MuJoCo for Advanced Physics Simulation: From manipulators to autonomous vehicles

Duc Cuong Vu

Motion Control and Applied Robotics Laboratory
School of Electrical and Electronic Engineering,
Hanoi University of Science and Technology
Email: vdcuong2002@gmail.com
Site: dc-vu.github.io

Version: 1.0

Last updated: June 15, 2025

Motivation:

- High-performance physics simulation for complex, dynamic robotics.
- **MuJoCo**: lightweight, real-time, accurate multi-body dynamics with efficient contact handling.

Seminar Highlights:

- Introduction to MuJoCo and comparison with traditional simulators (e.g., MATLAB Simulink, Simscape).
- Live demos of:
  - 7-DOF Serial Manipulator (kinematics & control)
  - Stewart Platform (constraint handling & stabilization)
  - Autonomous 3D Robots (AUVs & UAVs: motion planning, optimization, real-time control)

Control Topics: System stabilization, real-time implementation, motion planning, optimization-based decision making.

Duration: 90 minutes

Date: xx/06/2025

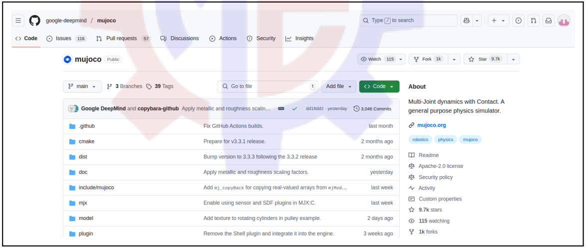
Table of Contents

- This seminar does not include mathematical equations.
- Programming skills are not covered in this seminar.

Table of Contents

What is MuJoCo?

"MuJoCo is a free and open source physics engine that aims to facilitate research and development in robotics, biomechanics, graphics and animation, and other areas where fast and accurate simulation is needed." [1], [2]



[1] https://mujoco.org/

[2] Todorov, E., Erez, T., & Tassa, Y. (2012, October). Mujoco: A physics engine for model-based control. In 2012 IEEE/RSJ international conference on intelligent robots and systems (pp. 5026-5033). IEEE.

Why MuJoCo?

- Fast, accurate simulation for complex multi-body robotic systems.
- Real-time capability for control and hardware-in-the-loop (HIL) testing.
- Efficient contact, constraint, and sensor modeling.
- Lightweight: easy integration with Python, C, or real hardware controllers.

Comparison

Comparisons of Simscape, MuJoCo and IsaacSim from 2015

from 2015 - present | from 2023 - present

IssacSim is the trend, but ...

Resources


Mobile Manipulators
Model
Preview
Google Robot
Hello Robot Stretch 2
Quadrupeds
Model
Preview
Unitree A1
Grippers & Hands
Model
Preview
Shadow E3M5
Robotiq 2F-85
Wonik Allegro
Arms
Back to top
Anybotics ANYmal C
Google Barkour v0
Google Barkour vB
Boston Dynamics Spot
Drones
Back to top
Model
Preview
Skydio X2
Bitcraze Crazyflie 2
Biomechanical
Model
Preview
Fruitfly
Arms
Model
Preview
Franka Emika Panda
Franka FR3
Universal Robots UR5e
Universal Robots UR10e
Bipeds
Model
Preview
Agility Cassie
Humanoids
Model
Preview
Unitree G1
Unitree H1

Model description

MuJoCo can load XML model files in its native MJCF format, as well as in the popular but more limited URDF format.

```
<mujoco>
  <default class="main">
    <geom rgba="1 0 0 1"/>
    <default class="sub">
      <geom rgba="0 1 0 1"/>
    </default>
  </default>

  <worldbody>
    <geom type="box"/>
    <body childclass="sub">
      <geom type="ellipsoid"/>
      <geom type="sphere" rgba="0 0 1 1"/>
      <geom type="cylinder" class="main"/>
    </body>
  </worldbody>
</mujoco>
```

Programming

MuJoCo has a C API and is intended for researchers and developers. The runtime simulation module is tuned to maximize performance and operates on low-level data structures that are preallocated by the built-in XML compiler.

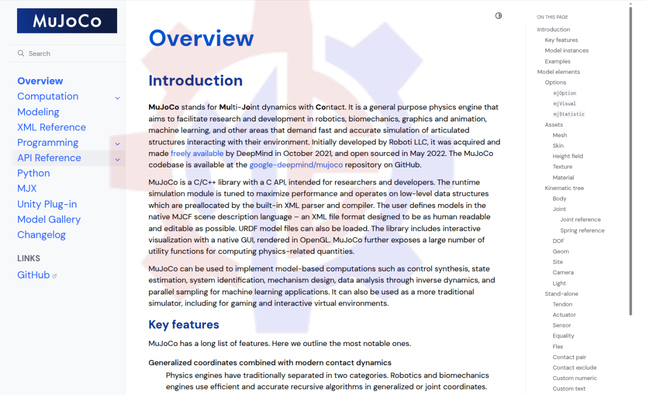
The Python bindings are distributed as the `mujoco` package on PyPI. These are low-level bindings that are meant to give as close to a direct access to the MuJoCo library as possible.

scripts > validate_kinematics.py > ...

```python
import mujoco as mj
from mujoco.glfw import glfw
import numpy as np
import os
from scipy.spatial.transform import Rotation

class MujocoSim:

    def __init__(self, xml_name, time_step, simulation_time, fps):

        #get the full path
        dirname = os.path.dirname(__file__)
        abspath = os.path.join(dirname + "/" + xml_name)

        # MuJoCo data structures
        self.model = mj.MjModel.from_xml_path(abspath)  # MuJoCo model
        if time_step is not None:
            self.model.opt.timestep = time_step
        self.data = mj.MjData(self.model)                # MuJoCo data
        self.cam = mj.MjvCamera()                        # Abstract camera
        self.opt = mj.MjvOption()                        # visualization options

        self.xml_path = abspath
        self.simulation_time = simulation_time
        self.fps = fps

        # Print camera config
        self.print_camera_config = False #set to True to print camera config
                                         #this is useful for initializing view of the model)
```

Learning

The documents about MuJoCo could be found at [3].



MuJoCo

Search

Overview
Computation
Modeling
XML Reference
Programming
API Reference
Python
MJX
Unity Plug-in
Model Gallery
Changelog

LINKS

GitHub

Overview

Introduction

MuJoCo stands for **Mu**lti-**Jo**int dynamics with **Co**ntact. It is a general purpose physics engine that aims to facilitate research and development in robotics, biomechanics, graphics and animation, machine learning, and other areas that demand fast and accurate simulation of articulated structures interacting with their environment. Initially developed by Roboti LLC, it was acquired and made freely available by DeepMind in October 2021, and open sourced in May 2022. The MuJoCo codebase is available at the google-deepmind/mujoco repository on GitHub.

MuJoCo is a C/C++ library with a C API, intended for researchers and developers. The runtime simulation module is tuned to maximize performance and operates on low-level data structures which are preallocated by the built-in XML parser and compiler. The user defines models in the native MJCF scene description language – an XML file format designed to be as human readable and editable as possible. URDF model files can also be loaded. The library includes interactive visualization with a native GUI, rendered in OpenGL. MuJoCo further exposes a large number of utility functions for computing physics-related quantities.

MuJoCo can be used to implement model-based computations such as control synthesis, state estimation, system identification, mechanism design, data analysis through inverse dynamics, and parallel sampling for machine learning applications. It can also be used as a more traditional simulator, including for gaming and interactive virtual environments.

Key features

MuJoCo has a long list of features. Here we outline the most notable ones.

Generalized coordinates combined with modern contact dynamics

Physics engines have traditionally separated in two categories. Robotics and biomechanics engines use efficient and accurate recursive algorithms in generalized or joint coordinates.

ON THIS PAGE

[3] https://mujoco.readthedocs.io/en/stable/overview.html

Table of Contents

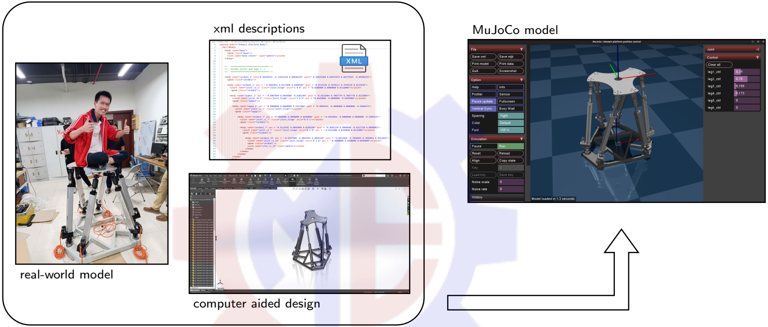
MATLAB Simscape multi-body traditional approach

- First step, install the 50GB of MATLAB (include 20GB for installer and 30GB for unzip files.
- Secondly, what if you don't have the budget to purchase a license???
- If you are a rich kid, in the next step, you spend 1 minute to 5 minutes for starting the MATLAB, and about more than one minutes for open the first Simulink windows.
- Finally, designing is quite easy, but ..., compile time and simulation time are significant! If a block of Matlab Function is modified, MATLAB re-compiles the simulation.

Installation

```
pip install mujoco
```

Configuration

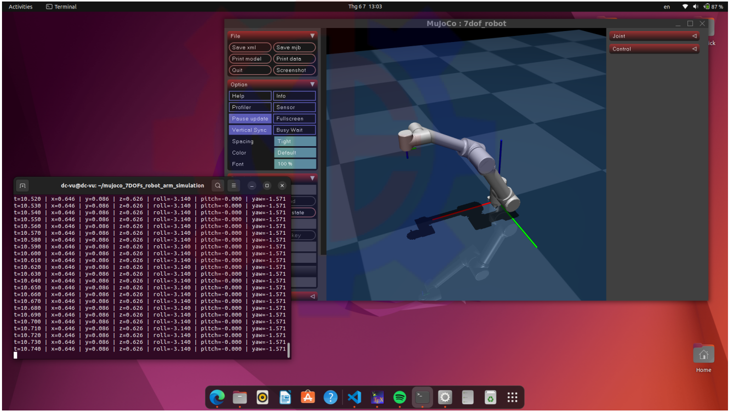


Process of replicating a real-world Stewart platform model into a MuJoCo simulation environment: Starting from the real-world model, the system is first designed in a computer-aided design (CAD) software (SolidWorks, Inventor). The CAD design is then translated into XML descriptions compatible with MuJoCo. Finally, the structured XML is used to construct the MuJoCo model with the actuators, sensors, ..., enabling high-fidelity physics-based simulation and control testing.

Table of Contents

7-DOF serial manipulator

7-DOF serial manipulator simulation in MuJoCo, the open-source is available at [4] (Contributor: Duc-Cuong Vu)



[4] https://github.com/dc-vu/mujoco_7DOFs_robot_arm_simulation.git

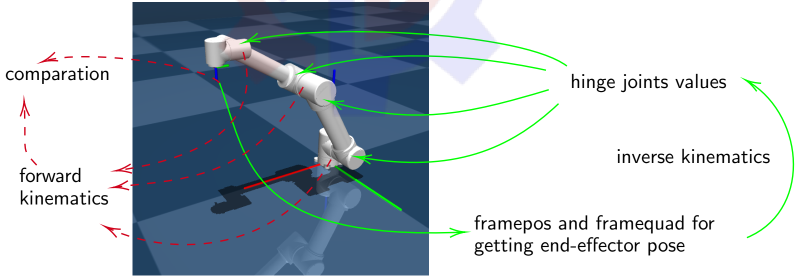
Robot definition

Robot XML definition

```xml
<asset>
  <mesh name="base_mesh" file="base.stl" scale="0.001 0.001 0.001"/>
  <mesh name="motor110_mesh" file="motor110.stl" scale="0.001 0.001 0.001"/>
  <mesh name="motor70_mesh" file="motor70.stl" scale="0.001 0.001 0.001"/>
  <mesh name="link2_mesh" file="link2.stl" scale="0.001 0.001 0.001"/>
  <mesh name="link3_mesh" file="link3.stl" scale="0.001 0.001 0.001"/>
</asset>


<!-- Link 1 -->
  <body name="link1">
    <joint name="joint1" type="hinge" axis="0 0 1"/>
    <geom type="mesh" mesh="motor110_mesh"/>

    <!-- Link 2 -->
    <body name="link2">
      <joint name="joint2" type="hinge" axis="0 0 1" />
      <geom type="mesh" mesh="motor110_mesh"/>

      <!-- Robot tree-based definition => easy -->

<!-- Sensing -->
<sensor>
  <framepos name="ee_pos" objtype="body" objname="ee"/>
  <framequat name="ee_quat" objtype="body" objname="ee"/>
</sensor>
```

Inverse kinematics validation

```python
ee_pos = data.body("ee").xpos
eex, eey, eez = ee_pos
ee_xmat_flat = data.body("ee").xmat
ee_R = np.array(ee_xmat_flat).reshape(3, 3)
ori = R.from_matrix(ee_R).as_euler('xyz')
print(f"t={t:.3f} | x={eex:.3f} | y={eey:.3f} | z={eez:.3f} | "
        f"roll={ori[0]:.3f} | pitch={ori[1]:.3f} | yaw={ori[2]:.3f}")
# Inverse kinematics calculation here => do not straightforward
for i in range(7):
        data.joint(f"joint{i+1}").qpos = joint_demand[i] - angles_offset[i]
```



.

- The inverse kinematics is not real-time, because an optimization problem is used for solving the joint states

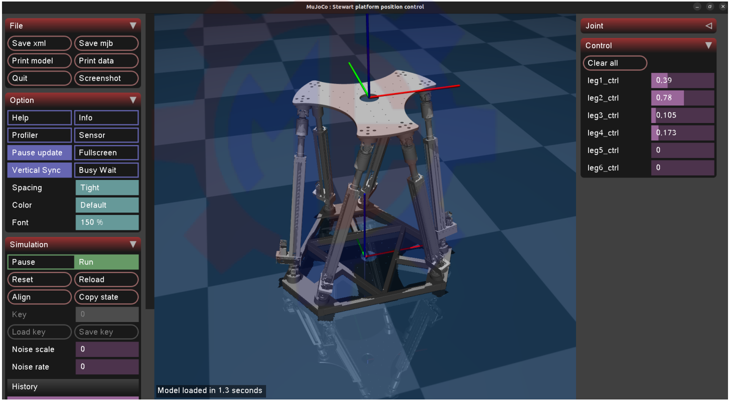
```python
def levenberg_marquardt_ik_dh(q_init, desired_pose):
  result = least_squares(
  ik_loss,           # the coss  function
  q_init,           # initial guess
  args=(desired_pose,),
  method='trf',  # Change to 'lm' if you want to use Levenberg-Marquardt
  ftol=1e-6,     # Tolerance for the cost function
  xtol=1e-6,     # Tolerance for the solution
  gtol=1e-6,     # Tolerance for the gradient
  max_nfev=200  # Maximum number of function evaluations
  )
  return result.x, result.cost
```

This function is implemented in Python, the solver is trust region method or Levenberg-Marquardt. Thus the real-time purpose is not guaranteed. This problem could be addressed by utilizing the C code in the remaining part of this talk.

- Need actuators

Stewart platform simulation in MuJoCo, the open-source is not available.

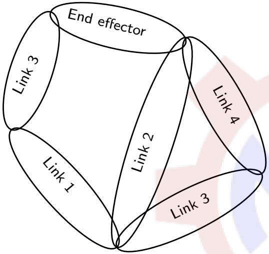
(Contributors: Viet Khanh Nguyen, Duc Cuong Vu)



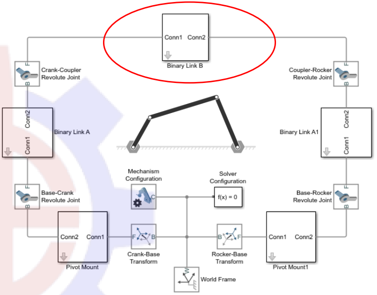
Closed loop kinematics chain

"Which ensures that the connection is valid?

closed loop kinematic chain



Inertial coordinate



The parallel mechanisms could be described in MuJoCo by equality attribute as follows

```
<mujoco model="Parallel mechanisms equality">
  <equality>
    <weld name="w_1" site1="site1" site2="site2"></weld>
    <weld name="w_2" site1="site2" site2="site4"></weld>
  </equality>
</mujoco>
```

Optimization-based kinematics - (ctypes in Python - C code for real-time implementation)

The algorithm is introduced in *2.2. Kinematics analysis* of the publication [5]

kinematics > forward_kinematics.py > ...

```python
import ctypes
import numpy as np
import time

lib = ctypes.cdll.LoadLibrary('kinematics/c_code/kinematics.so')

lib.lm_optimize.argtypes = [
    ctypes.POINTER(ctypes.c_float),             # float* x
    ctypes.POINTER(ctypes.c_float),             # const float l_target[6]
    ctypes.c_bool,                               # const bool print
    ctypes.POINTER(ctypes.c_int)                # int* num_iter
]
lib.lm_optimize.restype = None

x_array = (ctypes.c_float * 6)(0, 0, 1.1, 0, 0, 0)  # initial guess
# l_target = (ctypes.c_float * 6)(0.9287322759628296, 0.9264336228370667, 0.9265220761299133, 0.9288733601570129, 0.9278072714805603, 0.9
# [1.05833114 0.92127699 0.92419712 0.92959993 0.92540508 0.92480616]
l_target = (ctypes.c_float * 6)(1.05833114, 0.92127699, 0.92419712, 0.92959993, 0.92540508, 0.92480616)  # target lengths

num_iter = ctypes.c_int(0)

start_time = time.time()
lib.lm_optimize(x_array, l_target, False, ctypes.byref(num_iter))
end_time = time.time()

x_result = [x_array[i] for i in range(6)]
print("Optimized pose (x):", x_result)
print("Number of iterations:", num_iter.value)
print(f"Execution time: {(end_time - start_time)*1e6:.2f} microseconds")
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
(.venv) dc-vu@dc-vu:~/stewart_platform$ /home/dc-vu/stewart_platform/.venv/bin/python /home/dc-vu/stewart_platform/kinematics/forward_kinematics.py
Optimized pose (x): [-0.050542283803224564, 0.1859666407108307, 1.0930392742156982, 0.07483875751495361, -0.05862480774521828, 0.26324185729026794]
Number of iterations: 2
Execution time: 26.23 microseconds
(.venv) dc-vu@dc-vu:~/stewart_platform$
```

[5] Vu, D. C., Nguyen, T. L., & Nguyen, D. H. (2025). A novel approach of Consensus-based Finite-time Distributed Sliding Mode Control for Stewart platform manipulators motion tracking. Results in Engineering, 25, 103872.

Dynamics control

Control purpose → **The upper platform track a reference**

From the desired pose of the upper platform, by the inverse kinematics, the desired length of the actuators is calculated.

In the aim of this seminar, a simple PID controller is design to control the length of the actuators.

```python
# Get data from sensors ...
sensor_value = np.array([
  data.sensor('leg1_len_sensor').data,
  data.sensor("leg2_len_sensor").data,
  ...
  ]).reshape(-1) + self.legs_offset

# Define the controller (PID in this case) ...
self.controller = StewartPlatformPIDController(Kp, Ki, Kd, self.time_step )

# Apply the control signal to the model via data.ctrl
data.ctrl = self.controller.solve(errors)
```

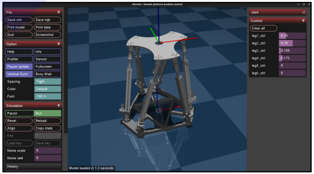
Peripherals connection

Get pose of the joystick as the references values of the upper platform.

```python
import pygame
pygame.init()
pygame.joystick.init()

roll = normalize_euler_workingspace(round_to_step(self.joystick.get_axis(0)))
pitch = -normalize_euler_workingspace(round_to_step(self.joystick.get_axis(1)))
yaw = -normalize_euler_workingspace(round_to_step(self.joystick.get_axis(5)))

z = normalize_pos_workingspace(round_to_step(self.joystick.get_axis(2)))
y = normalize_pos_workingspace(round_to_step(self.joystick.get_axis(3)))
x = normalize_pos_workingspace(round_to_step(self.joystick.get_axis(4)))
```

read data from flight joystick

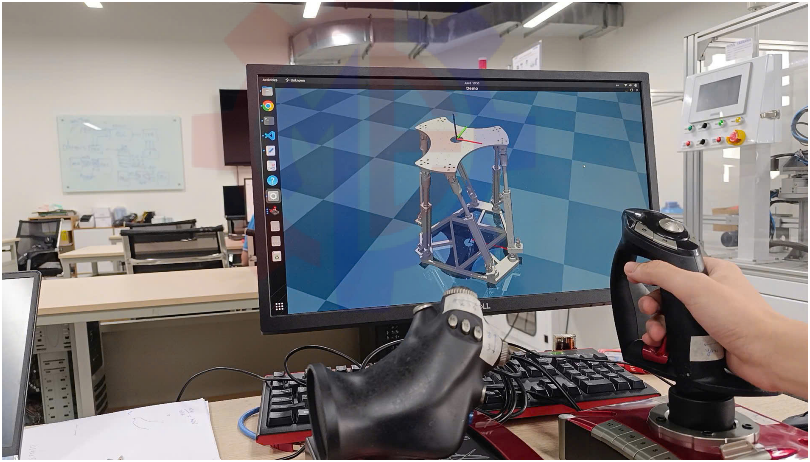
evdev
pygame



Peripherals connection

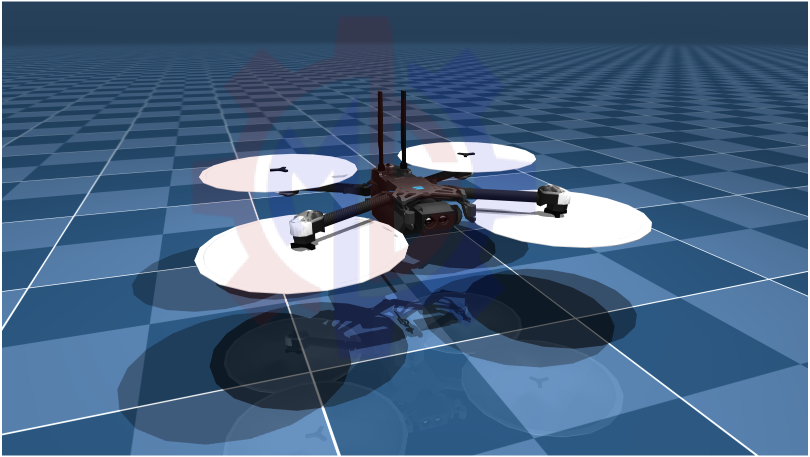
The source code for this project is not publicly available.

For access or further information, please contact Viet Khanh Nguyen or Duc Cuong Vu.



Skydio X2 model

The model is available at [6]



[6] texthttps://github.com/google-deepmind/mujoco_menagerie/blob/main/skydio_x2/x2.xml

Skydio X2 model

The AUV is a standard model. Actuator allocation can be found in the <actuator> tag below.

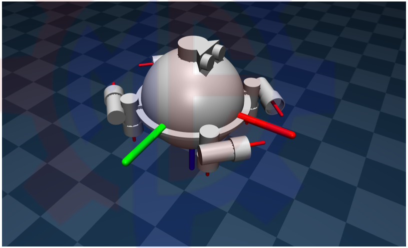
In addition, an IMU sensor is used in this model.

```
<actuator>
  <motor class="x2" name="thrust1" site="thrust1" gear="0 0 1 0 0 -.0201"/>
  <motor class="x2" name="thrust2" site="thrust2" gear="0 0 1 0 0  .0201"/>
  <motor class="x2" name="thrust3" site="thrust3" gear="0 0 1 0 0  .0201"/>
  <motor class="x2" name="thrust4" site="thrust4" gear="0 0 1 0 0 -.0201"/>
</actuator>

<sensor>
  <gyro name="body_gyro" site="imu"/>
  <accelerometer name="body_linacc" site="imu"/>
  <framequat name="body_quat" objtype="site" objname="imu"/>
</sensor>
```

The source code for this project is not publicly available.

For access or further information, please contact Viet Khanh Nguyen.

This repository presents a simulation framework for an Autonomous Underwater Vehicle (AUV) that combines Model Predictive Control (MPC), Control Barrier Functions (CBF) for robust path tracking and obstacle avoidance in dynamic underwater environments. The theories are shown in the publication [7]



The source code for this project is not publicly available.

For access or further information, please contact Duc Cuong Vu.

[7] Pham, M. D., Vu, D. C., Nguyen, T. T. H., Nguyen, T. V. A., Vu, M. N., & Nguyen, T. L. (2025). CBFs-based Model Predictive Control for Obstacle Avoidance with Tilt Angle Limitation for Ball-Balancing Robots. IEEE Access.

ODIN actuators

The ODIN is driven by 8 thruster, it could be defined as follows

```
<actuator>
  <general name = "thruster1" site="thruster1_site" gear="0 0 1 0 0 0"/>
  <general name = "thruster2" site="thruster2_site" gear="0 0 1 0 0 0" />
  <general name = "thruster3" site="thruster3_site" gear="0 0 1 0 0 0" />
  <general name = "thruster4" site="thruster4_site" gear="0 0 1 0 0 0" />
  <general name = "thruster5" site="thruster5_site" gear="0 0 1 0 0 0" />
  <general name = "thruster6" site="thruster6_site" gear="0 0 1 0 0 0" />
  <general name = "thruster7" site="thruster7_site" gear="0 0 1 0 0 0" />
  <general name = "thruster8" site="thruster8_site" gear="0 0 1 0 0 0" />
</actuator>
```

and the general force and moment of underwater environment

```
<actuator>
  <general name="Force_X" site="odin_site" gear="1 0 0 0 0 0"/>
  <general name="Force_Y" site="odin_site" gear="0 1 0 0 0 0"/>
  <general name="Force_Z" site="odin_site" gear="0 0 1 0 0 0"/>
  <general name="Moment_K" site="odin_site" gear="0 0 0 1 0 0"/>
  <general name="Moment_M" site="odin_site" gear="0 0 0 0 1 0"/>
  <general name="Moment_N" site="odin_site" gear="0 0 0 0 0 1"/>
</actuator>
```

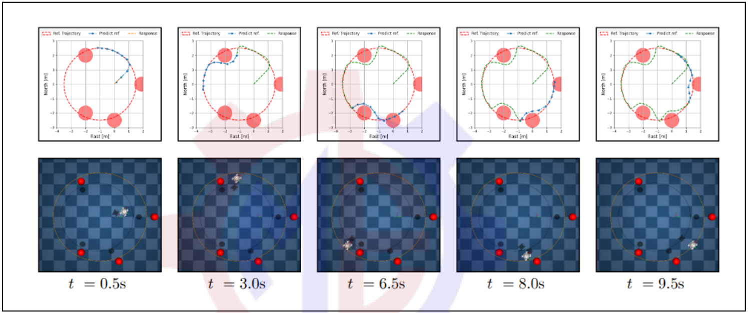
Following the SNAME(1950) notation, a general underwater vehicle could be controlled by consider the control input as the forces and moments at the origin. Thus, the force allocation for thruster could be achieved by an energy-optimization decision.

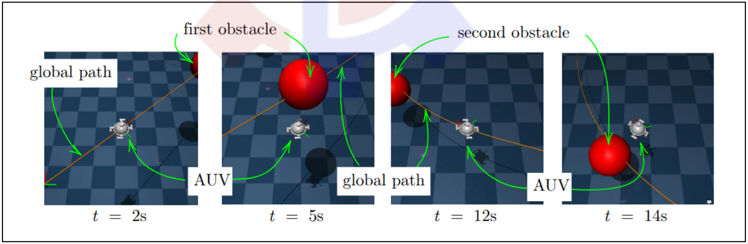
```
from scipy.optimize import minimize, Bounds

def objective(x):  # Objective: minimize x^T x
        return np.dot(x, x)

def eq_constraint(x):  # Equality constraint: E x = A
        return force_allocation(x) - FandM

bounds = Bounds([0, 0, 0, 0, -np.inf, -np.inf, -np.inf, -np.inf], [np.inf]*8)
linear_constraint = {'type': 'eq', 'fun': eq_constraint}

x0 = thurster_prev   # Initial guess

result = minimize(objective, x0, method='SLSQP',
  constraints=[linear_constraint],
  bounds=bounds)
```

Model Predictive Control with CasADi

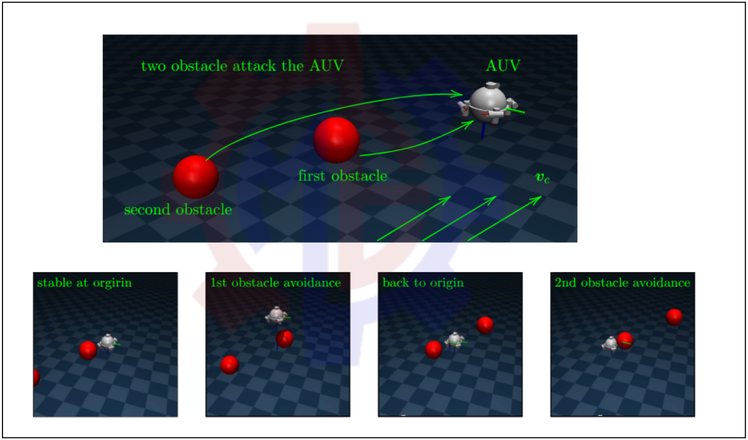
```python
import casadi as ca
for j in range(N):
  # Apply dynamics to get next state
  x_next = rk4_step(x[:, j], u[:, j], dtc)
  opti.subject_to(x[:, j+1] == x_next)  # Dynamics constraint

  # Tracking error: difference between current position and reference
  e = x[0:3, j] - QR[:, j]
  # Add to cost (only position error considered)
  cost += ca.mtimes([e.T, Q, e])
  # Note: control effort term (u.T * R * u) can be added here

  dist = obs_detected * ((x_next[0] - x_obs)**2 +
    (x_next[1] - y_obs)**2 + (x_next[2] - z_obs)**2 - r_obs**2)

  # Update obstacle position based on velocity
  x_obs += current_vel[0]*dtc
  y_obs += current_vel[1]*dtc
  z_obs += current_vel[2]*dtc

  # Change in distance between time steps
  diff_dist = dist - dist_before
  dist_before = dist  # Update for next step

  # CBF constraint to ensure distance grows fast enough
  opti.subject_to(diff_dist >= -gamma*dist)
```

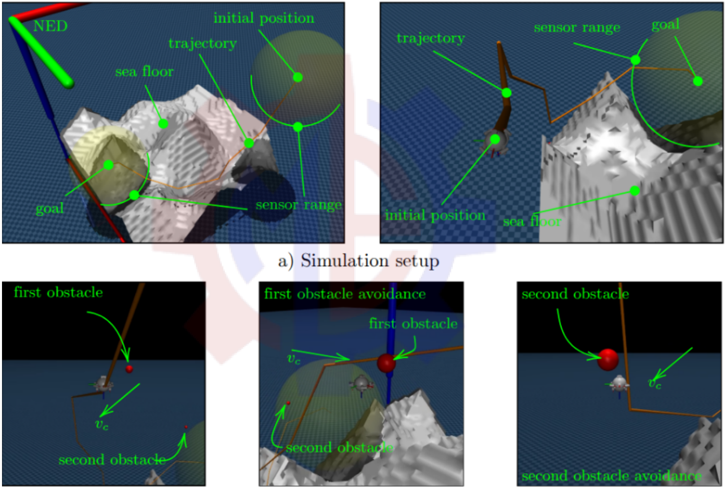
Model Predictive Control with CasADi





Model Predictive Control with CasADi





a) Simulation setup

b) Simulation results

Table of Contents

Thanks for your attention! Any questions?

Hope you slept comfortably!