{Learn, Create, Innovate};

ROS2

Solving ODE's in ROS 2

Manchester Robotics

Introduction

Introduction

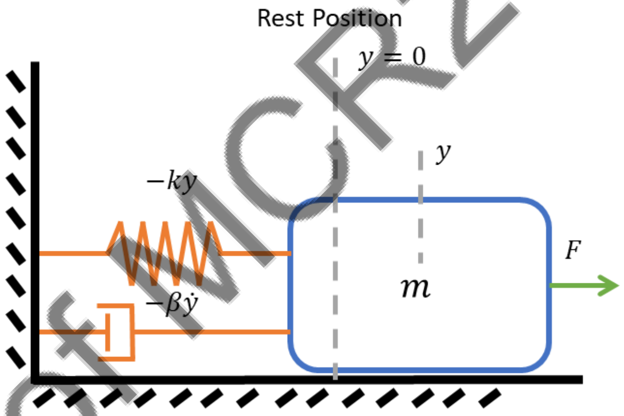
- Ordinary Differential Equations (ODEs) play a crucial role in robotics for modelling dynamic systems such as robot motion, control, and sensor behaviour.
- In this class, we will focus on solving second-order ODEs using **Euler's Method** and implement it in **ROS2**.

Dynamic Simulation

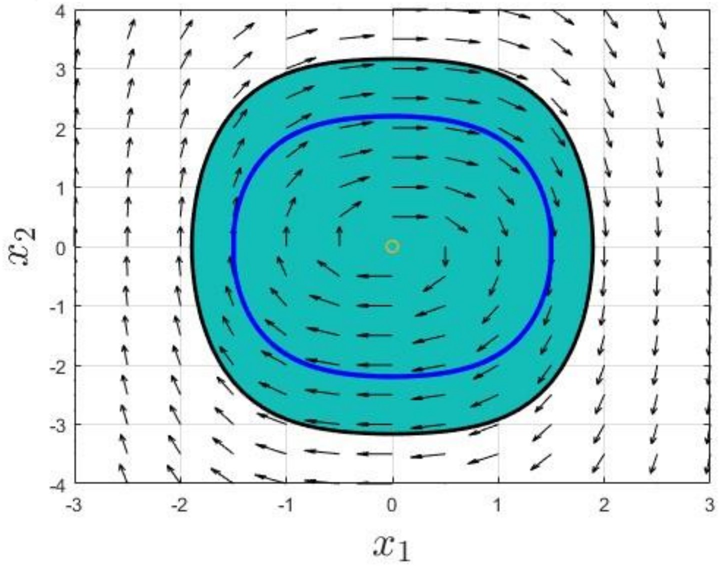
- **Dynamic simulation** (or dynamic system simulation) is using a computer program to model the time-varying behaviour of a dynamical system.
- Ordinary differential equations or partial differential equations typically describe the systems.
- The simulator solves these equations to determine the behaviour of state variables over a specified time.
- Creating a model of a dynamic system allows for predicting the values of the model-system state variables based on past state values.

Dynamic Simulation

- Simulation models are commonly obtained from discrete-time approximations of continuous-time mathematical models.
- Models can incorporate real-world constraints, like gear backlash, collisions, and rebound from a hard stop.
- As models are more complex, equations can become nonlinear, chaotic, with added disturbances and noise.



$$\dot{\boldsymbol{x}} = \begin{bmatrix} 0 & 1 \\ -\frac{k}{m} & -\frac{\beta}{m} \end{bmatrix} \boldsymbol{x} + \begin{bmatrix} 0 \\ 1/m \end{bmatrix} F$$



Dynamic Simulation

- Solving some nonlinear equations “by hand” can become difficult or almost impossible.
- Thanks to the advancement of computers, it is now possible to solve them using different computational algorithms.
- Dynamical models, in general, are solved through numerical integration methods to produce the transient behaviour of the state variables.
- Therefore, it can be said that a numerical simulation is done by stepping through a time interval and calculating the solution of the mathematical model solution through numerical integration.

Lorenz Attractor

Discrete-time dynamic models

Introduction

Discrete-time dynamic models

- A digital computer by its very nature, deals internally with discrete-time data or numerical values of functions at equally spaced intervals determined by the sampling period.
- Thus, discrete-time models such as difference equations are widely used in computer control applications.
- One way a continuous-time dynamic model can be converted to discrete-time form is by employing a finite difference approximation.

- Consider a nonlinear differential equation

$$\frac{dy(t)}{dt} = f(y,u) \qquad (30)$$

where y is the output variable and u is the input variable.

Discrete-time dynamic models

- This equation can be numerically integrated (for instance using Euler method) by introducing a finite difference approximation for the derivative.
- For example, the first-order, backward difference approximation to the derivative at $t = k\Delta t$ is:

$$\frac{dy(t)}{dt} \cong \frac{y(k) - y(k-1)}{\Delta t} \quad (31)$$

where Δt is the integration interval (the control engineers name it sampling time) specified by the user and $y(k)$ denotes the values of $y(k)$ at $t = k\Delta t$.

- So,

$$\frac{y(k) - y(k-1)}{\Delta t} \cong f(y(k-1), u(k-1)) \quad (32)$$

- or:

$$y(k) = y(k-1) + \Delta t \cdot f(y(k-1), u(k-1)) \quad (33)$$

- This is a first-order difference equation that can be used to predict $y(k)$ based on information at the previous time step $(k-1)$. This type of expression is called a recurrence relation.

Euler’s Method

- Euler's method is a numerical approach for solving differential equations by approximating solutions iteratively.
- For a first-order ODE:

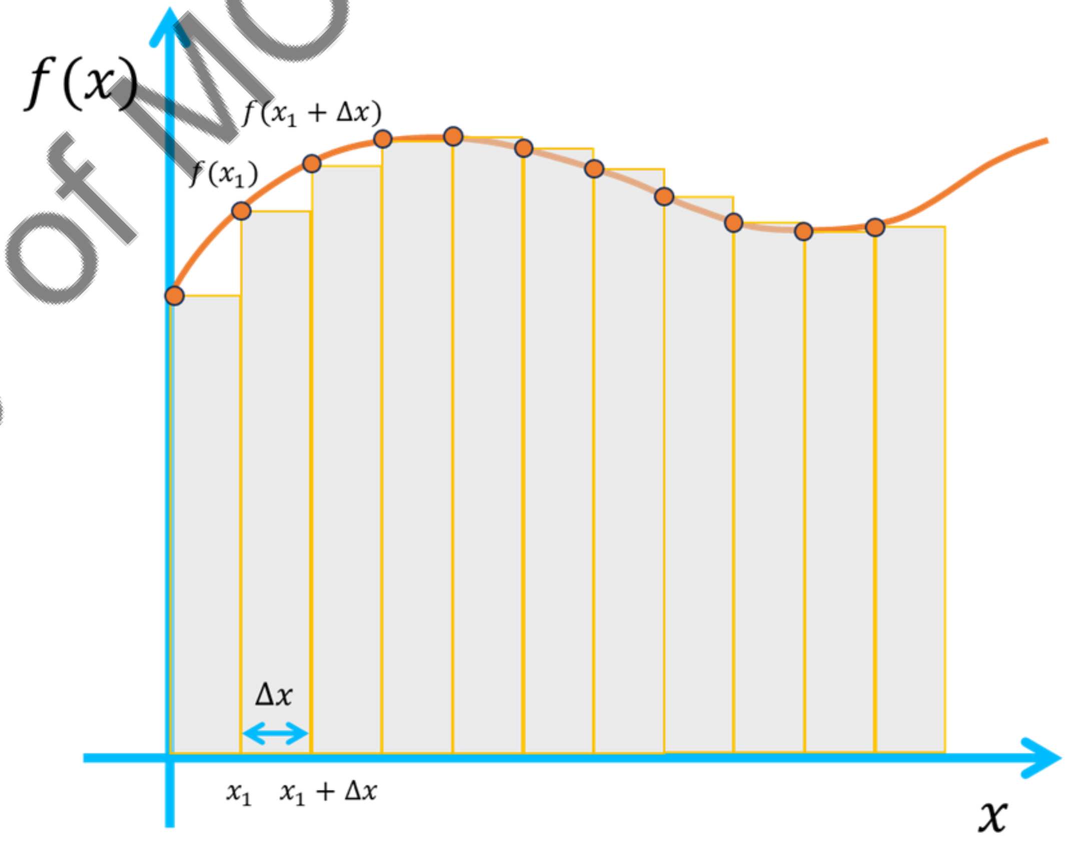
$$\dot{x} = f(x)$$

- Euler's approximation is given by:

$$x_{n+1} = x_n + \Delta x \cdot f(x_n)$$

- where: Δx is the step size, x_n is the current value and x_{n+1} is the next estimated value.

Euler’s Method



Discrete-time dynamic models

- For higher-order ODEs, we can use a generalisation of the Euler method that we used for solving first-order ODEs. To illustrate the method, let us consider a 2nd order ODE:

$$\frac{d^2\,y(t)}{dt^2} = f(t, y, \frac{dy(t)}{dt}) \qquad (34)$$

- Or:

$$\ddot{y} = f(t, y, \dot{y}) \qquad (35)$$

- For discretization, the idea is to write the second order system (ODE) as a system of two first order systems (ODEs) and then apply Euler's method to the first order equations.
- So, defining a new variable:

$$\begin{cases} y = x_1 \\ \dot{y} = x_2 = \dot{x}_1 \end{cases} \qquad (36)$$

$$\begin{cases} \dot{x}_1 = x_2 \\ \dot{x}_2 = f(t, x_1, x_2) \end{cases} \qquad (37)$$

- We need initial conditions:

For Newton the initial conditions are the initial position and initial velocity.

$$\begin{cases} x_1(t_0) = 0 \\ x_2(t_0) = 0 \end{cases} \qquad (38)$$

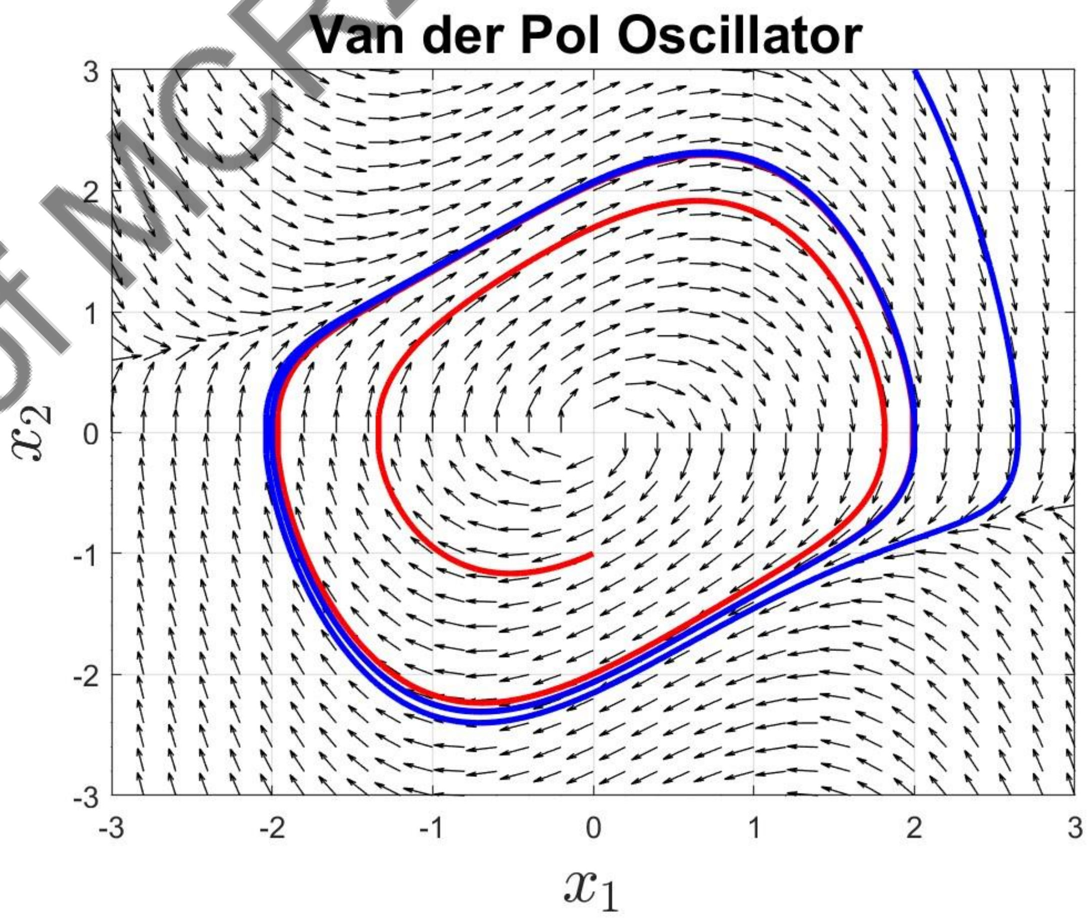
Discrete-time dynamic models

- Now, the idea is to solve both x_1 and x_2 simultaneously using Euler's method for both first order ODEs:

$$\begin{cases} \dfrac{x_1(k) - x_1(k-1)}{\Delta t} = x_2(k-1) \\ \dfrac{x_2(k) - x_2(k-1)}{\Delta t} = f((k-1), x_1, x_2) \end{cases} \quad (39)$$

$$\begin{cases} x_1(k) = x_1(k-1) + \Delta t \cdot x_2(k-1) \\ x_2(k) = x_2(k-1) + \Delta t \cdot f((k-1), x_1, x_2) \end{cases} \quad (40)$$

- This can be generalized to third order ODEs, or fourth order ODEs, as well as n order ODEs.



Activity

DC Motor Simulation

`{Learn, Create, Innovate};`

Manchester Robotics

motor_control package

Requirements

- Download the motor_control template package from Github.

Instructions

- Download the motor_control package from GitHub (inside Templates).
- Add it to your source directory inside your workspace

```
motor_control/
├── launch
│   └── motor_launch.py
├── LICENSE
├── motor_control
│   ├── dc_motor.py
│   ├── __init__.py
│   └── set_point.py
├── package.xml
├── resource
│   └── motor_control
├── setup.cfg
├── setup.py
└── test
    ├── test_copyright.py
    ├── test_flake8.py
    └── test_pep257.py
```

DC Motor Simulation

Motor Control package

- The package is composed of two nodes:
  - dc_motor node: Simulate a First Order System, representing a DC Motor.
  - set_point node: Providing an input for the system

```
motor_control/motor_control/dc_motor.py
motor_control/motor_control/set_point.py
```

- You can see the contents of each node by opening the file on any text editor (gedit, vscode, nano, vim, etc.)

DC Motor Node

- The DC Motor will be simulated using a First Oder system shown in here.

$$\tau \frac{dy(t)}{dt} + y(t) = Ku(t).$$

where, τ is the time constant, K is the system gain, $y(t)$ is the system output (speed rad/s) and $u(t)$ the input signal (volts).

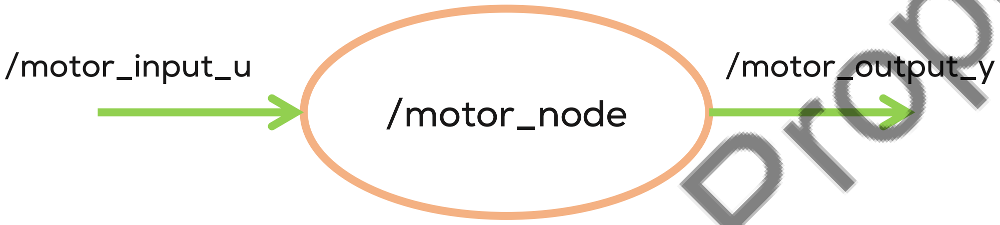
$$y[k+1] = y[k] + \left(-\frac{1}{\tau} \cdot y[k] + \frac{K}{\tau} \ u[k] \right) T_s$$

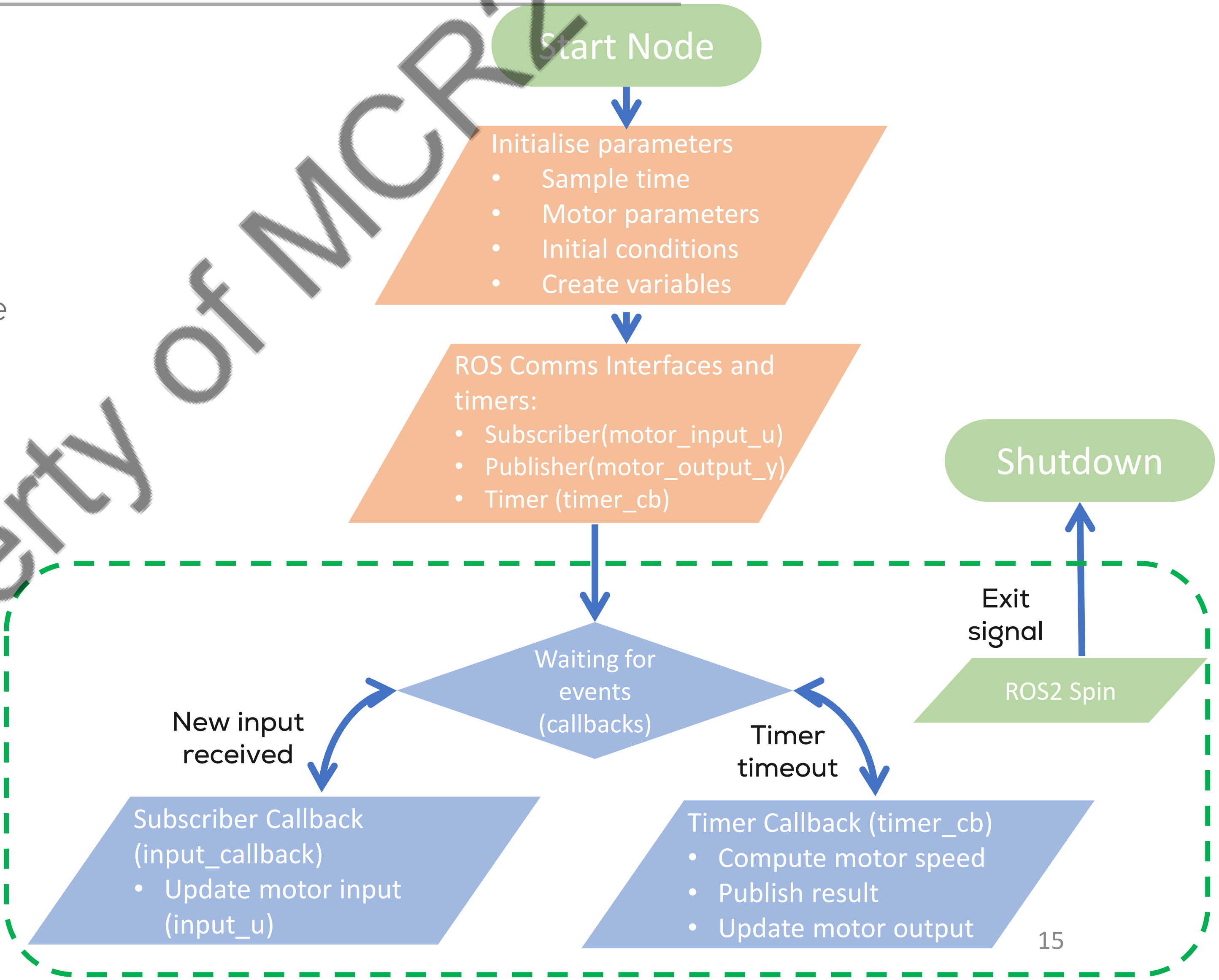
Where T_s is the sampling time.

DC Motor Simulation

DC Motor Node Structure

- The node subscribes to the topic "/motor_input_u" and publishes the vales of the motor speed on the topic "/motor_output_y".
- Both topics contain an interface (message) Float32





dc_motor.py

```python
# Imports
import rclpy
from rclpy.node import Node
from std_msgs.msg import Float32

#Class Definition
class DCMotor(Node):
    def __init__(self):
        super().__init__('dc_motor')

        # DC Motor Parameters
        self.sample_time = 0.02
        self.param_K = 1.75
        self.param_T = 0.5
        self.initial_conditions = 0.0

        #Set the messages
        self.motor_output_msg = Float32()

        #Set variables to be used
        self.input_u = 0.0
        self.output_y = self.initial_conditions

        #Declare publishers, subscribers and timers
        self.motor_input_sub = self.create_subscription(Float32, 'motor_input_u', 
self.input_callback,10)
        self.motor_speed_pub = self.create_publisher(Float32, 'motor_speed_y', 10)
        self.timer = self.create_timer(self.sample_time, self.timer_cb)

        #Node Started
        self.get_logger().info('Dynamical System Node Started \U0001F680')
```

Libraries

Initialise parameters

ROS publishers, subscribers Timers

```python
    #Timer Callback
    def timer_cb(self):
        #DC Motor Simulation
        #DC Motor Equation y[k+1] = y[k] + ((-1/τ) y[k] + (K/τ) 
u[k]) T_s
        self.output_y += (-1.0/self.param_T * self.output_y + 
self.param_K/self.param_T * self.input_u) * self.sample_time
        #Publish the result
        self.motor_output_msg.data = self.output_y
        self.motor_speed_pub.publish(self.motor_output_msg)

    #Subscriber Callback
    def input_callback(self, input_sgn):
        self.input_u = input_sgn.data

#Main
def main(args=None):
    rclpy.init(args=args)

    node = DCMotor()

    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass
    finally:
        node.destroy_node()
        rclpy.try_shutdown()

#Execute Node
if __name__ == '__main__':
    main()
```

Timer callback: Motor simulation

Subscriber callback

Main

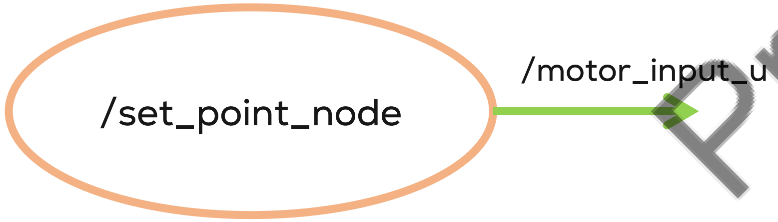
DC Motor Simulation

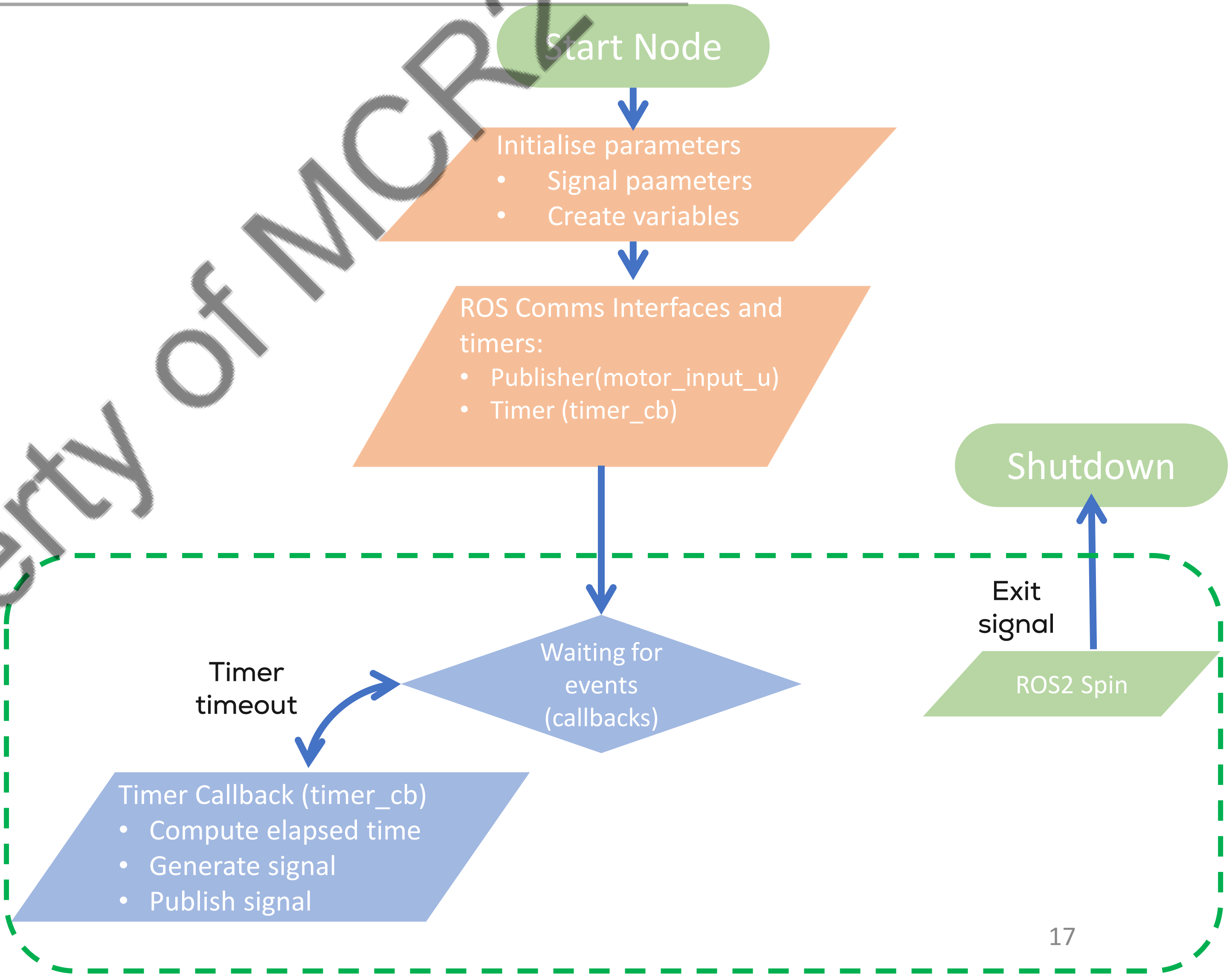
Set Point node structure

- The node publishes the vales of input signal on the topic "/motor_input_u".

$$u(t) = A\ sin(\omega t)$$

- The topic contain an interface (message) Float32





set_point.py

```python
# Imports
import rclpy
from rclpy.node import Node
import numpy as np
from std_msgs.msg import Float32

#Class Definition
class SetPointPublisher(Node):
    def __init__(self):
        super().__init__('set_point_node')

        # Retrieve sine wave parameters
        self.amplitude = 2.0
        self.omega  = 1.0

        #Create a publisher and timer for the signal
        self.signal_publisher = self.create_publisher(Float32,
'motor_input_u', 10)
        timer_period = 0.1 #seconds
        self.timer = self.create_timer(timer_period, self.timer_cb)

        #Create a messages and variables to be used
        self.signal_msg = Float32()
        self.start_time = self.get_clock().now()

        self.get_logger().info("SetPoint Node Started \U0001F680")
```

Libraries

Initialise parameters

ROS publishers, subscribers Timers

```python
    # Timer Callback: Generate and Publish Sine Wave Signal
    def timer_cb(self):
        #Calculate elapsed time
        elapsed_time = (self.get_clock().now() -
self.start_time).nanoseconds/1e9
        # Generate sine wave signal
        self.signal_msg.data = self.amplitude *
np.sin(self.omega * elapsed_time)
        # Publish the signal
        self.signal_publisher.publish(self.signal_msg)

#Main
def main(args=None):
    rclpy.init(args=args)

    set_point = SetPointPublisher()

    try:
        rclpy.spin(set_point)
    except KeyboardInterrupt:
        pass
    finally:
        set_point.destroy_node()
        rclpy.try_shutdown()

#Execute Node
if __name__ == '__main__':
    main()
```

Timer callback:

Signal Generator

Main

DC Motor Simulation

Instructions

- Compile the package using colcon

```
$ cd ~/ros2_ws
$ colcon build
$ source install/setup.bash
```

- Launch the package

```
$ ros2 launch motor_control motor_launch.py
```

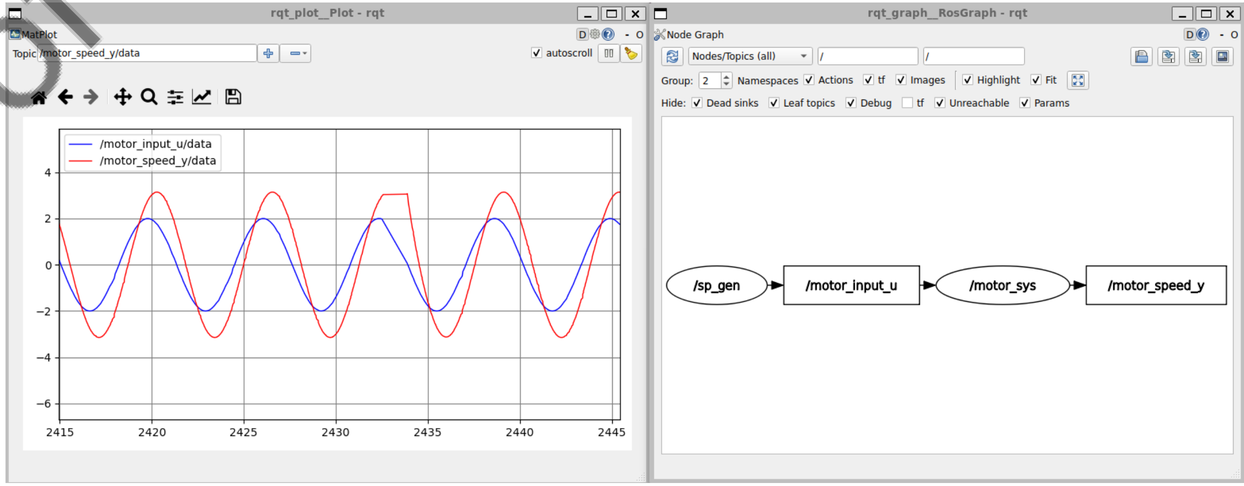
- Open two terminals run the rqt_graph and the rqt_plot

```
$ ros2 run rqt_plot rqt_plot
```

```
$ ros2 run rqt_graph rqt_graph
```

Results

- If everything goes well, you should see the following



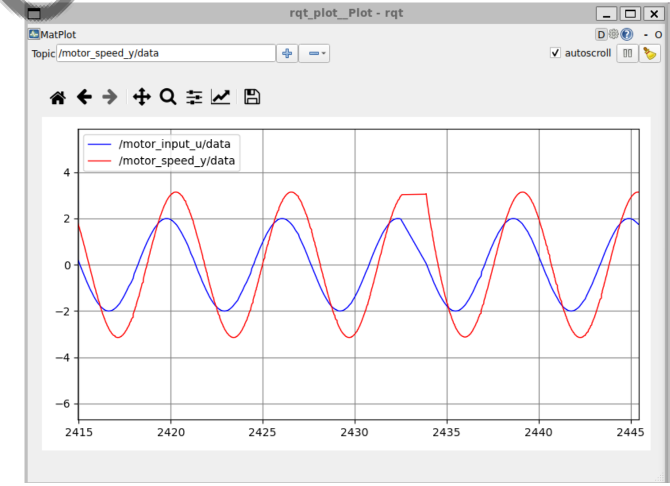
- Check the published topics

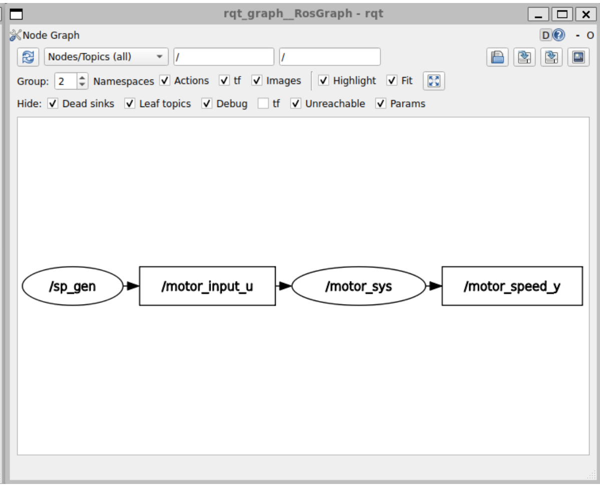
```
mario@MarioPC:~$ ros2 topic list
/motor_input_u
/motor_speed_y
/parameter_events
/rosout
```

motor_launch.py

```python
from launch import LaunchDescription
from launch_ros.actions import Node

def generate_launch_description():
    motor_node = Node(name="motor_sys",
                       package='motor_control',
                       executable='dc_motor',
                       emulate_tty=True,
                       output='screen',
                       )

    sp_node = Node(name="sp_gen",
                       package='motor_control',
                       executable='set_point',
                       emulate_tty=True,
                       output='screen',
                       )

    l_d = LaunchDescription([motor_node, sp_node])

    return l_d
```

- The launch file starts a motor_node and a set_point node.





Q&A

Questions?

{Learn, Create, Innovate};

Manchester Robotics

Thank You

Robotics For Everyone

{Learn, Create, Innovate};

MCR2

Manchester **Robotics**

T&C

Terms and conditions

`{Learn, Create, Innovate};`

Manchester Robotics

Terms and conditions

- *THE PIECES, IMAGES, VIDEOS, DOCUMENTATION, ETC. SHOWN HERE ARE FOR INFORMATIVE PURPOSES ONLY. THE DESIGN IS PROPRIETARY AND CONFIDENTIAL TO MANCHESTER ROBOTICS LTD. (MCR2). THE INFORMATION, CODE, SIMULATORS, DRAWINGS, VIDEOS PRESENTATIONS ETC. CONTAINED IN THIS PRESENTATION IS THE SOLE PROPERTY OF MANCHESTER ROBOTICS LTD. ANY REPRODUCTION, RESELL, REDISTRIBUTION OR USAGE IN PART OR AS A WHOLE WITHOUT THE WRITTEN PERMISSION OF MANCHESTER ROBOTICS LTD. IS STRICTLY PROHIBITED.*
- *THIS PRESENTATION MAY CONTAIN LINKS TO OTHER WEBSITES OR CONTENT BELONGING TO OR ORIGINATING FROM THIRD PARTIES OR LINKS TO WEBSITES AND FEATURES IN BANNERS OR OTHER ADVERTISING. SUCH EXTERNAL LINKS ARE NOT INVESTIGATED, MONITORED, OR CHECKED FOR ACCURACY, ADEQUACY, VALIDITY, RELIABILITY, AVAILABILITY OR COMPLETENESS BY US.*
- *WE DO NOT WARRANT, ENDORSE, GUARANTEE, OR ASSUME RESPONSIBILITY FOR THE ACCURACY OR RELIABILITY OF ANY INFORMATION OFFERED BY THIRD-PARTY WEBSITES LINKED THROUGH THE SITE OR ANY WEBSITE OR FEATURE LINKED IN ANY BANNER OR OTHER ADVERTISING.*