Hybrid Control Car

This project will demonstrated a hybrid machine learning and optimal control theory approach to controlling a simulated racecar. A neural network - which will henceforce be referred to as the pathfinding network - will be trained using reinforcement learning to pathfind, selecting the position of and the desired velocity of the car at the next path node based on sensor readings and the current state of the car. An inner control loop will use model predictive control with LQR feedback to drive the car to the next node.

The underlying physics simulation is written in the Rust programming language using Rapier as the physics engine with Python bindings generated by PyO3. Its source code is avaiable at src/lib.rs.

We will first start by importing the necessary libraries and establishing a few parameters.

In [1]:
```
from hybrid_control_car import CarSimulation # The simulation developed for this pr
import numpy as np
import scipy as sp
import matplotlib.pyplot as plt
import pysindy

nstates = 4 # The number of state variables (x, z, v_x, v_z)
ninputs = 2 # The number of inputs (F_e, phi)
```

1 Deriving a Model of the Car

While an accurate model of the car itself could be derived mathematically, for the purposes of this project we will not attempt to do so, treating the car as a standing for a far more complex system. To that end, we will use Sparse Indentification of Nonlinear Dynamics (SINDy) to create a model of the system computationally using the pySINDy library. This algorithm was chosen for several reasons:

1. It may be difficult to linearize about an equilibrium point while still maintaining a high level of accuracy.
2. pySINDy allows for finding the nonlinear dynamics using multiple trajectories of test data.
3. []

A model with at least 75% accuracy for the first few seconds is desired. Later on, feedback control will be used to compensate for any inaccuracies in the model, so a near-perfect model is not necessary, but we will aim to create one anyways.

1.1 Simulation and Analysis

Selecting the state and input vectors requires balancing adequet representation of the system and control in the inner MPC controller loop with the complexity of the model. A model with more states specifically will require the pathfinding network to determine more states for the next node and will make it more difficult to fit the SINDy model with accuracy. With these considerations, the following state and input vectors have been selected.

$$\vec{x} = \begin{bmatrix} x \\ z \\ v \\ \phi \end{bmatrix}, \vec{u} = \begin{bmatrix} \theta \\ F_e \end{bmatrix}$$

Where x and z are the position of the car along those axis, v is the forward velocity scalar of the car, ϕ is the rotation of the car about the y-axis, θ is the steering angle of the car, and F_e is the force exerted by the engine. While the angular velocity about the y-axis could be considered, it seems like it would add more complexity to the model for very little gain.

In terms of units, we will define one second as equivalent to 60 simulation timesteps. Since x and z are returned in units and v is returned in units per timestep, we will define one meter as equivalent to 60 units, otherwise we will get very large velocities and distances.

Training data will be generated by holding both inputs steady at multiple different values, holding one steady while one is changes, and by having both change; Testing data will be generated in the same way.

In [3]:
```
# Set up the simulation
sim = CarSimulation()
sim.create_floor()

hz = 60 # Timesteps per second
upm = 60 # Units per meter

def simulate(sim: CarSimulation, engine_force: float, steering_angle: float):
    # Convert force to kg*units/timestep^2
    engine_force = engine_force*(hz/upm**2)

    state, colliding, checkpoint, finish = sim.step(engine_force, steering_angle)
    state = np.array(state)
    # Convert units
    state[0] = state[0]/upm # x
    state[1] = state[1]/upm # z
    state[2] = state[2]*(hz/upm) # v
    # Remap phi to [pi, -pi)
    phi = state[3]

    if phi > np.pi:
        state[3] = phi - 2*np.pi
```

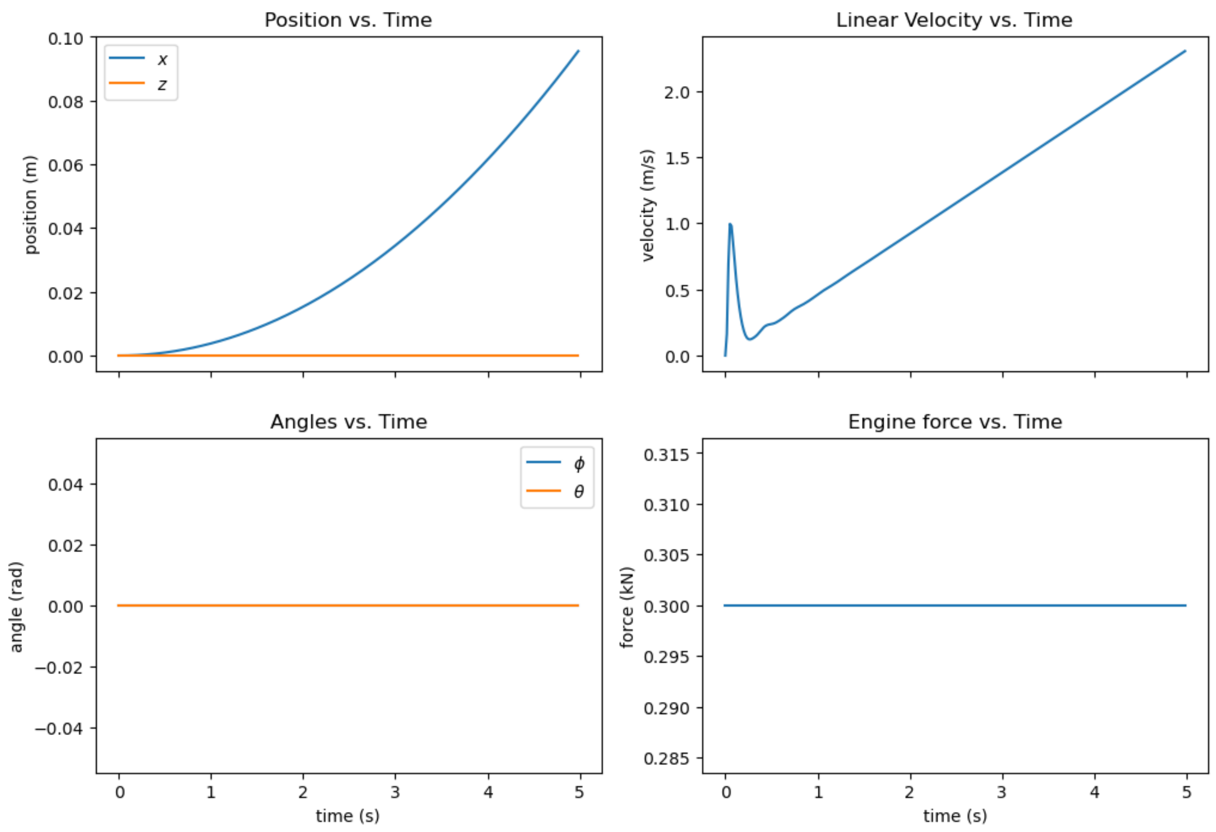
```python
    return state, colliding, checkpoint, finish


def generate_training_data(sim: CarSimulation, rotation: float, nsamples, nstates,
    sim.reset_car(rotation)

    data = np.zeros((nsamples, nstates))
    # x, z, v, phi
    x0 = np.array([0.0, 0.0, 0.0, rotation])

    for n in range(nsamples):
        engine_force = inputs[n, 0]
        steering_angle = inputs[n, 1]
        state, _, _, _ = simulate(sim, engine_force, steering_angle)
        data[n, :] = x0
        x0 = state

    return data

nsamples = hz*5 # 20 seconds worth of sample data
trajectories = []
trajectories_inputs = []

rotations = np.linspace(0, 2*np.pi, 4)
Fe_samples = [
    300 * np.ones(nsamples),
    1500 * np.ones(nsamples),
    np.linspace(0, 600, nsamples),
    np.linspace(600, 0, nsamples),
    np.linspace(-600, 600, nsamples),
    np.linspace(600, -600, nsamples),
]
Theta_samples = [
    0 * np.ones(nsamples),
    0.35*np.ones(nsamples),
    -0.35*np.ones(nsamples),
    np.concatenate(
        (
            0.35*np.ones(int(nsamples/4)),
            np.zeros(3*(int(nsamples/4)))
        )
    ),
    np.concatenate(
        (
            -0.35*np.ones(int(nsamples/4)),
            np.zeros(3*(int(nsamples/4)))
        )
    ),
    np.linspace(0.1, -0.1, nsamples),
    np.linspace(-0.1, 0.1, nsamples)
]

for tseconds in range(len(rotations)):
    for j in range(len(Fe_samples)):
        for k in range(len(Theta_samples)):
            rotation = rotations[tseconds]
            Fe = Fe_samples[j]
            theta = Theta_samples[k]
            inputs = np.array([Fe, theta]).T
            trajectory = generate_training_data(sim, rotation, nsamples, nstates, i

            trajectories.append(trajectory)
            trajectories_inputs.append(inputs)

print(f"Generated a total of {len(trajectories)} sample trajectories.")
```

```
Generated a total of 168 sample trajectories.
```

To visualize the system dynamics, lets plot some of the test data for analysis, starting with the very first trajectory.

```
In [4]: def plot_data(t, trajectory, trajectory_input):
    x = trajectory[:, 0]
    z = trajectory[:, 1]
    v = trajectory[:, 2]
    phi = trajectory[:, 3]
    Fe = trajectory_input[:, 0]
    Fe = Fe/1000 # Convert to kN
    theta = trajectory_input[:, 1]

    fig, axs = plt.subplots(2, 2, sharex=True, figsize = (12, 8))
    axs = axs.flatten()

    axs[0].plot(t, x, label='$x$')
    axs[0].plot(t, z, label='$z$')
    axs[0].set_title("Position vs. Time")
    axs[0].set_ylabel("position (m)")
    axs[0].legend()

    axs[1].plot(t, v)
    axs[1].set_title("Linear Velocity vs. Time")
    axs[1].set_ylabel("velocity (m/s)")

    axs[2].plot(t, phi, label=r'$\phi$')
    axs[2].plot(t, theta, label=r"$\theta$")
    axs[2].set_title("Angles vs. Time")
    axs[2].set_xlabel("time (s)")
    axs[2].set_ylabel("angle (rad)")
    axs[2].legend()

    axs[3].plot(t, Fe)
    axs[3].set_title("Engine force vs. Time")
    axs[3].set_xlabel("time (s)")
    axs[3].set_ylabel("force (kN)")



tseconds = np.linspace(0, nsamples - 1, nsamples)/hz

plot_data(tseconds, trajectories[0], trajectories_inputs[0])
```

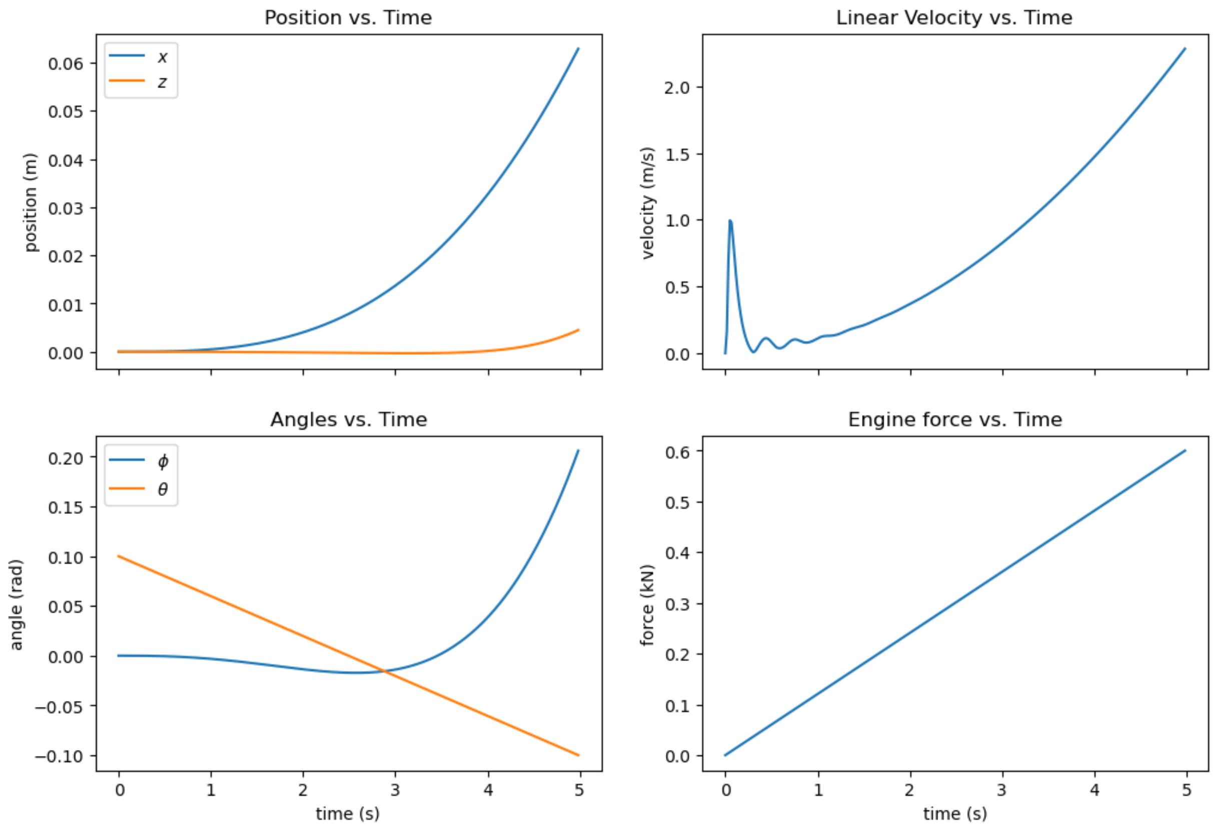
```
plt.show()
```



As expected, holding the engine force steady results in an increasing forward velocity and position, with the position increasing exponentially. What's not expected is the car being rotated slightly, likely a floating point error. I attempted to fix this by adjusting the height the car is created at in the simulation code to no avail.

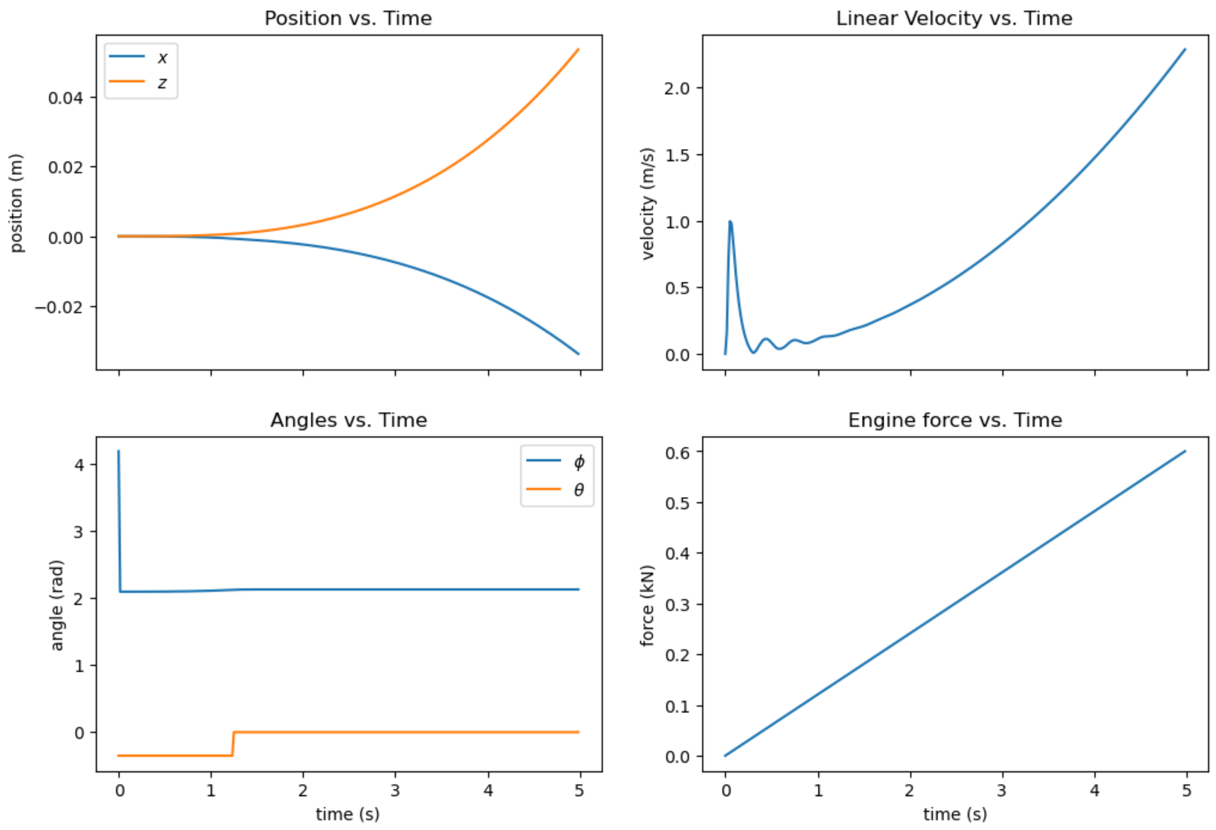
Let's plot a later data set and see if there are more interesting results.

In [5]:
```
plot_data(tseconds, trajectories[19], trajectories_inputs[19])
```



A higher engine force to start with naturally leads to faster oscillations in the position and velocity responses. Now, let's make one final plot.

```
In [6]: plot_data(tseconds, trajectories[102], trajectories_inputs[102])
```

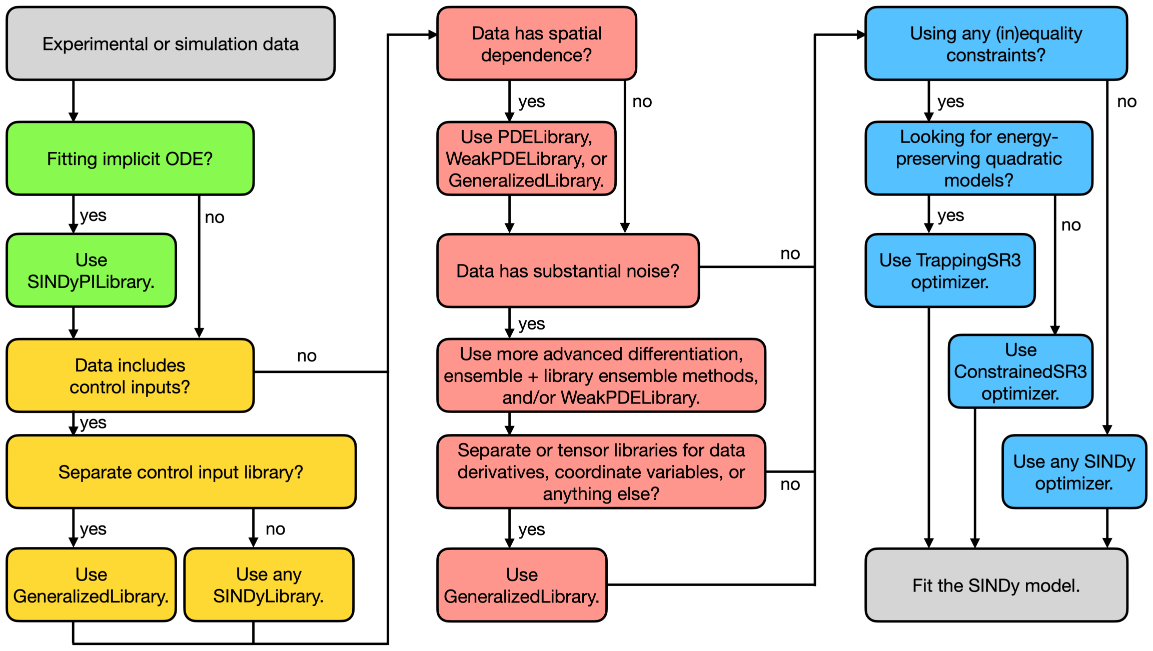


It is interesting how holding the engine force steady while changing the steering angle leads to oscillations in the state responses becoming a higher frequency.

1.2 Modeling with pySINDy

This data will now be used to fit a SINDy model;

Fortunately, the pySINDy Github repo provides a handy flowchart for choosing the right function library/libraries and optimizer for fitting the SINDy model to the measurement/test data.


Experimental or simulation data
Fitting implicit ODE?
yes
no
Use SINDyPILibrary.
Data includes control inputs?
no
yes
Separate control input library?
yes
no
Use GeneralizedLibrary.
Use any SINDyLibrary.
Data has spatial dependence?
yes
no
Use PDELibrary, WeakPDELibrary, or GeneralizedLibrary.
Data has substantial noise?
no
yes
Use more advanced differentiation, ensemble + library ensemble methods, and/or WeakPDELibrary.
Separate or tensor libraries for data derivatives, coordinate variables, or anything else?
no
yes
Use GeneralizedLibrary.
Using any (in)equality constraints?
yes
no
Looking for energy-preserving quadratic models?
yes
no
Use TrappingSR3 optimizer.
Use ConstrainedSR3 optimizer.
Use any SINDy optimizer.
Fit the SINDy model.

Since we are not solving for an implicit ODE (as far as I'm aware), the data has control inputs, is not spatially dependent and does not have significant noise, the flowchart says to use any pySINDy library. Since there are no inequality constraints in the system (as far as I'm aware), we can also use any pySINDy optimizer.

The default library is the `PolynomialLibrary`, which contains a variety of polynomial functions (to the second degree by default). PySINDy also provides a `CustomLibrary`, which will allow us to insert our own functions - we will use this to create a feature library with `sin` and `cos`, then combine with a `PolynomialLibrary`.

In [7]:
```
from pysindy.feature_library import *

functions = [
    lambda a: np.sin(a),
    lambda a: np.cos(a),
    lambda a, b: a*np.sin(b),
    lambda a, b: a*np.cos(b),
    ]
function_names = [
    lambda a: f"sin({a})",
    lambda a: f"cos({a})",
    lambda a, b: f"{a} sin({b})",
    lambda a, b: f"{a} cos({b})",
    ]
lib = PolynomialLibrary(degree=1) + CustomLibrary(functions, function_names)
model = pysindy.SINDy(
    feature_names=["x", "z", "v", "phi", "Fe", "theta"],
    feature_library=lib,
    optimizer=pysindy.optimizers.STLSQ(threshold=0.02)
    )

# Attempt to compensate for the jolting seen at the begining of the data
X = [x[60:, :] for x in trajectories]
U = [u[60:, :] for u in trajectories_inputs]
T = tseconds[60:]

model.fit(
    x=X,
    t=T,
    u=U,
    multiple_trajectories=True,
    )
print("pySINDy indentified the following dynamics:")
model.print()
```

```
pySINDy indentified the following dynamics:
(x)' = -8.257 x + 4.611 sin(x) + 1.318 z sin(theta) + 4.180 x cos(theta)
(z)' = -11.630 z + 6.970 sin(z) + 0.064 x sin(z) + -1.566 x sin(theta) + 5.228 z cos
(theta)
(v)' = 11.297 1 + 33.660 x + -7.792 v + 37.989 sin(x) + -0.022 sin(v) + -2.772 cos
(x) + -3.488 cos(z) + -0.298 cos(v) + -4.823 cos(theta) + -0.928 x sin(Fe) + -13.383
z sin(phi) + 2.012 z sin(theta) + -0.085 phi sin(theta) + -48.115 x cos(z) + 0.320 x
cos(v) + -13.161 x cos(phi) + -24.528 x cos(theta) + 8.415 v cos(theta)
(phi)' = -129.728 z + 0.080 phi + 20.054 theta + -123.211 sin(z) + -0.181 sin(phi) +
-20.691 sin(theta) + -21.565 x sin(z) + 7.029 x sin(phi) + -50.665 x sin(theta) + 2.
615 z sin(v) + 3.901 z sin(Fe) + -0.124 v sin(phi) + -0.046 phi sin(Fe) + -0.295 z c
os(v) + 3.027 z cos(phi) + -0.355 z cos(Fe) + 259.166 z cos(theta) + -0.010 phi cos
(theta)
```

Unfortunately, pySINDy's method of simulating SINDy models is rather slow. In an effort to remedy this, we will use the following function written by Dr. Rico Picone (dynamicslab/pysindy #358) to extract the dynamics into a callable function that runs much quicker.

In [8]:
```
def extract_sindy_dynamics(sindy_model, eps=1e-12):
    """Extract SINDy dynamics"""
    variables = sindy_model.feature_names  # e.g., ["x", "y", "z", "u"]
    coefficients = sindy_model.coefficients()
    features = sindy_model.get_feature_names()
        # e.g., ["1", "x", "y", "z", "u", "x * y", "x * z", "x * u", "y * z", ...]
    features = [f.replace("^", "**") for f in features]
    features = [f.replace(" ", " * ") for f in features]
    def rhs(coefficients, features):
        rhs = []
        for row in range(coefficients.shape[0]):
            rhs_row = ""
            for col in range(coefficients.shape[1]):
                if np.abs(coefficients[row, col]) > eps:
                    if rhs_row:
                        rhs_row += " + "
                    rhs_row += f"{coefficients[row, col]} * {features[col]}"
            rhs.append(rhs_row)
        return rhs
    rhs_str = rhs(coefficients, features)  # Eager evaluation
    n_equations = len(rhs_str)
    def sindy_dynamics(t, x_, u_, params={}):
        states_inputs = x_.tolist() + np.atleast_1d(u_).tolist()
        variables_dict = dict(zip(variables, states_inputs))
        # This modification is needed to work with our model as it uses trig functi
        variables_dict["sin"] = lambda a: np.sin(a)
        variables_dict["cos"] = lambda a: np.cos(a)
        variables_dict["tan"] = lambda a: np.tan(a)

        return [eval(rhs_str[i], variables_dict) for i in range(n_equations)]
    return sindy_dynamics
```

In [10]:
```
from scipy.integrate import solve_ivp

nsteps = hz*4
tsteps = np.linspace(0, nsteps, nsteps)
test_trajectory_inputs = np.array([
    np.linspace(250, 300, nsteps),
    np.concatenate((
        0.45*np.ones(int(nsteps/4)),
        np.linspace(0.45, -0.1, 3*int(nsteps/4))
    ))
]).T
test_trajectory = generate_training_data(sim, 0.0, nsteps, nstates, test_trajectory

# Now to generate a comparison trajectory
x0 = np.array([0.0, 0.0, 0.0, 0.0])
# model_trajectory = model.simulate(x0, t=tsteps, u=test_trajectory_inputs, integra
# model_trajectory = np.zeros((nsteps, nstates))
dynamics = extract_sindy_dynamics(model)

# for t in range(nsteps):
#     u = test_trajectory_inputs[t, :]
#     u = np.array(u)
#     x = dynamics(t, x0, u)

#     model_trajectory[t, :] = x0

#     x0 = np.array(x)

# def simulate_dynamics(t, x):
#     t = int(t)
#     u = test_trajectory_inputs[t, :]
#     return dynamics(t, x, u)

# soln = solve_ivp(simulate_dynamics, (0, nsteps-1), x0)
# model_trajectory = soln.y.T
# print(soln)

# print(model_trajectory[:240, :].shape)

# plot_data(tsteps, test_trajectory, test_trajectory_inputs)
# plot_data(tsteps, model_trajectory[:240, :], test_trajectory_inputs)
```