Guia empírico - ArduPilot e SITL via MAVProxy no Gazebo Harmonic

Primeiro, um disclaimer importante: este guia foi montado de acordo com o Ubuntu 24.04

Tenha pelo menos 4GB de armazenamento sobrando para este projeto

As informações gerais foram retiradas de pesquisas na internet, mas a principal fonte, que inclusive é a oficial, é: https://ardupilot.org/dev/docs/sitl-with-gazebo.html

PRÉ REQUISITOS

Antes de tudo, tenha certeza de atualizar seu sistema:

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

Dependências gerais:

```
sudo apt install git curl wget lsb-release gnupg software-properties-common
-y
```

Ambos os comandos acima devem ser feitos na pasta root do sistema (geralmente ~/), fora de uma venv. Após isso, podemos começar o setup das simulações.

Mais dependências (não tenho certeza se já estão em outro lugar):

```
sudo apt update
sudo apt install -y git wget python3 python3-dev python3-pip python3-
setuptools \
    python3-numpy python3-opencv python3-pyparsing python3-serial \
    python3-scipy python3-pygame python3-lxml python3-dev \
    build-essential ccache g++ gawk make \
    libtool libxml2-dev libxslt-dev \
    pkg-config genromfs \
    libncurses-dev libudev-dev libsdl1.2-dev libtool \
    libxml2-utils python3-matplotlib
```

1 - Criar pasta do projeto e Ambiente Virtual (venv)

Para evitar qualquer tipo de conflito, o ideal é se criar uma pasta onde tudo vai ser instalado. Sério. Não pule essa etapa, de verdade. O venv único também é importante.

Exemplo:

```
mkdir -p ~/sim_ardupilot_gazebo
cd ~/sim_ardupilot_gazebo
```

```
sudo apt install python3-venv -y
python3 -m venv venv
source venv/bin/activate
```

Após isso, instale algumas das dependências usadas para as simulações pelo comando:

```
pip install -r requirements.txt
```

O arquivo requirements.txt possui o seguinte script:

```
contourpy==1.3.2
cycler==0.12.1
fonttools==4.58.0
future==1.0.0
kiwisolver==1.4.8
lxml==5.4.0
matplotlib==3.10.3
MAVProxy==1.8.71
numpy==2.2.5
opencv-python==4.11.0.86
packaging==25.0
pillow==11.2.1
pymavlink==2.4.43
pynmeagps==1.0.50
pyparsing==3.2.3
pyserial==3.5
python-dateutil==2.9.0.post0
six==1.17.0
wxPython==4.2.3
```

2 - Instalar MAVProxy e pymavlink

O MAVProxy é a ferramenta que vai ser utilizada como SITL (Software-in-the-Loop, um simulador que permite testar o código do piloto automático de veículos aéreos não tripulados), ele é um software de estação terrestre baseado em linha de comandos que complementa as estações terrestres gráficas (GUI) como o Mission Planner. - Definição da internet.

!!! - CERTIFIQUE-SE DE ESTAR COM A VENV ATIVADA PELO COMANDO `source venv/bin/activate`

Para qualquer linux baseado em sistemas Debian (como o Ubuntu), a instalação é dada por:

```
sudo apt-get install python3-dev python3-opencv python3-wxgtk4.0 python3-pip
python3-matplotlib python3-lxml python3-pygame
python3 -m pip install PyYAML mavproxy
echo 'export PATH="$PATH:$HOME/.local/bin"' >> ~/.bashrc
```

Note que na terceira linha, diferente da fonte original, não há o comando `--user`, pois estamos instalando o MAVProxy diretamente na venv. A inclusão deste comando causaria erro, e a instalação do software fora de uma venv pode corromper o sistema operacional completo, além de causar outros tipos de falhas.
A instalação do pymavlink é dada pelo código:

```
pip install pymavlink MAVProxy
```

3 - Clonar o ArduPilot, configurar as variáveis de ambiente e compilar o SITL

Lembre-se de estar na pasta raiz do seu diretório das simulações (que neste exemplo é em `~/sim_ardupilot_gazebo`)

```
git clone https://github.com/ArduPilot/ardupilot.git
cd ardupilot
git submodule update --init --recursive
```

Após isso, configure as variáveis de ambiente (alterando de acordo com seu diretório):

```
echo "export PATH=$PATH:$HOME/sim_ardupilot_gazebo/ardupilot/Tools/autotest"
>> ~/.bashrc
echo "export PATH=$PATH:$HOME/.local/bin" >> ~/.bashrc
source ~/.bashrc
```

Instalar as dependências do ArduPilot e compilar o SITL

Ainda na pasta do raiz do ArduPilot (neste exemplo, `~/sim_ardupilot_gazebo/ardupilot`), instale as dependências para seu funcionamento através do comando:

```
./Tools/environment_install/install-prereqs-ubuntu.sh -y
. ~/.profile
```

Após isso, compile o SITL pela primeira vez, para o Copter (é um drone quadrotor):

```
cd ~/ardupilot
./waf configure --board sitl
./waf copter
```

Se houver algum erro, verifique o caminho cd e vá para a pasta raiz do clone do ArduPilot

Se tudo ocorrer normalmente, teste o SITL com MAVProxy pelo comando (lembra de estar na pasta `/ardupilot`):

```
../Tools/autotest/sim_vehicle.py -v ArduCopter -f gazebo-iris --model JSON --map --console
```

4 - Instalar e configurar o Gazebo Harmonic

Seguindo a orientação da documentação oficial do Gazebo Harmonic, primeiro deve-se instalar algumas ferramentas. Lembre-se de voltar para a pasta principal do projeto.

```
sudo apt-get update
sudo apt-get install curl lsb-release gnupg
```

Daí, instale o gazebo com o seguinte comando:

```
sudo curl https://packages.osrfoundation.org/gazebo.gpg --output /usr/share/keyrings/pkgs-osrf-archive-keyring.gpg
echo "deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings/pkgs-osrf-archive-keyring.gpg] http://packages.osrfoundation.org/gazebo/ubuntu-stable $(lsb_release -cs) main" | sudo tee /etc/apt/sources.list.d/gazebo-stable.list > /dev/null
sudo apt-get update
sudo apt-get install gz-harmonic
```

Teste se o gazebo foi instalado corretamente pelo comando `gz sim`. Caso haja algum erro, tente resolvê-lo pelos logs que o terminal retornar. Outro teste interessante para mostrar o funcionamento do gazebo é com o comando `gz sim -v4 -r shapes.sdf`.

5 - Instalar o plugin ardupilot_gazebo

Como o nome já diz, esse plugin tem a função de conectar o gazebo com o ardupilot através do SITL. Seguindo as orientações do repositório oficial para o Ubuntu, instalamos primeiro algumas dependências adicionais pelo comando a seguir. LEMBRE-SE DE ATIVAR

A VENV DO PROJETO E DE IR PARA A PASTA PRINCIPAL ANTES DE INSTALAR O PLUGIN.

```
sudo apt update
sudo apt install libgz-sim8-dev rapidjson-dev
sudo apt install libopencv-dev libgstreamer1.0-dev libgstreamer-plugins-
base1.0-dev gstreamer1.0-plugins-bad gstreamer1.0-libav gstreamer1.0-gl
```

Após isso, clone o repositório oficial do plugin por meio do seguinte comando:

```
git clone https://github.com/ArduPilot/ardupilot_gazebo
cd ardupilot_gazebo
mkdir build && cd build
cmake .. -DCMAKE_BUILD_TYPE=RelWithDebInfo
make -j4
```

para configurar as variáveis de ambiente do plugin, vamos inserir dois comandos. O primeiro ativa as variáveis para esta utilização no terminal. O segundo altera o .baschrc do sistema para manter as variaveis ativas. Este segundo código é útil pois faz com que o sistema consiga acessar as variáveis sempre, e sem ele, teríamos que rodar o primeiro comando toda vez que fôssemos utilizar o gazebo com o ardupilot.
Comando 1:

```
export
GZ_SIM_SYSTEM_PLUGIN_PATH=$HOME/sim_ardupilot_gazebo/ardupilot_gazebo/build:
$GZ_SIM_SYSTEM_PLUGIN_PATH
export
GZ_SIM_RESOURCE_PATH=$HOME/sim_ardupilot_gazebo/ardupilot_gazebo/models:$HOM
E/sim_ardupilot_gazebo/ardupilot_gazebo/worlds:$GZ_SIM_RESOURCE_PATH
```

Comando 2:

```
echo 'export
GZ_SIM_SYSTEM_PLUGIN_PATH=$HOME/sim_ardupilot_gazebo/ardupilot_gazebo/build:
$GZ_SIM_SYSTEM_PLUGIN_PATH' >> ~/.bashrc
echo 'export
GZ_SIM_RESOURCE_PATH=$HOME/sim_ardupilot_gazebo/ardupilot_gazebo/models:$HOM
E/sim_ardupilot_gazebo/ardupilot_gazebo/worlds:$GZ_SIM_RESOURCE_PATH' >>
~/.bashrc
source ~/.bashrc
```

Note que em ambos os comandos, estamos considerando que o plugin foi clonado dentro da pasta principal do projeto, que fica em `~/sim_ardupilot_gazebo` (identificado por `$HOME/sim_ardupilot_gazebo`)

6 - Testar e rodar uma simulação + Como rodar no Windows

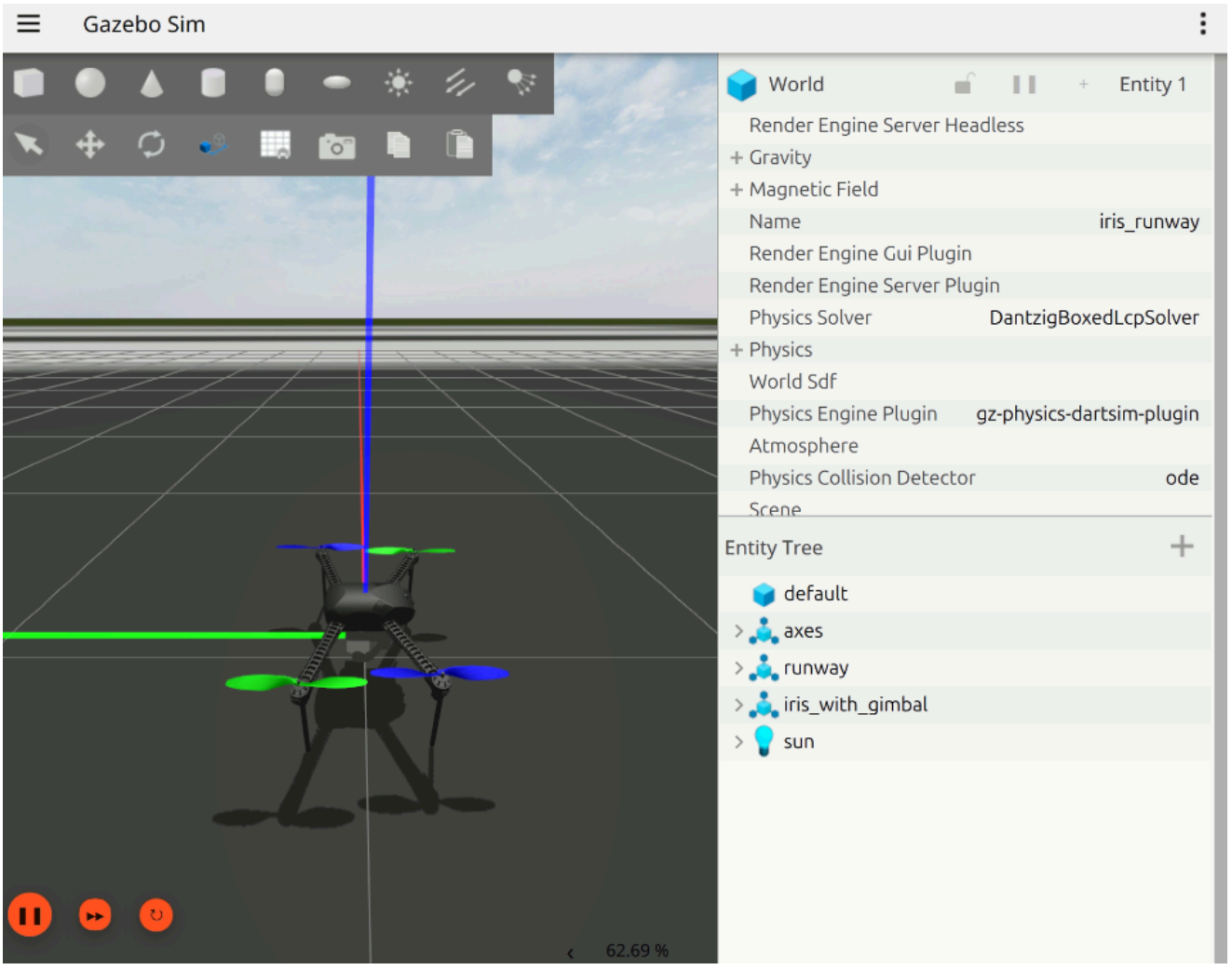
Teste do Gazebo

Agora, vamos testar uma simulação do Copter, do ardupilot, no gazebo. Caso haja algum erro no processo, procure resolvê-lo antes de seguir para o guia de utilização.

Em um terminal, na root do seu sistema operacional, e fora da venv, utilize o comando:

```
gz sim -v4 -r iris_runway.sdf
```

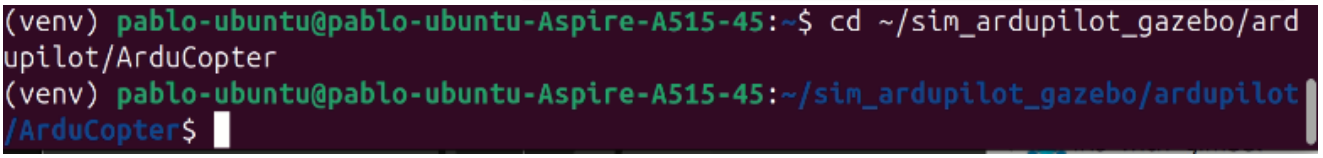
O terminal vai se transformar em um log com as informações do gazebo. Qualquer erro vai estar registrado nele, então fique atento. Após a compilação, o simulador 3D do gazebo vai abrir, e nele deve ter um drone quadrotor centralizado, como na imagem a seguir.



Em caso de erro, verifique se as variáveis de ambiente estão configuradas corretamente. Se necessário, execute o comando 1 da etapa anterior novamente.

Teste do SITL

Ainda com a simulação no gazebo aberta, abra outro terminal. Neste terminal, ative a venv do projeto e vá até a pasta ArduCopter, dentro do diretório clonado do ardupilot. No caso deste exemplo, seria pelo comando `cd ~/sim_ardupilot_gazebo/ardupilot/ArduCopter` .

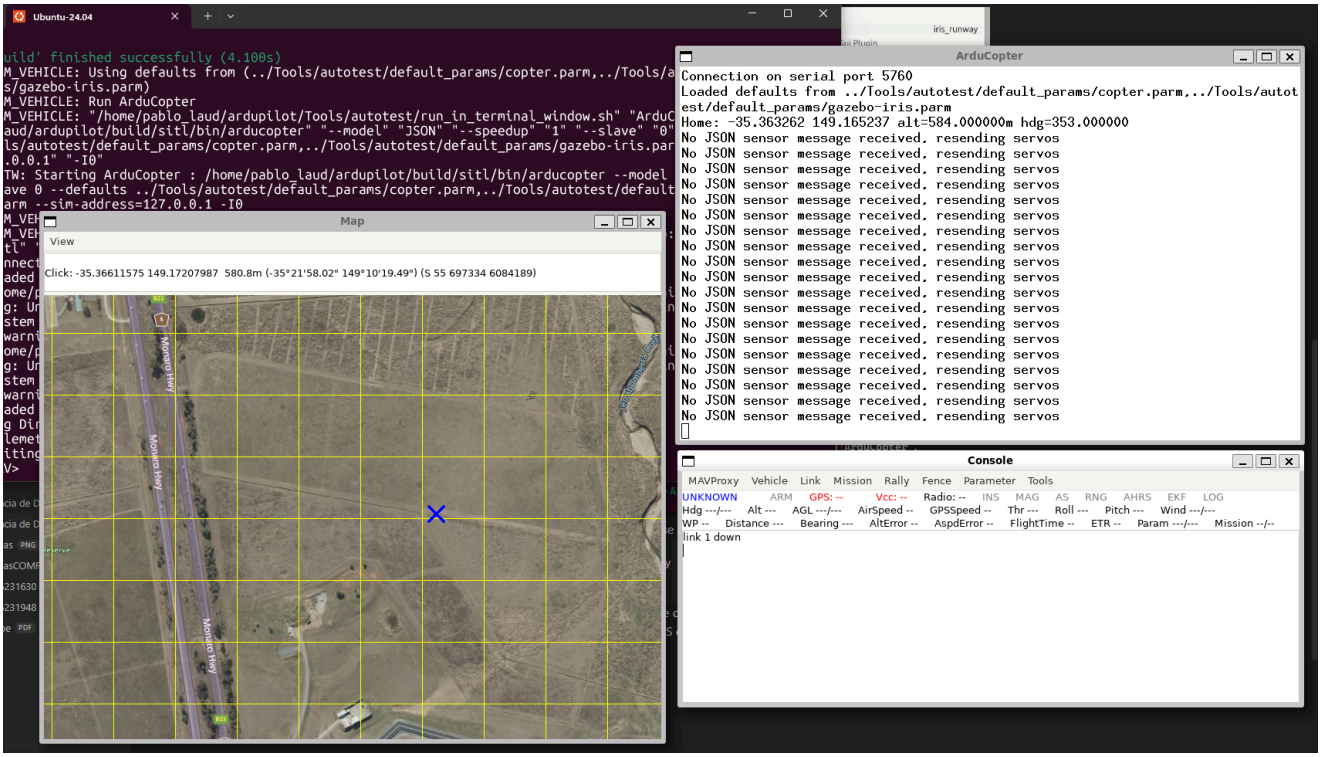


Após isso, vamos verificar se o SITL vai se conectar corretamente. Execute o comando:

```
../Tools/autotest/sim_vehicle.py -v ArduCopter -f gazebo-iris --model JSON
--console --map
```

Esse comando vai compilar o SITL, e qualquer erro será exibido no terminal, mesmo que este erro não impeça o processo de rodar, então fique atento caso haja algum problema ou algo faltando.

Após a compilação, o terminal se tornará um log do SITL. Deve abrir o console que mostra os dados de telemetria do drone em simulação, um outro log do SITL e também um mapa com GPS que mostra a localização do drone. Caso deseje encerrar a conexão, no terminal aperte `CTRL + C` . Neste caso, não vamos encerrar a conexão ainda.



Se o mapa não aparecer ou houver algum problema com as dependências do projeto, simplesmente instale de acordo com o resultado da compilação no terminal (Instale dentro da venv).

Teste de algum voo autônomo

Por último, vamos testar uma missão autônoma. A seguir, vou fornecer um código de teste, que guia o drone para um movimento incremental (aos poucos) no formato de um quadrado, juntamente com um botão de pouso emergencial via tkinter. Este arquivo deve ser salvo no formato .py, e salvo na sua pasta de preferência, onde guardará o código das outras simulações que fizer no futuro.
Disclaimer importante: Se, após a execução do comando, nada ocorrer (sem nem mesmo um log de erro), há alguma falha na execução do arquivo da missão. Encerre o processo com CTRL + C e leia o log de erro para tentar corrigir.

A primeira etapa é ter um arquivo para a simulação da missão. Neste caso, vamos usar o seguinte código:

Código exemplo: quad_in.py

```
import time
import math
import argparse
import threading
import tkinter as tk
from collections import deque
from pymavlink import mavutil

# ------------------------------------------------
# CONFIGURAÇÕES GLOBAIS
# ------------------------------------------------
MAX_INCREMENTO = 0.5         # distância máxima por comando (m)
PAUSA_ENTRE_MOVIMENTOS = 1.0 # tempo entre comandos (s)
TELEM_RATE_HZ = 5            # Hz para LOCAL_POSITION_NED
TOLERANCIA_POS = 0.1         # tolerância para chegada (m)

# evento para emergência
e_emergency = threading.Event()
# fila circular para última posição (x,y,z)
pos_queue = deque(maxlen=1)

# ------------------------------------------------
# FUNÇÕES DE CONEXÃO E PARÂMETROS
# ------------------------------------------------
def connect_drone():
    master = mavutil.mavlink_connection('udp:127.0.0.1:14550', baud=57600)
    master.wait_heartbeat()
    print("Conexão estabelecida!")
```

```
    # solicita telemetria LOCAL_POSITION_NED
    master.mav.request_data_stream_send(
        master.target_system, master.target_component,
        mavutil.mavlink.MAV_DATA_STREAM_POSITION,
        TELEM_RATE_HZ, 1
    )
    return master


def set_limits(master):
    params = {
        'MPC_XY_VEL_MAX': 1.0,
        'MPC_MAN_TILT_MAX': 15.0,
        'WP_YAW_BEHAVIOR': 0
    }
    for p, v in params.items():
        master.mav.param_set_send(
            master.target_system, master.target_component,
            p.encode(), v,
            mavutil.mavlink.MAV_PARAM_TYPE_REAL32 if isinstance(v, float)
            else mavutil.mavlink.MAV_PARAM_TYPE_INT32
        )
    print("Parâmetros configurados:", params)

# ------------------------------------------------
# THREAD DE TELEMETRIA
# ------------------------------------------------
def telemetry_reader(master):
    while not e_emergency.is_set():
        msg = master.recv_match(type='LOCAL_POSITION_NED', blocking=True,
timeout=1)
        if msg:
            pos_queue.append((msg.x, msg.y, -msg.z))

# ------------------------------------------------
# FUNÇÕES DE VOO
# ------------------------------------------------
def arm_and_takeoff(master, altitude):
    master.set_mode('GUIDED')
    master.mav.command_long_send(
        master.target_system, master.target_component,
        mavutil.mavlink.MAV_CMD_COMPONENT_ARM_DISARM,
        0, 1, 0,0,0,0,0,0
```

```
    )
    time.sleep(1)
    master.mav.command_long_send(
        master.target_system, master.target_component,
        mavutil.mavlink.MAV_CMD_NAV_TAKEOFF,
        0,0,0,0,0,0,0, altitude
    )
    print(f"Decolando para {altitude}m...")
    time.sleep(altitude * 2)
    print("Decolagem concluída.")


def move_increments(master, dx, dy, dz):
    # aguarda posição inicial
    while not pos_queue:
        time.sleep(0.1)
    x0, y0, z0 = pos_queue[0]
    # yaw atual
    msg = master.recv_match(type='ATTITUDE', blocking=True, timeout=3)
    yaw = math.degrees(msg.yaw)%360 if msg else 0.0
    print(f"Yaw atual: {yaw:.1f}°")
    # calcula passos
    dist = math.sqrt(dx*dx + dy*dy + dz*dz)
    steps = max(1, int(dist / MAX_INCREMENTO))
    xi, yi, zi = dx/steps, dy/steps, dz/steps
    print(f"Movendo em {steps} passos: ΔX={dx}, ΔY={dy}, ΔZ={dz}")
    mask = 0b0000111111000000
    for i in range(1, steps+1):
        if e_emergency.is_set():
            print("Movimento interrompido por emergência.")
            return
        tx = x0 + xi*i
        ty = y0 + yi*i
        tz = z0 + zi*i
        master.mav.send(
            mavutil.mavlink.MAVLink_set_position_target_local_ned_message(
                0, master.target_system, master.target_component,
                mavutil.mavlink.MAV_FRAME_LOCAL_NED,
                mask,
                tx, ty, -tz,
                0,0,0,0,0,0,
                math.radians(yaw), 0
            )
```

```
        )
        # espera alvo
        while True:
            if e_emergency.is_set(): return
            rx, ry, rz = pos_queue[0]
            if abs(rx-tx)<TOLERANCIA_POS and abs(ry-ty)<TOLERANCIA_POS:
                break
            time.sleep(0.1)
        print(f"Alcançado => X={rx:.2f}, Y={ry:.2f}, Z={rz:.2f}")


def land_drone(master):
    master.mav.command_long_send(
        master.target_system, master.target_component,
        mavutil.mavlink.MAV_CMD_NAV_LAND,
        0,0,0,0,0,0,0,0
    )
    print("Pousando...")
    time.sleep(5)
    if pos_queue:
        fx, fy, fz = pos_queue[0]
        print(f"Posição final após pouso: X={fx:.2f}, Y={fy:.2f}, Z=
{fz:.2f}")

# ------------------------------------------------
# INTERFACE DE EMERGÊNCIA
# ------------------------------------------------
def start_emergency_ui(master):
    def on_land():
        print("EMERGÊNCIA acionada! Pouso imediato.")
        e_emergency.set()
        land_drone(master)
        root.destroy()
    root = tk.Tk()
    root.title("EMERGÊNCIA")
    root.geometry("220x100")
    btn = tk.Button(root, text="POUSAR AGORA", fg="white", bg="red",
                    font=("Helvetica",16,"bold"), command=on_land)
    btn.pack(expand=True, fill="both", padx=10, pady=10)
    root.mainloop()


def launch_emergency_ui(master):
```

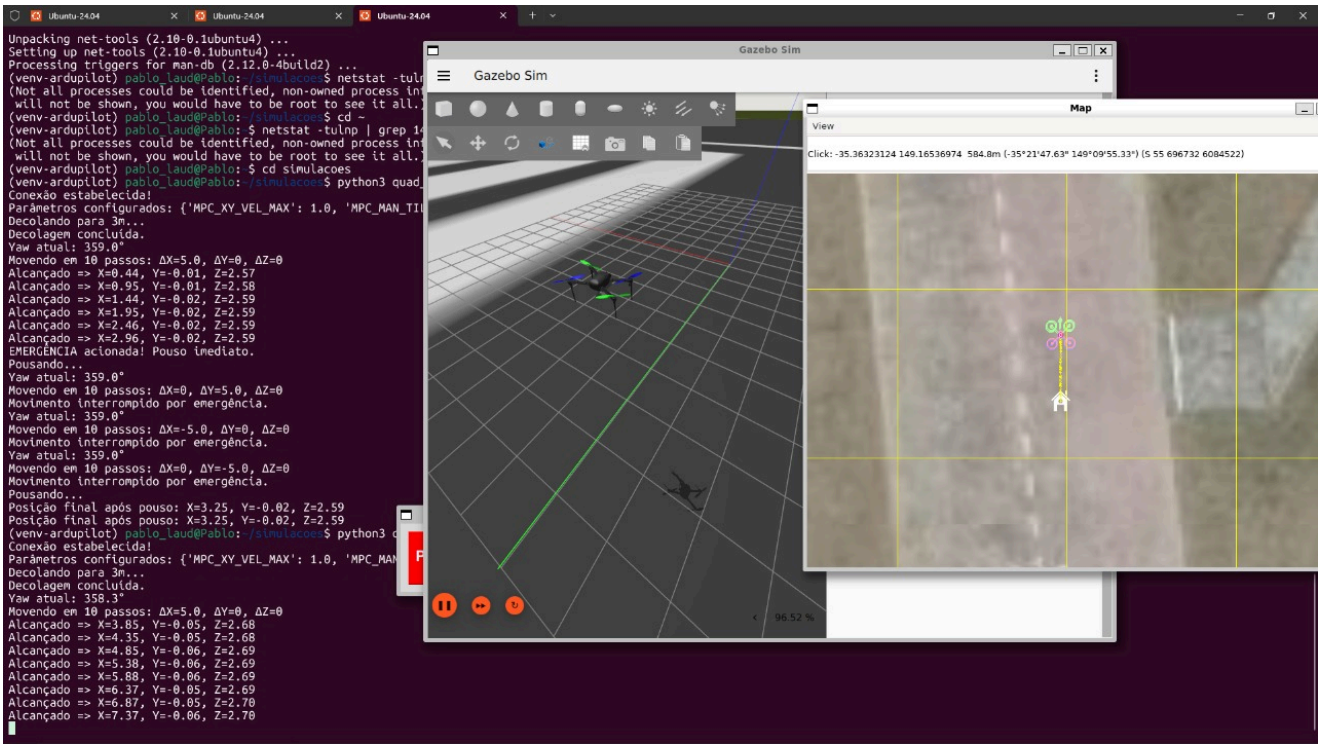
```
    threading.Thread(target=start_emergency_ui, args=(master,),
daemon=True).start()

# ------------------------------------------------
# MAIN
# ------------------------------------------------
if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument('--size', type=float, default=5.0,
                        help='tamanho do quadrado')
    args = parser.parse_args()
    master = connect_drone()
    set_limits(master)
    threading.Thread(target=telemetry_reader, args=(master,),
daemon=True).start()
    launch_emergency_ui(master)
    arm_and_takeoff(master, 3)
    if not e_emergency.is_set():
        # reutiliza fly_perfect_square
        for axis, dist in [('x', args.size),('y',args.size),('x',-
args.size),('y',-args.size)]:
            if axis=='x': move_increments(master, dist,0,0)
            else: move_increments(master, 0,dist,0)
    land_drone(master)
    master.close()
```

Execução

Verifique se o SITL está funcionando normalmente. Após o retorno de `pre-arm good` no console, podemos continuar.
Ative o venv do projeto e vá para a pasta onde está o código da simulação. Lá, execute o comando normalmente:

```
python3 quad_inc.py
```

A partir daí, você vai receber os retornos de posição do drone, e ele vai realizar a missão (com resultados no mapa via GPS e no 3D do gazebo).



Extra: Utilizando no Windows, via Ubuntu WSL

Caso você esteja no Windows, a única forma de rodar a simulação na configuração que desejamos é instalar um Ubuntu 24.04 por meio do WSL. A partir daí, o processo é praticamente idêntico, com exceção de um detalhe: a execução da missão, o código dela e as saídas do MAVProxy devem ser alterados.

Isso acontece porque, ao executar pelo WSL, o MAVProxy tem problemas de conexão, e precisa de uma porta dedicada somente à saída dos dados, que será utilizada para a conexão com o drone.

Realize todos os passos até a abertura do SITL normalmente. Após isso, no terminal do MAVProxy, adicione a porta desejada. Um exemplo é:

```
output add 127.0.0.1:14550
```

Daí, no arquivo quad_inc.py (utilizado no exemplo anterior), na função `connect_drone()`, deve-se alterar a linha de código que define a conexão, trocando o ip. Segue o exemplo:
ANTES

```
master = mavutil.mavlink_connection('udp:127.0.0.1:14550', baud=57600)
```

DEPOIS

```
master = mavutil.mavlink_connection('udp:127.0.0.1:14550')
```

Após isso, vá até o diretório do arquivo da missão (quad_inc.py) e execute normalmente.

7 - Guia de utilização

Aqui está um roteiro resumido, que deve ser seguido sempre para o setup inicial das simulações. Vamos precisar de ***TRÊS* (3) terminais diferentes**.

1 - Abra o primeiro terminal e execute o comando a seguir para abrir o Copter no Gazebo

```
gz sim -v4 -r iris_runway.sdf
```

2 - Abra outro terminal e ative a venv do projeto. Depois, vá para o diretório ArduCopter, dentro da pasta clonada do ardupilot (ex.: `cd ~/sim_ardupilot_gazebo/ardupilot/ArduCopter`). Lá, execute o seguinte comando para compilar e iniciar o SITL pelo MAVProxy:

```
 ../Tools/autotest/sim_vehicle.py -v ArduCopter -f gazebo-iris --model JSON
--console --map
```

Se estiver no Windows, via WSL, lembre-se de adicionar a porta de saída no MAVProxy e alterar o código da simulação.

3 - Após o console do SITL retornar a mensagem `pre-arm good`, abra o terceiro terminal e ative a venv no projeto também neste terminal. Daí, execute o script da missão desejada normalmente como no exemplo abaixo.

```
python3 quad_inc.py
```