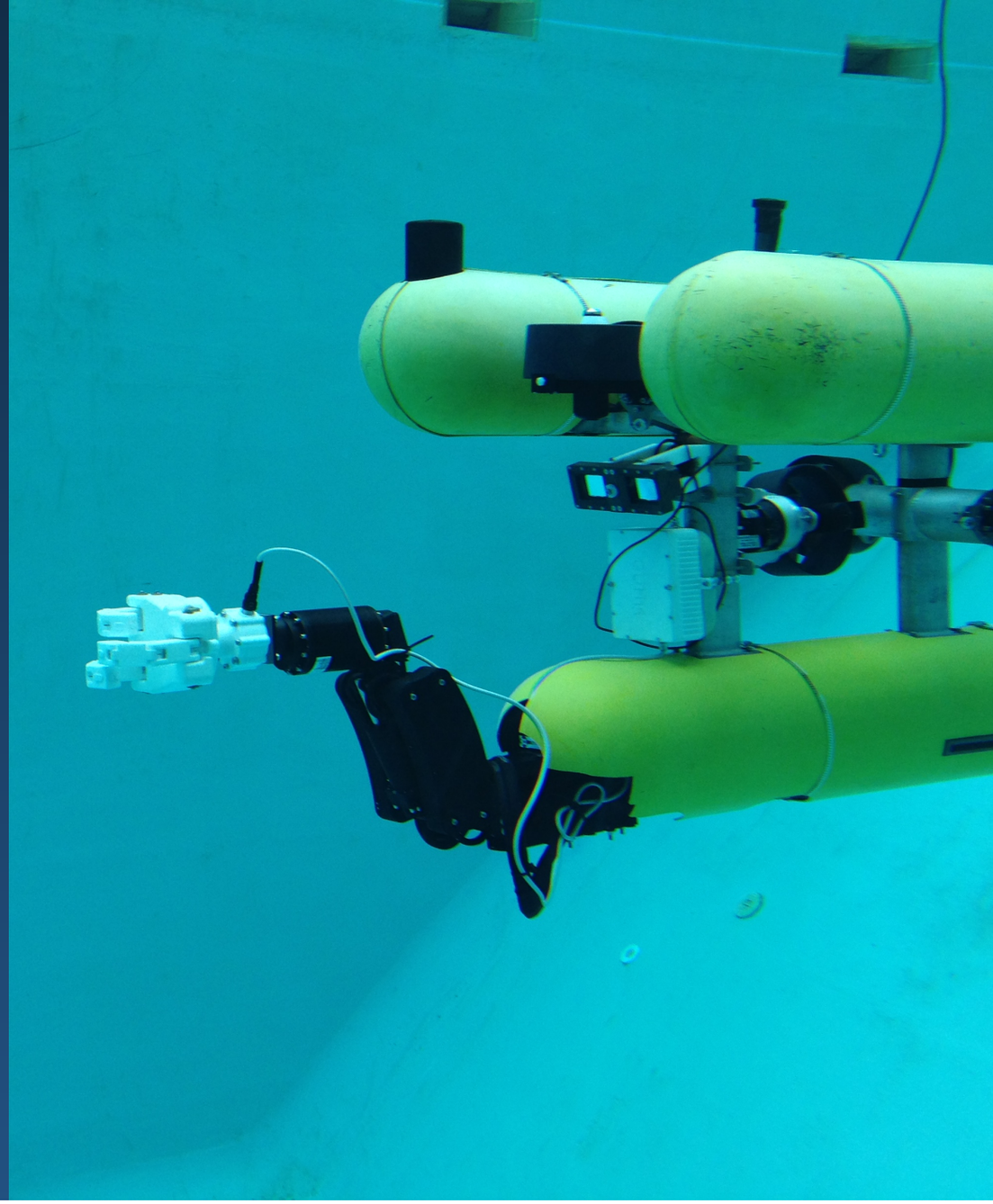
Universitat de Girona

IFRoS MIRS

HANDS-ON INTERVENTION:

Vehicle-Manipulator Systems



Patryk Cieślak

patryk.cieslak@udg.edu

Lecture 1: Introduction

Contents

1.1. Organization: Course contents

1. Introduction (1 lecture)

- 1.1. Vehicle-manipulator systems
- 1.2. Numpy
- 1.3. Matplotlib
- 1.4. Classes & inheritance

2. Resolved-rate motion control (1 lecture)

- 2.1. Differential kinematics
- 2.2. Numerical computation of Jacobian
- 2.3. Inverse kinematics
- 2.4. Singularities

3. Task-Priority control algorithm (2 lectures)

- 3.1. Redundancy
- 3.2. Null-space
- 3.3. Equality & inequality tasks
- 3.4. Recursive implementation
- 3.5. Mobile base
- 3.6. Practical extensions

4. Project (1 lecture)

- 4.1. Tasks
- 4.2. Deliverables
- 4.3. Goals for HOI course
- 4.4. Goals for integration with other courses

Organisation: 5 lectures, 5 lab sessions, theoretical exam, 7 weeks of project

1.2. Organization: Tools & experimental platform

Software

- **Visual Studio Code** - Python/C++/ROS programming IDE
- **Python** - theory understanding, simple simulation (Numpy, Matplotlib)
- **ROS** - robot architecture, controller development
- **Stonefish** - simulation tool, already prepared VMS model and scenarios
- **RVIZ** - visualisation of control algorithm operation



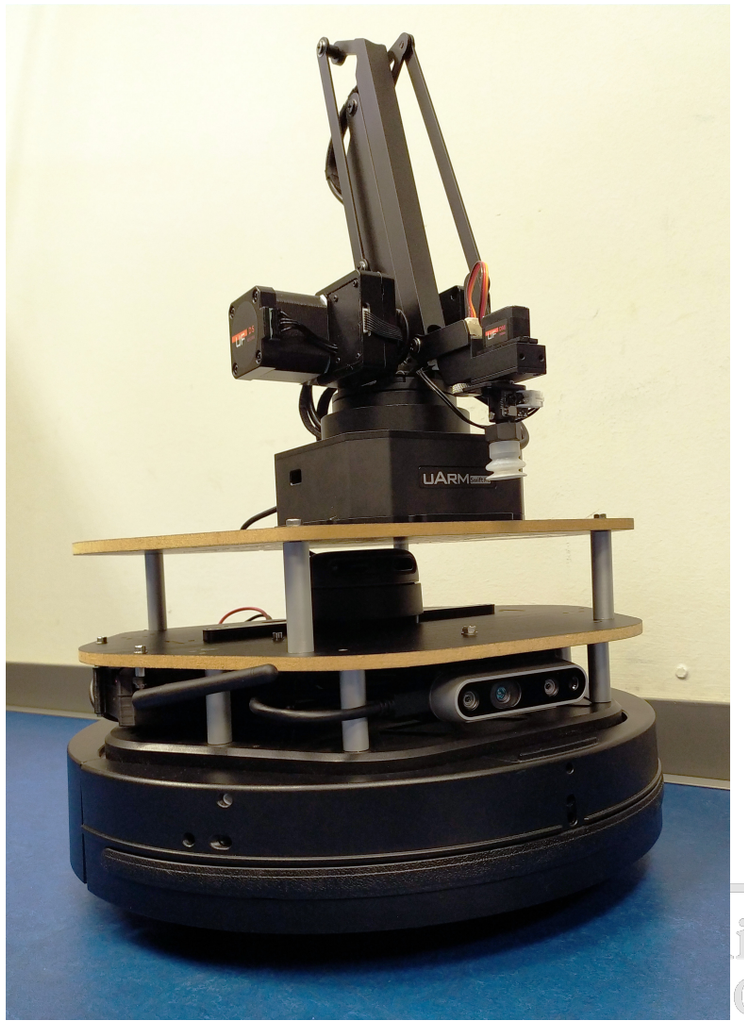


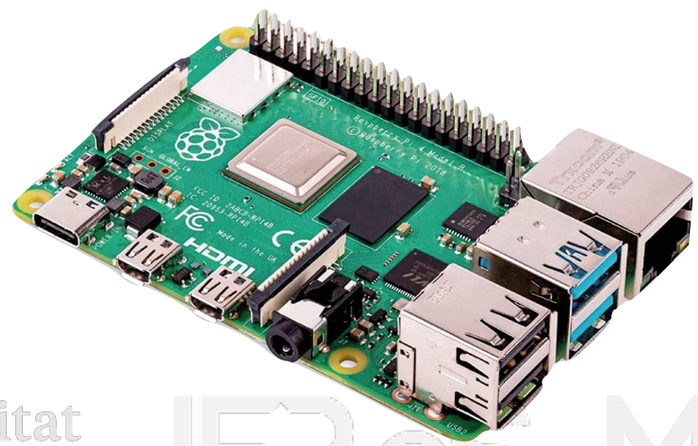


1.2. Organization: Tools & experimental platform

Hardware

- **Turtlebot 2** with vastly improved battery capacity
- **Raspberry Pi 4B** with WiFi antenna
- **uFactory uArm SwiftPro** 4DOF manipulator with a vacuum gripper (suction cup)
- **Intel RealSense D435i** (RGBD, colour & depth camera)
- **RPLidar A2** (rotating 2D Lidar)





1.3. Organization: Evaluation

1. Lab sessions

- Work in **groups of 2** people **during the lab session**, **reports** prepared **individually as homework.**
- **During the lab (group): implementation** of presented **theory** in the form of necessary **functions and structures**, which will result in a **functioning simulation of the given robot model.**
- **Homework (individual):** preparation of a **report**, including: **commented code** developed during the lab session, **answers to questions**, presentation of different types of **plots** to support the **discussion** about algorithm functioning, problems, etc.

2. Project

- Each project is developed by a **group of 3 people (the same group in all hands-on subjects).**
- **3 levels:**
  - Basic implementation of the TP algorithm on the experimental platform, allowing for teleoperation of the VMS by commanding end-effector pose with a gamepad.
  - Autonomous pick and place operation.
  - Pick and place operation with obstacle avoidance.
- **Combining control** implementation with developments related to **other hands-on subjects**.
- **Projects** will be **presented in a special session** with all hands-on subject teachers.

3. Mark

- To pass the subject it is a minimum requirement to pass the lab sessions, the exam, and the hands-on project.
- The final mark is composed of: **30%** lab sessions, **20%** exam, **30%** hands-on project, **20%** oral presentation of the project.

1.4. Organization: Reports

- Prepared **individually**.
- Only **PDF** format accepted (A4, portrait orientation) + Python code in ***.txt** files.
- **First page is a title page**, including:
    1. Title of the session.
    2. Date.
    3. Full name of the author.
    4. Full name of the person working with the author during the lab session.
- Following pages **numbered**.
- Contents according to the **session instructions**.

Late delivery penalty

$T \leq 24h \rightarrow -1$ pt

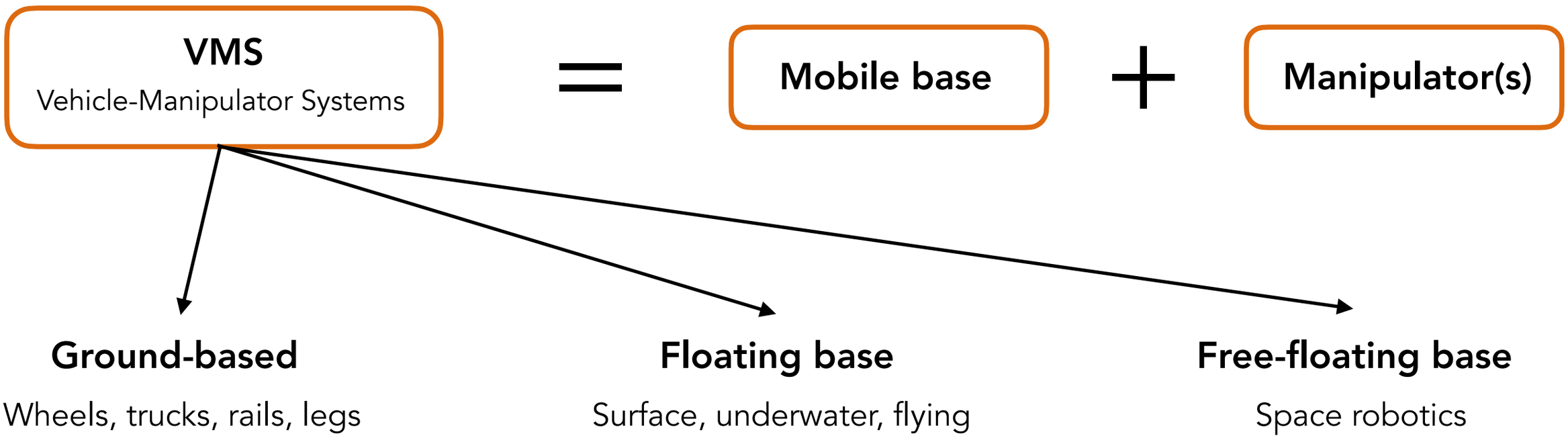
$24h < T \leq 48h \rightarrow -2$ pt

$48h < T \leq 72h \rightarrow -3$ pt

$T > 72h \rightarrow 0$ pt

- Questions answered **thoroughly**.
- All pieces of **code** properly **formatted** and **fully commented** (each line), with **clear variables meaning**.
- All **equations** written using **Latex or equation tool** embedded in the selected text editor.
- All **figures numbered**, with **clear captions**, **labeled axes**, **physical units** (SI), scaled to be easily **readable**.
- Report written in a **clear**, **understandable**, **technical** English.
- Deadline according to the teacher's requirements.
- **Mark components (points for each exercise according to the instructions):**
    1. Text, equations, figures readable and well formatted.
    2. Code is functional.
    3. Code well formatted and explained.
    4. Answers to questions.

2.1. Robotic intervention: Vehicle-manipulator systems



- The base can easily maintain fixed position (stop completely).
- The base position and velocity can be precisely controlled.
- The payload capacity is high.
- The base can often keep position without consuming energy.
- Navigation based on dead reckoning can give enough precision depending on environment.
- Walking robots are an exception: hard to control, not easy to measure motion.

- The base position has to be continuously corrected and never remains fixed.
- The base position and velocity is often not possible to control precisely due to environmental disturbances and complicated system dynamics.
- The payload capacity is severely limited.
- Navigation based on dead-reckoning is not sufficient and global positioning has to be used, combined with vision based approaches.
- Base instability can be corrected to some extent using a fast manipulator.

- The base position and orientation is coupled with the pose of the manipulator and the payload.
- Base orientation can be controlled using reaction wheels.
- Environmental conditions are easy to model and predict (space).
- Dynamics of the system can be modelled and controlled to the high degree.
- Control algorithms are complex.
- Experimental testing is only partially possible on Earth.

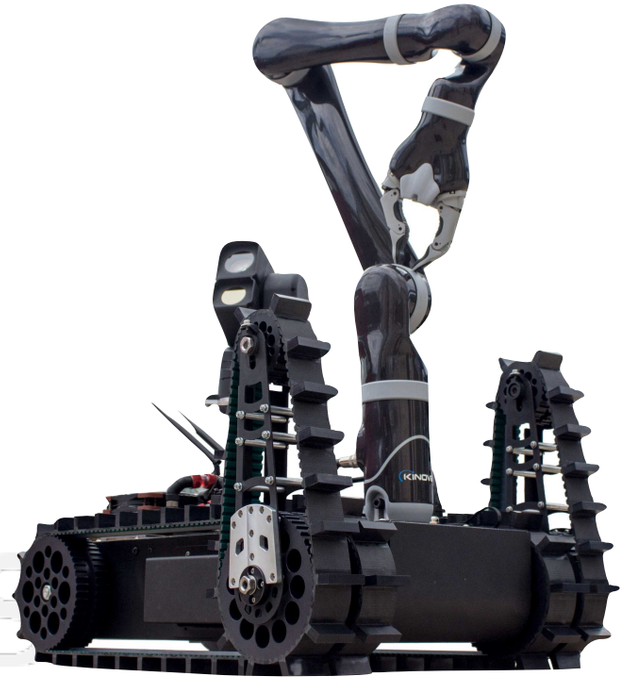
2.2. Robotic intervention: Ground-based VMS

KUKA KMR QUANTEC

KUKA KMR iiwa

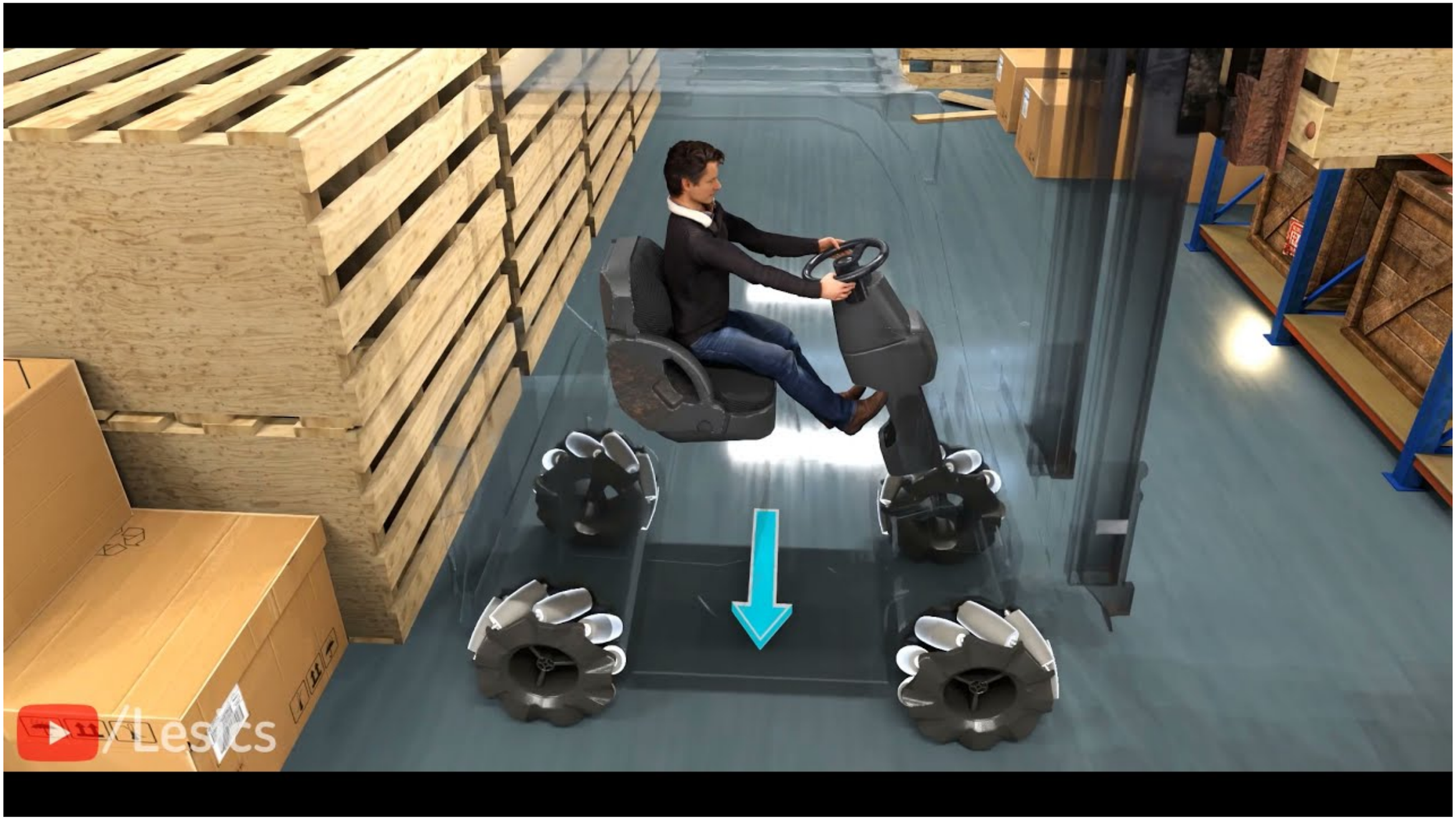
ROBOTNIK XL-GEN

ROBOTNIK Rising



2.2. Robotic intervention: Ground-based VMS

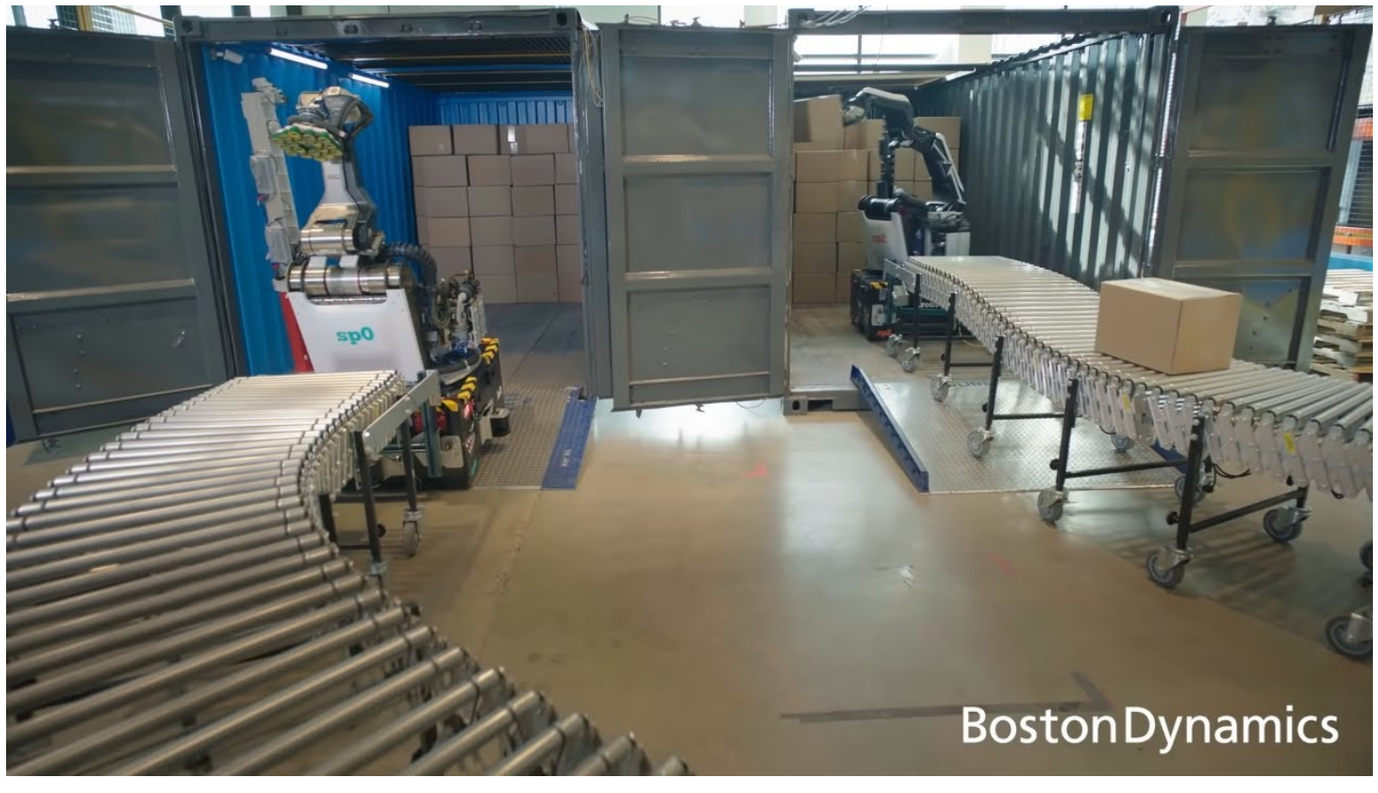
Mecanum wheels



2.2. Robotic intervention: Ground-based VMS

BionicHIVE SQUID

Boston Dynamics Stretch



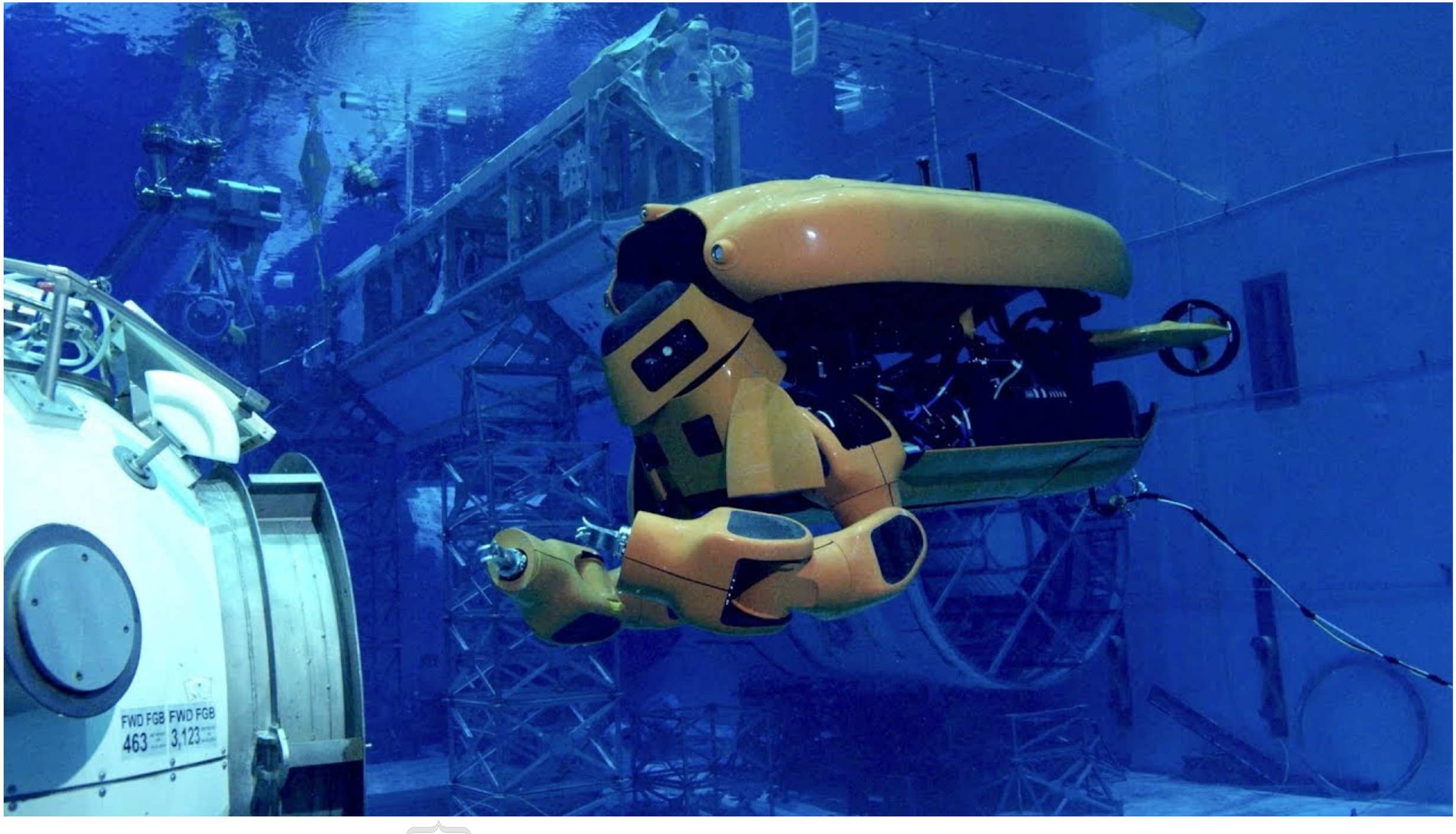
PAL Robotics TIAGo

2.2. Robotic intervention: Ground-based VMS

Boston Dynamics Spot

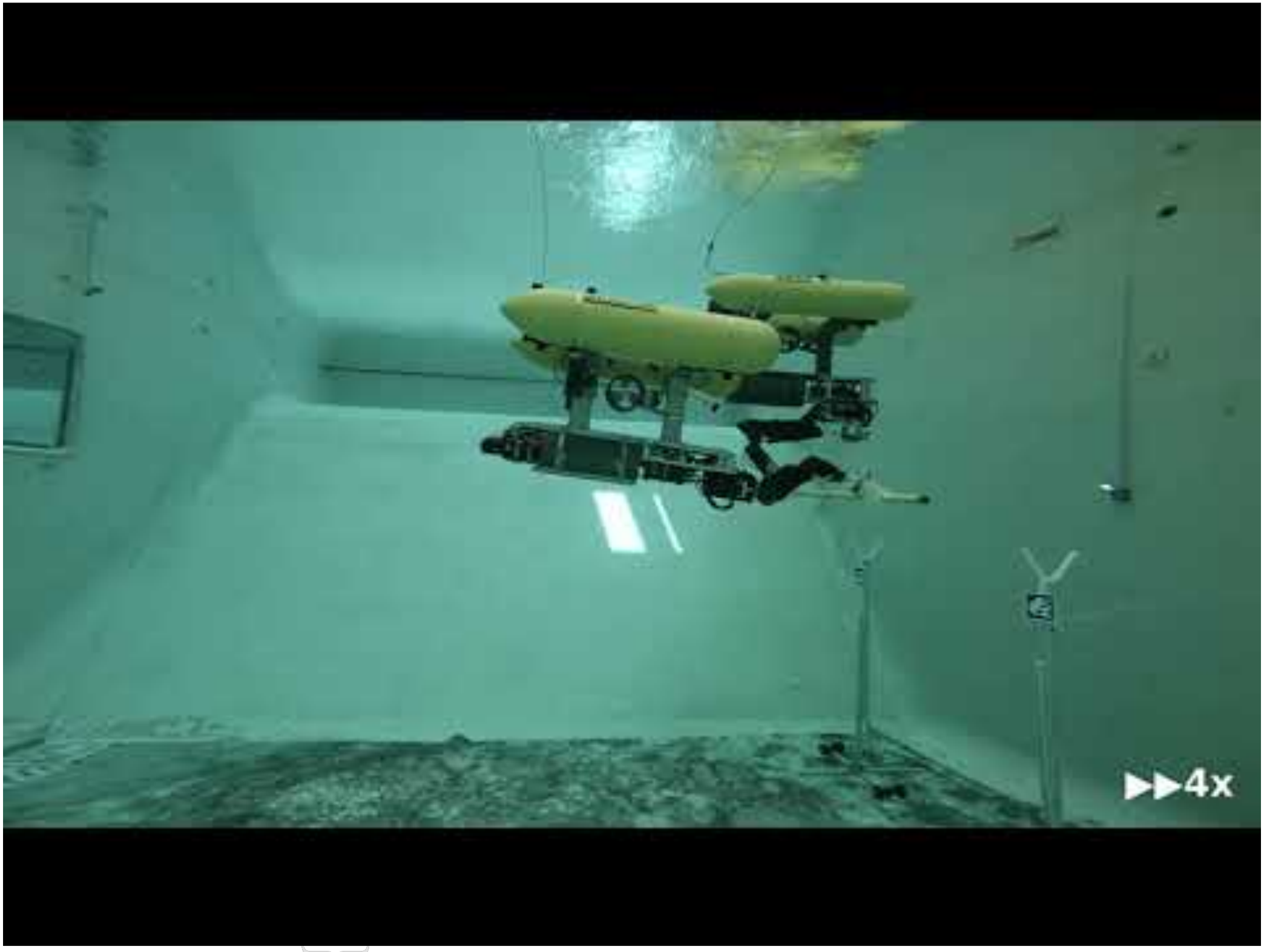
2.3. Robotic intervention: Floating VMS

Huston Mechatronics Aquanaut



2.3. Robotic intervention: Floating VMS

UdG's Girona500 IAUV



2.3. Robotic intervention: Floating VMS

Eelume - hyper-redundant mobile underwater manipulator

2.4. Robotic intervention: Typical tasks & challenges

Typical tasks

- Intervention in environments dangerous or inaccessible to humans (rescue, disaster relief, underwater, heights, space).
- Handling, pick and place, transportation (logistics).
- Turning valves, plugging in/out connectors, checking gauges (industrial installations).
- Sampling in natural environment, structure health monitoring (SHM) requiring contact.
- Performing manipulation tasks on large structures (requiring enormous workspaces).
- Building structures.

Challenges

- Precise control of the mobile base (position and velocity).
- Coordinated control of base and manipulator (redundancy).
- Dealing with the difference in dynamic response between the mobile base and the manipulator/s.
- Grasping of objects in presence of uncertainty (compliant control).
- Navigation and obstacle avoidance (taking into account the base and all the manipulator links).
- Path planning for the whole system (high dimension).
- Calibration between the mobile base and the manipulator.

3.1. Python programming: Linear algebra with Numpy

Matrices and vectors

```
import numpy as np # Library import

A = np.array([[1,2],[3,4]]) # 2x2 matrix
rv = np.array([[1,2,3]]) # 1x3 row vector
ar = np.array([1,2,3]) # 3 element array!!!
v1 = np.array([[1],[2],[3]]) # 3x1 vector
v2 = np.array([4,5,6]).reshape(3,1) # 3x1 vector
Ar = A.reshape((1, -1)) # Converting matrix to row vector
Ac = A.reshape((-1, 1)) # Converting matrix to column vector
c = A[0:2,0] # First column of the matrix
d = A[-1,-1] # Element with highest indexes

print('A: shape', A.shape, '\n', A)
print('rv: shape', rv.shape, '\n', rv)
print('ar: shape', ar.shape, '\n', ar)
print('v1: shape', v1.shape, '\n', v1)
print('v2: shape', v2.shape, '\n', v2)
print('Ar: shape', Ar.shape, '\n', Ar)
print('Ac: shape', Ac.shape, '\n', Ac)
print('c: shape', c.shape, '\n', c)
print('d: shape', d.shape, '\n', d)
```

Special constructors

```
i = np.arange(5) # Array of integer elements
t = np.linspace(0, 0.5, 11) # Array of equally spaced real numbers
B = np.eye(2) # 2x2 identity matrix
C = np.ones((2,2)) # 2x2 matrix filled filled with 1
D = np.ones((2,3)) * 3 # 2x3 matrix filled with 3
Z = np.zeros((3,3)) # 3x3 zero matrix
K = np.diag([2,5,8]) # 3x3 diagonal matrix
R = np.random.rand(2,3) # Matrix of random numbers

print('i:', i)
print('t:', t)
print('B:\n', B)
print('C:\n', C)
print('D:\n', D)
print('Z:\n', Z)
print('K:\n', K)
print('R:\n', R)
```

Output

```
A: shape (2, 2)
 [[1 2]
 [3 4]]
rv: shape (1, 3)
 [[1 2 3]]
ar: shape (3,)
 [1 2 3]
```

Important!!!

```
v1: shape (3, 1)
 [[1]
 [2]
 [3]]
v2: shape (3, 1)
 [[4]
 [5]
 [6]]
Ar: shape (1, 4)
 [[1 2 3 4]]
Ac: shape (4, 1)
 [[1]
 [2]
 [3]
 [4]]
c: shape (2,)
 [1 3]
d: shape ()
 4
```

Important!!!

```
i: [0 1 2 3 4]
t: [0.   0.05 0.1  0.15 0.2  0.25 0.3  0.35
0.4  0.45 0.5 ]
B:
 [[1. 0.]
 [0. 1.]]
C:
 [[1. 1.]
 [1. 1.]]
D:
 [[3. 3. 3.]
 [3. 3. 3.]]
Z:
 [[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
K:
 [[2 0 0]
 [0 5 0]
 [0 0 8]]
R:
 [[0.95600171 0.20768181 0.82844489]
 [0.14928212 0.51280462 0.1359196 ]]
```

3.1. Python programming: Linear algebra with Numpy

Block operations

```
AT1 = A.T # Matrix transpose
AT2 = np.transpose(A) # Matrix transpose
v1T = v1.T # Vector transpose
ABh = np.hstack((A,B)) # Stacking matrices horizontally
ABv = np.vstack((A,B)) # Stacking matrices vertically
L = np.block([[A, A], [B, B]]) # Building matrix from blocks
Z[0:2, 0:2] = B # Block assignment

print('AT:\n', AT1)
print('AT:\n', AT2)
print('v1T:\n', v1T)
print('AB horizontal:\n', ABh)
print('AB vertical:\n', ABv)
print('L:\n', L)
print('Z:\n', Z)
```

Linear algebra

```
J = A + B * 3 # Addition/subtraction
F = A * B # Coefficient-wise multiplication
G1 = A @ B # Matrix multiplication
G2 = np.matmul(A, B) # Matrix multiplication
a = np.dot(v1.T, v2) # Dot product
b = np.cross(v1, v2, 0, 0) # Cross product
iJ = np.linalg.inv(J) # Matrix inverse
iABh = np.linalg.pinv(ABh) # Matrix pseudoinverse

print('J:\n', J)
print('F:\n', F)
print('G:\n', G1)
print('G:\n', G2)
print('a:\n', a)
print('b:\n', b)
print('Inverse of J:\n', iJ)
print('Pseudoinverse of AB horizontal:\n', iABh)
```

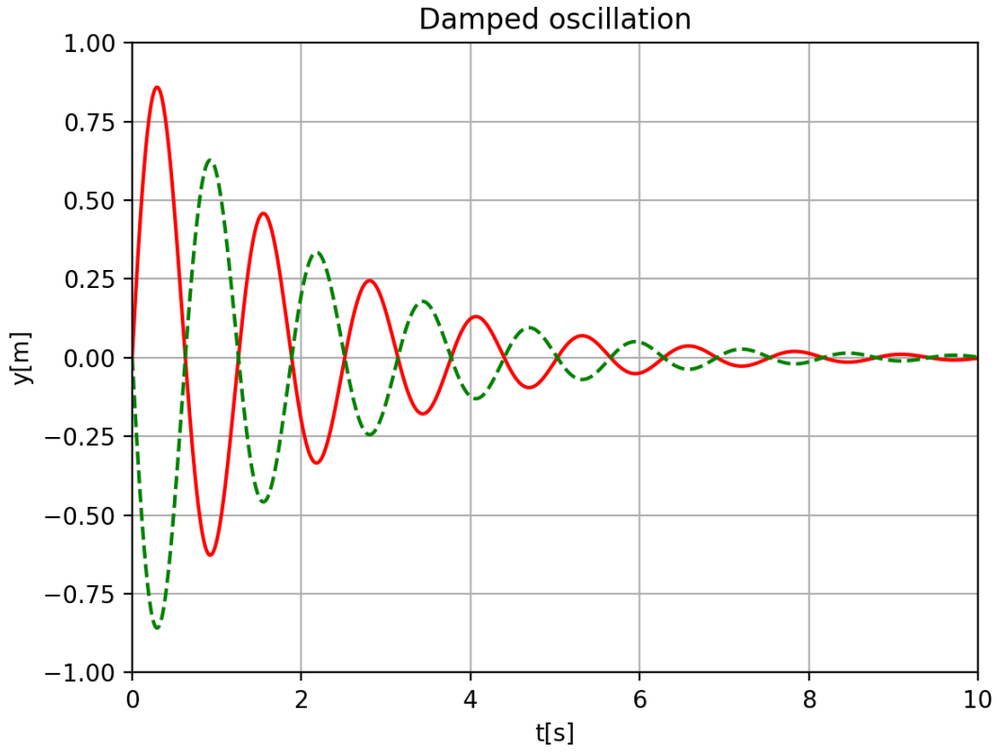
Output

```
AT:
 [[1 3]
 [2 4]]
AT:
 [[1 3]
 [2 4]]
v1T:
 [[1 2 3]]
AB horizontal:
 [[1. 2. 1. 0.]
 [3. 4. 0. 1.]]
AB vertical:
 [[1. 2.]
 [3. 4.]
 [1. 0.]
 [0. 1.]]
L:
 [[1. 2. 1. 2.]
 [3. 4. 3. 4.]
 [1. 0. 1. 0.]
 [0. 1. 0. 1.]]
Z:
 [[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 0.]]
J:
 [[4. 2.]
 [3. 7.]]
F:
 [[1. 0.]
 [0. 4.]]
G:
 [[1. 2.]
 [3. 4.]]
G:
 [[1. 2.]
 [3. 4.]]
a:
 [[32]]
b:
 [[-3  6 -3]]
Inverse of J:
 [[ 0.31818182 -0.09090909]
 [-0.13636364  0.18181818]]
Pseudoinverse of AB horizontal:
 [[-0.2         0.2        ]
 [ 0.22857143  0.05714286]
 [ 0.74285714 -0.31428571]
 [-0.31428571  0.17142857]]
```

3.2. Python programming: Plotting & animation with Matplotlib

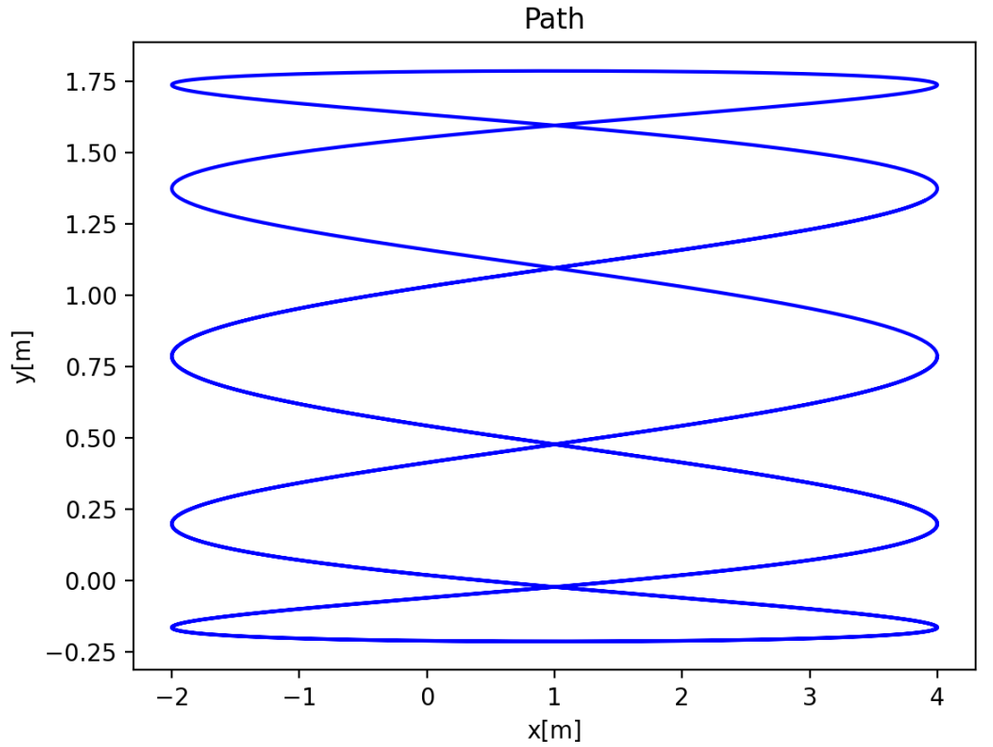
Time plot

```
import numpy as np # Linear algebra library
import matplotlib.pyplot as plt # Plotting library

t = np.linspace(0.0, 10.0, 1000) # Equally spaced time vector
y = np.sin(5.0*t) * np.e ** (-0.5*t) # Function value
plt.plot(t, y, 'r-') # Plot of function value
plt.plot(t, -y, 'g--') # Plot of inverted function value
plt.title('Damped oscillation') # Title of the plot
plt.xlabel('t[s]') # Title of the X axis
plt.ylabel('y[m]') # Title of the Y axis
plt.xlim(left=0.0, right=10.0) # Limits of the X axis
plt.ylim((-1.0, 1.0)) # Limits of the Y axis
plt.grid() # Grid
plt.show() # Show the plot
```



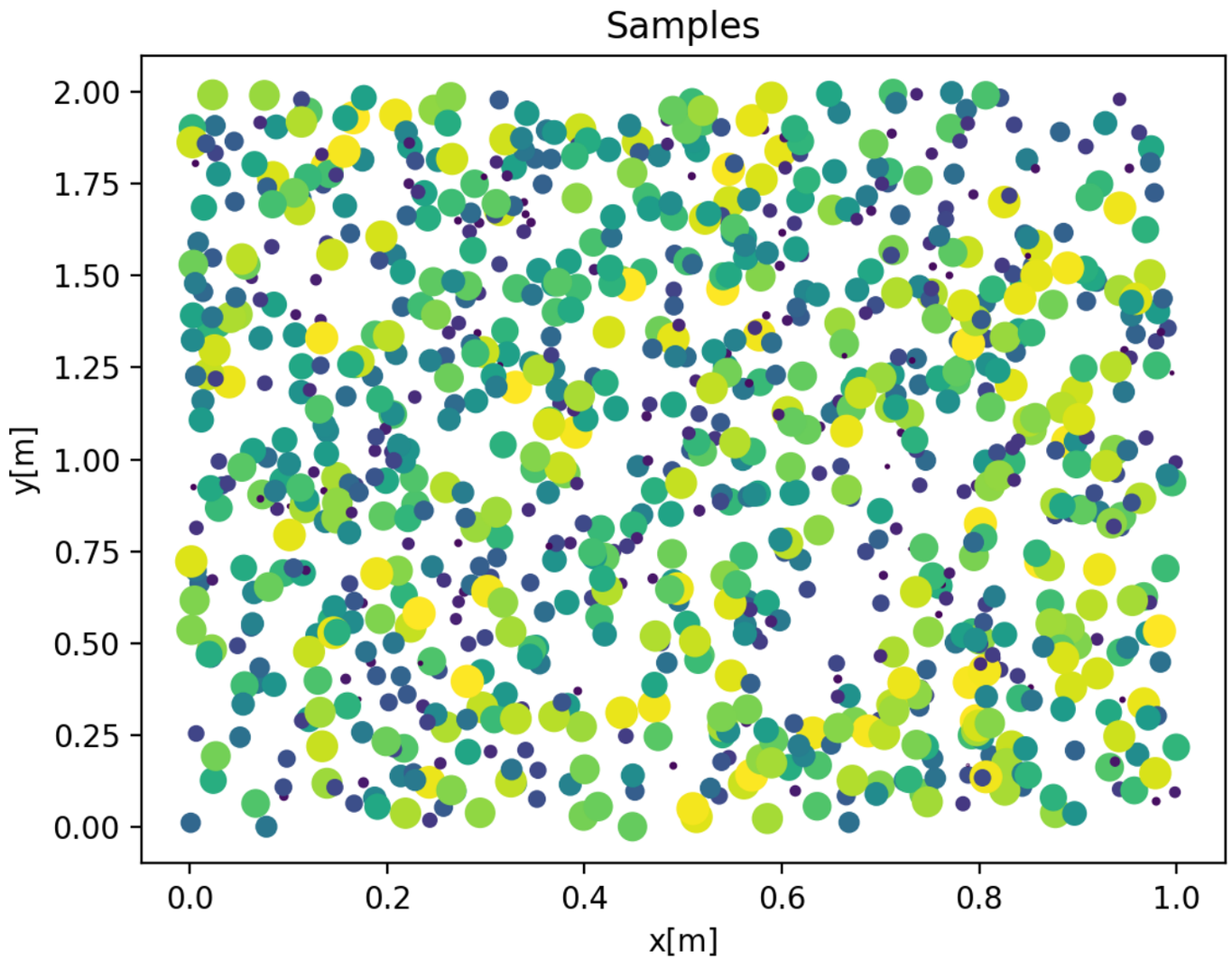
X-Y plot

```
x = 1+3*np.cos(5*t)
y = np.pi/4 - np.sin(t)
plt.plot(x, y, 'b')
plt.title('Path')
plt.xlabel('x[m]')
plt.ylabel('y[m]')
plt.show()
```



Scatter plot

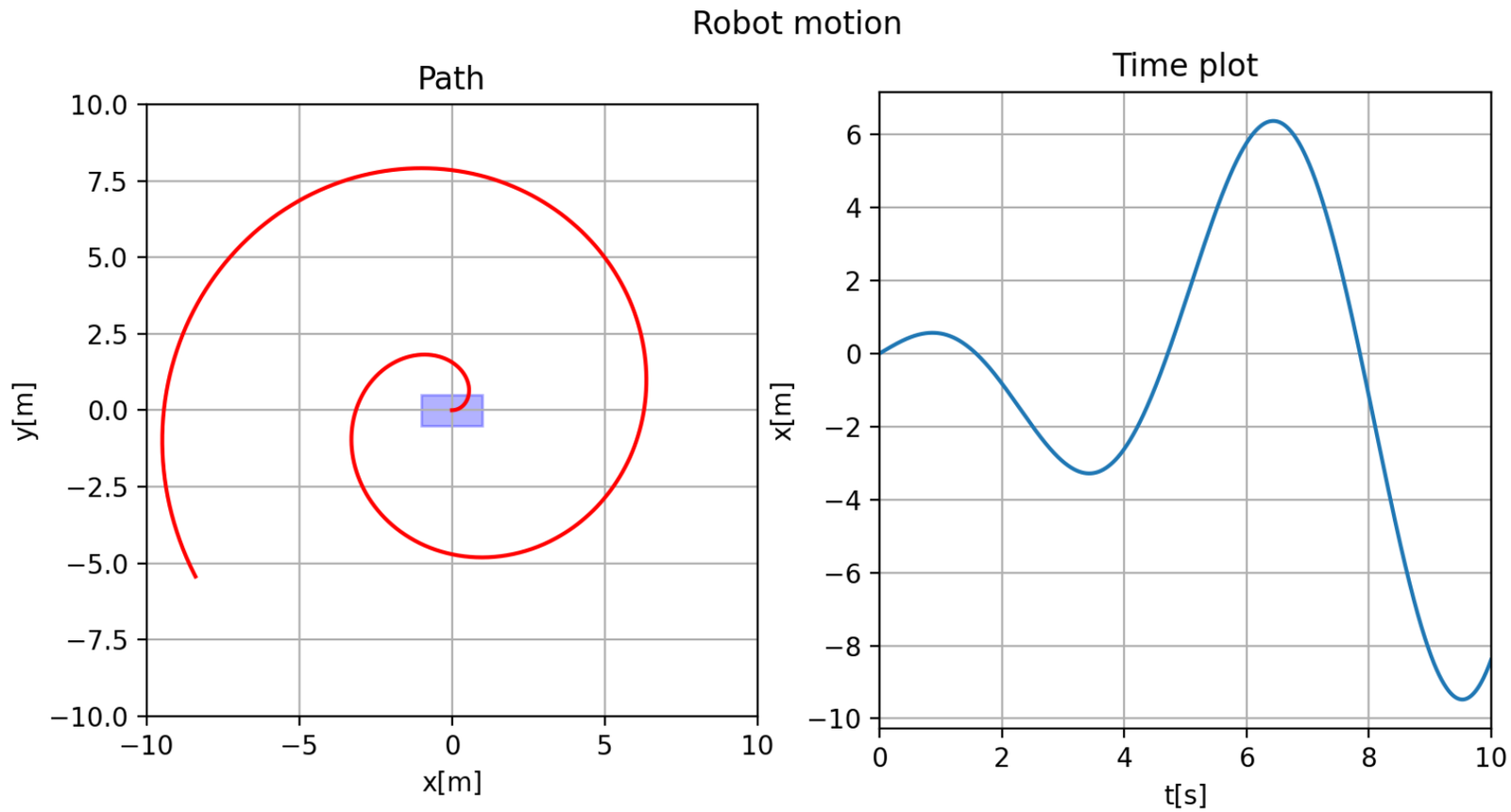
```
r = np.random.rand(1000,3) # Random samples
s = np.array([1.0, 2.0, 100.0]) # Scale
r = s * r # Sample scaling
plt.scatter(r[:,0], r[:,1], s=r[:,2], c=r[:,2], marker='o')
plt.title('Samples')
plt.xlabel('x[m]')
plt.ylabel('y[m]')
plt.show()
```



3.2. Python programming: Plotting & animation with Matplotlib

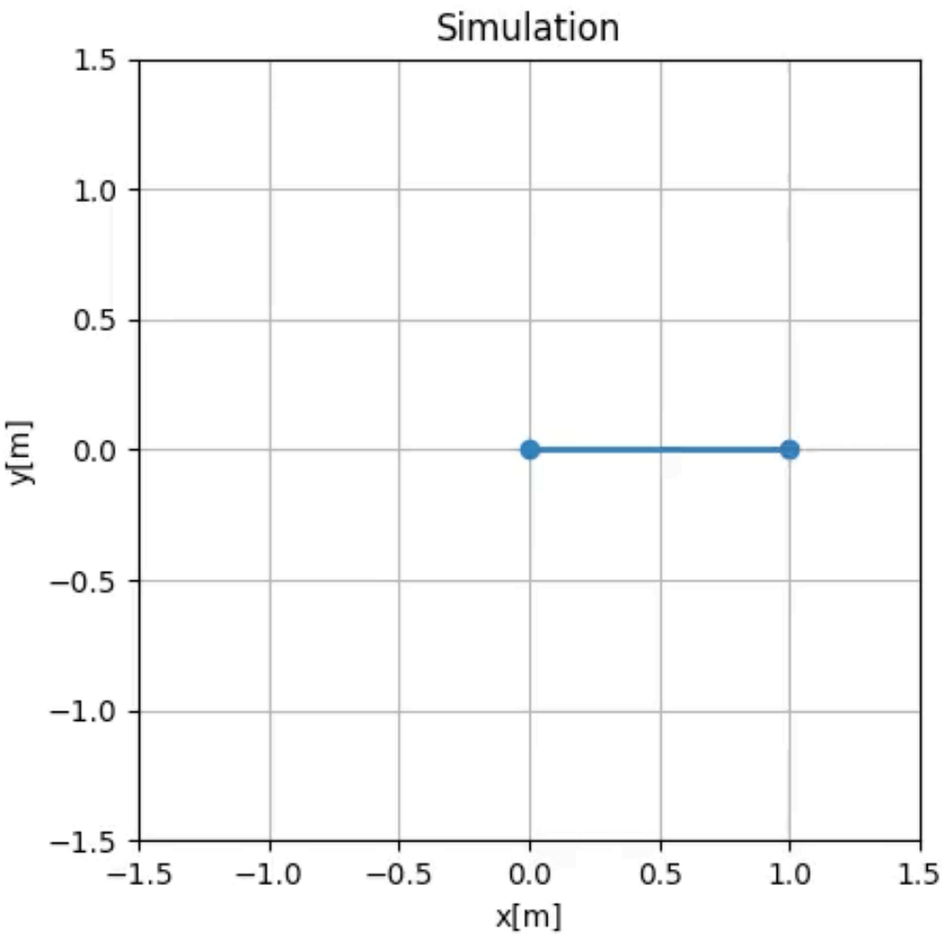
Subplots & geometric figures

```
import matplotlib.patches as patch # Drawing shapes
x = np.cos(t)*t
y = np.sin(t)*t
fig, ax = plt.subplots(1,2) # Plot with 2 subplots
fig.suptitle('Robot motion') # Title of the whole plot
rectangle = patch.Rectangle((-1.0, -0.5), 2.0, 1.0, color='blue', alpha=0.3)
ax[0].add_patch(rectangle) # Robot representation
ax[0].plot(x,y,'r') # First subplot
ax[0].set_title('Path') # Title of the first subplot
ax[0].set(xlabel='x[m]', ylabel='y[m]') # Labels of axes of the first subplot
ax[0].set(xlim=[-10,10], ylim=[-10,10])
ax[0].set_aspect('equal') # Aspect of axes of the first subplot
ax[0].grid() # Grid of the first subplot
ax[1].plot(t,x) # Second plot
ax[1].set_title('Time plot') # Title of the second subplot
ax[1].set(xlabel='t[s]', ylabel='x[m]') # Labels of axes of the second subplot
ax[1].set(xlim=[0,t[-1]]) # Limit of X axis
ax[1].grid() # Grid of the second subplot
plt.show()
```



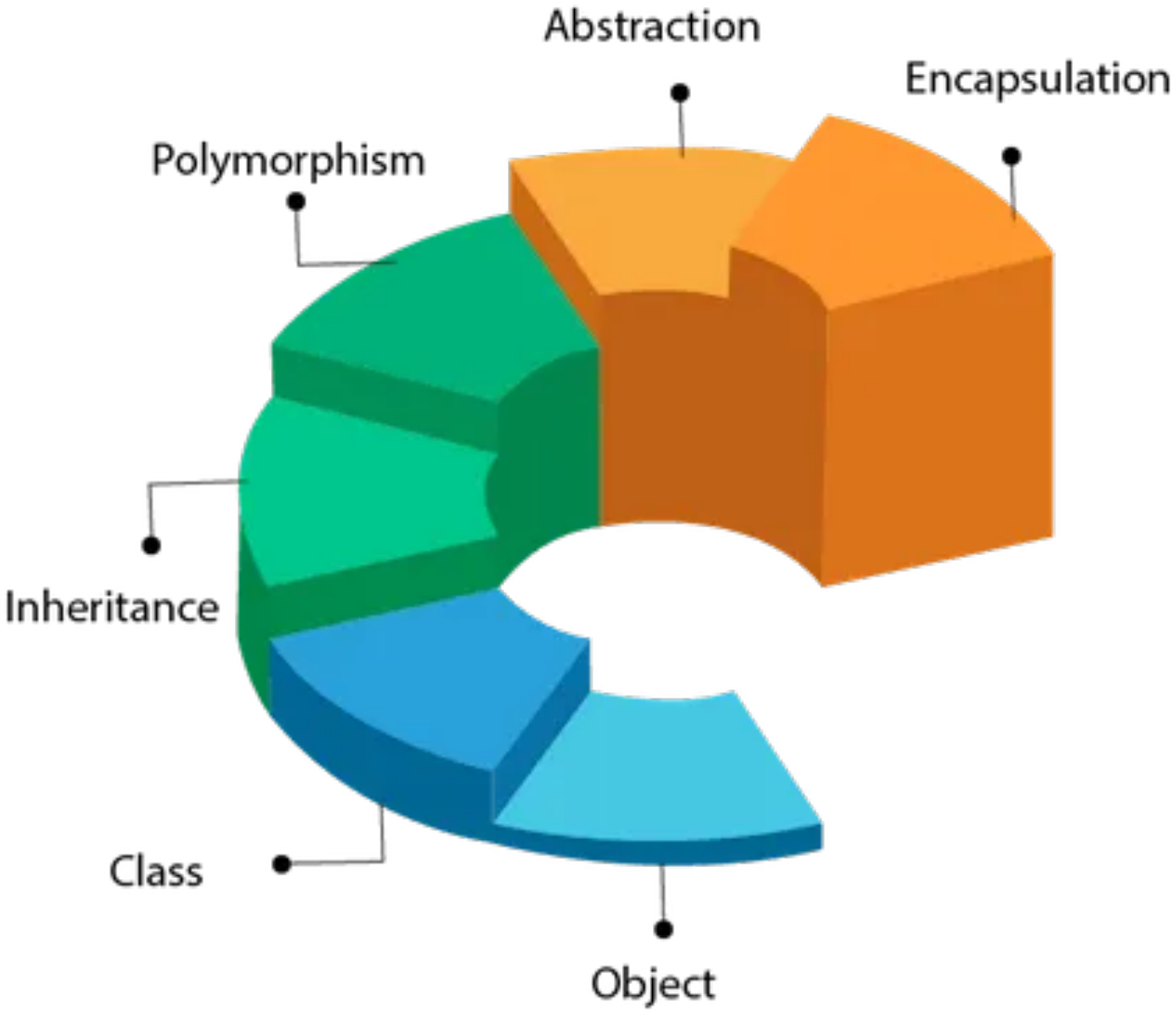
3.2. Python programming: Plotting & animation with Matplotlib

Animation




```python
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as anim

# Simulation params
T = 5*2*np.pi # Simulation time
dt = 1.0/60.0 # Sampling time

# Drawing preparation (figure with one subplot)
fig = plt.figure()
ax = fig.add_subplot(111, autoscale_on=False, xlim=(-1.5, 1.5), ylim=(-1.5, 1.5))
ax.set_title('Simulation')
ax.set_aspect('equal')
ax.set(xlabel='x[m]', ylabel='y[m]')
ax.grid()
line, = ax.plot([], [], 'o-', lw=2) # Line for displaying the structure
path, = ax.plot([], [], 'r-', lw=1) # Line for displaying the path

# Path memory
Px = []; Py = []

# Function initialising simulation and drawing
def init():
    line.set_data([], [])
    path.set_data([], [])
    return line, path

# Function updating the simulation (main loop)
def simulate(t):
    # Computing new position based on time
    P = np.zeros((2,2))
    r = 1.0 + 0.2 * np.sin(t*7.2)
    P[0,1] = np.cos(t) * r
    P[1,1] = np.sin(t) * r

    # Saving new point in path memory
    Px.append(P[0,1])
    Py.append(P[1,1])

    # Updating the drawing
    line.set_data(P[0,:], P[1,:])
    path.set_data(Px, Py)
    return line, path

# Create and run simulation
animation = anim.FuncAnimation(fig, simulate, np.arange(0, T, dt),
                               interval=10, blit=True, init_func=init, repeat=False)
plt.show()
#animation.save('anim.mp4', fps=60) # To save simulation to a file
```

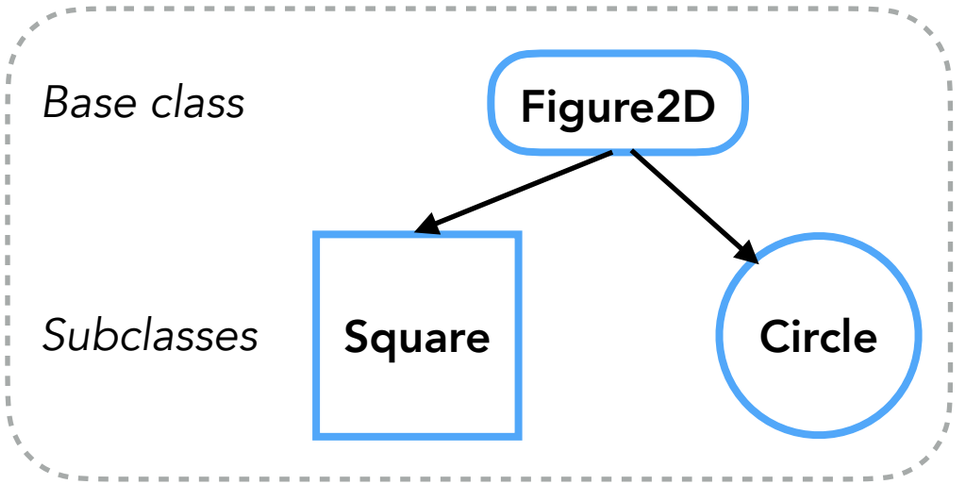
Object-oriented programming (OOP)

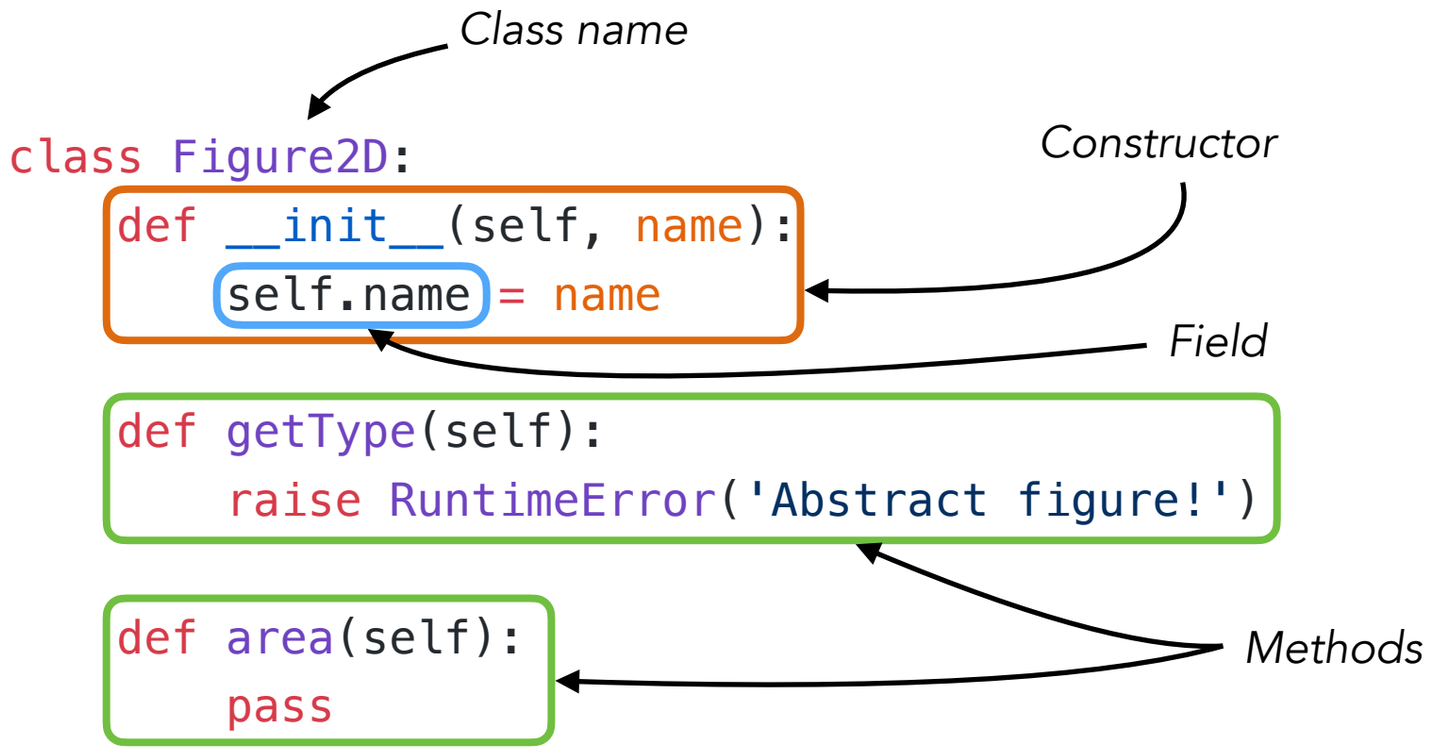


3.3. Python programming: Classes & inheritance

Simple classes

```
# Class definitions
# Note: there is no support for abstract classes in Python
class Figure2D:
    def __init__(self, name):
        self.name = name

    def getType(self):
        raise RuntimeError('Abstract figure!')

    def area(self):
        pass

class Square(Figure2D):
    def __init__(self, name, sideLength):
        super().__init__(name)
        self.a = sideLength

    def getType(self):
        return 'Square'

    def area(self):
        return self.a**2

class Circle(Figure2D):
    def __init__(self, name, radius):
        super().__init__(name)
        self.r = radius

    def getType(self):
        return 'Circle'

    def area(self):
        return 3.14159*self.r**2

# Main loop
figures = [Square('square1', 0.5), Circle('circle1', 0.2)]

for f in figures:
    print(f.name, f.getType(), f.area())
```






```
square1 Square 0.25
circle1 Circle 0.12566370614359174
```

3.3. Python programming: Classes & inheritance

Inheritance

Subclass name *Base class name*

```
class Square(Figure2D):
    def __init__(self, name, sideLength):
        super().__init__(name)
        self.a = sideLength

    def getType(self):
        return 'Square'

    def area(self):
        return self.a**2
```

Constructor

Superclass contructor (constructor of Figure2D)

Overriden methods

```
class Circle(Figure2D):
    def __init__(self, name, radius):
        super().__init__(name)
        self.r = radius

    def getType(self):
        return 'Circle'

    def area(self):
        return 3.14159*self.r**2
```

Creating objects

Objects

```
figures = [Square('square1', 0.5), Circle('circle1', 0.2)]

for f in figures:
    print(f.name, f.getType(), f.area())
```

Accessing fields

Accessing methods

```
square1 Square 0.25
circle1 Circle 0.12566370614359174
```