival accuracy under environmental stress
- Battery consumption patterns during challenging conditions
- Navigation algorithm performance in adverse weather simulation

This comprehensive mission planning foundation prepares students for advanced autonomous operations and sets the stage for companion computer integration in Lab04.

5. Multi-Vehicle SITL Operations (Advanced)

5.1 Multi-Vehicle Support Overview

SITL provides capabilities for simulating multiple vehicles simultaneously, enabling development and testing of coordinated autonomous behaviors, swarm operations, and inter-vessel communication protocols. This functionality supports maritime scenarios involving multiple boats operating in formation or coordinated missions.

Official Multi-Vehicle Approach ArduPilot SITL includes built-in support for launching multiple vehicles:

```
# Launch multiple rovers with automatic system ID assignment
sim_vehicle.py -v Rover --count 2 --auto-sysid --auto-offset-line
90,10 \
--location syros --map --console
```

Current Implementation Challenges Multi-vehicle SITL simulation remains a developing area with known limitations and bugs. Connection management between multiple vehicles and ground control stations can be inconsistent, particularly when attempting to establish discrete UDP connections for each vehicle.

5.2 Practical Multi-Vehicle Workaround

Separate Instance Approach Based on practical experience and community solutions, a reliable approach involves running independent SITL instances for each vehicle:

Instance 1 (Primary vessel):

```
sim_vehicle.py -v Rover -L syros \
--add-param-file=/home/<username>/custom-parms/boat.parm \
--map --console --out=udp:127.0.0.1:14550
```

Instance 2 (Secondary vessel):

```
sim_vehicle.py -v Rover -L syros2 \
--add-param-file=/home/<username>/custom-parms/boat.parm \
--instance 1 --console --map --sysid=2 \
--out=udp:127.0.0.1:14560 --out=udp:<host_ip>:14550
```

Connection Strategy:

- Each SITL instance operates independently with unique system IDs
- Both instances can transmit to Mission Planner on the same port for unified visualization

- Discrete localhost connections enable independent Python script control of each vessel
- Multiple terminal windows provide separate MAVProxy access for each vehicle

Connection String Configuration The multi-vehicle setup requires careful connection string management to enable both unified visualization and independent control access.

For WSL2/Windows Environment:

```
# Instance 1 (Primary vessel) - System ID 1
sim_vehicle.py -v Rover -L syros \
--add-param-file=/home/<username>/custom-parms/boat.parm \
--map --console --out=udp:127.0.0.1:14550
```

```
# Instance 2 (Secondary vessel) - System ID 2
sim_vehicle.py -v Rover -L syros2 \
--add-param-file=/home/<username>/custom-parms/boat.parm \
--instance 1 --console --map --sysid=2 \
--out=udp:127.0.0.1:14560 --out=udp:<windows_host_ip>:14550
```

Connection String Details:

- **Instance 1 localhost**: `127.0.0.1:14550` (accessible to Python scripts in WSL2)
- **Instance 2 localhost**: `127.0.0.1:14560` (discrete connection for second vehicle)
- **Windows Mission Planner**: `<windows_host_ip>:14550` (both vehicles visible on same map)

Determining Windows Host IP:

```
# From WSL2, find Windows host IP
grep nameserver /etc/resolv.conf | awk '{print $2}'
# Example output: 172.21.176.1
```

Python Script Connection Strings (Lab04 Preview):

```
# Vehicle 1 connection
connection_string_1 = "127.0.0.1:14550"

# Vehicle 2 connection
connection_string_2 = "127.0.0.1:14560"
```

Connection Verification:

```
# Verify SITL outputs are configured correctly
# In MAVProxy console for each instance:
output

# Expected Instance 1 output:
# 0: 127.0.0.1:14550

# Expected Instance 2 output:
# 0: 127.0.0.1:14560
# 1: <windows_host_ip>:14550
```

Mission Planner Integration:

1. **Start Mission Planner** on Windows host
2. **Connect to UDP port 14550**
3. **Both vehicles appear** on same map with different system IDs
4. **Python scripts connect** independently to discrete localhost ports

This configuration enables complete multi-vehicle testing with unified visualization and independent programmatic control.

5.3 Multi-Vehicle Applications

Coordinated Maritime Operations Multi-vehicle simulation enables testing of:

- **Formation sailing** with multiple vessels maintaining relative positions
- **Leader-follower** behaviors for automated convoy operations
- **Distributed survey** patterns covering larger areas efficiently
- **MQTT-based coordination** between vessels and shore control

Communication Testing Independent vehicle instances allow validation of:

- **Inter-vessel messaging** via MQTT protocols
- **Shore-based coordination** with multiple vessel monitoring
- **Collision avoidance** algorithms between autonomous vessels
- **Mission synchronization** across multiple platforms

5.4 Known Issues and Community Resources

Connection Management Bug A documented issue exists with MAVProxy's handling of system ID-specific UDP outputs, where adding discrete connections can cause loss of control for all vehicles. Community discussion and developer acknowledgment available at: [ArduPilot Discourse - Multiple Rovers SITL Setup]

Development Status Multi-vehicle SITL functionality continues to evolve. While basic capabilities exist, complex swarm operations may require specialized setup procedures or workarounds. The separate instance approach provides reliable functionality for most multi-vehicle testing scenarios.

Future Development ArduPilot development continues to improve multi-vehicle simulation capabilities. Students interested in advanced swarm behaviors should monitor community forums for updates and contribute to testing efforts when possible.

This multi-vehicle capability extends SITL's utility for testing complex autonomous behaviors and prepares the foundation for real-world multi-vessel operations using the companion computer architectures introduced in Lab04.

6. SITL Session Management and Troubleshooting

6.1 Proper SITL Shutdown Procedures

Correct Shutdown Sequence Proper SITL termination ensures parameter persistence and prevents corruption of simulation data:

```
# 1. Disarm vehicle first (if armed)
disarm

# 2. Stop any running missions
mode MANUAL

# 3. Exit MAVProxy via keyboard interrupt
Ctrl+C

# 4. Terminate SITL with Ctrl+C in main terminal
```

6.2 Shutdown Process

When you press `Ctrl+C` to terminate the simulation, it performs a graceful shutdown. The key action is that the simulator saves its current state to files, most importantly writing all parameters to `eeprom.bin`.

These files are always saved in the current working directory from which the `sim_vehicle.py` script was launched. So, if you run the simulation from your `~/sitl-tests` directory, the `eeprom.bin` file will be created and updated in `~/sitl-tests`. This ensures that your configuration is preserved for the next session you launch from that same directory. After saving, all processes are terminated cleanly.

6.3 Connection Verification and UDP Testing

Basic Connection Health Check Verify SITL is ready for external connections before attempting Python script integration:

```
# Check MAVProxy output configuration
output
```

```
# Expected output should include:
# 0: 127.0.0.1:14550

# Verify UDP port is active (Linux/WSL)
netstat -an | grep 14550
# Should show: udp 0.0.0.0:14550
```

Connection String Validation Test basic UDP connectivity before Lab04 integration:

```
# From separate terminal, test UDP port accessibility
telnet 127.0.0.1 14550
# Connection successful indicates UDP endpoint is active
```

6.4 Common SITL Issues and Solutions

Display and Graphics Problems

```
# WSL2 X server issues
echo $DISPLAY
# Should output: <IP>:0
# If empty, restart VcXsrv and reload ~/.bashrc

# Map window not appearing
source ~/.profile
# Ensure PATH includes SITL tools
```

Build and Dependency Issues

```
# SITL build failures
cd ~/ardupilot
./waf clean
./waf configure --board sitl
./waf rover
```

```
# Missing dependencies
Tools/environment_install/install-prereqs-ubuntu.sh -y
source ~/.profile
```

Parameter Loading Problems

```
# Custom parameter file not loading
# Verify file path is absolute
--add-param-file=/home/<username>/custom-parms/boat.parm

# Check parameter file syntax
# Ensure format: PARAM_NAME VALUE
# No extra spaces or special characters
```

Mission File Compatibility

```
# Mission loading failures
wp list    # Check if any mission is loaded
wp clear   # Clear existing mission
wp load mission_file.txt    # Reload mission

# Verify mission file format
# First line should be: QGC WPL 110
# Subsequent lines: tab-delimited values
```

6.5 Log Files and Debugging Information

SITL Log Locations

```
# SITL generates logs in working directory
ls -la *.bin *.log
```

```
# Key log files:
# mav.tlog - MAVLink telemetry log
# terrain/ - Terrain data cache
# eeprom.bin - Parameter storage
```

Basic Log Analysis

```
# MAVProxy flight data logs
ls logs/
# Contains timestamped flight sessions

# Mission Planner can replay .tlog files
# Useful for mission validation and debugging
```

This session management foundation ensures reliable SITL operation and smooth transition to companion computer integration in Lab04, where students will build upon these verified connections to implement Python-based autonomous control systems.