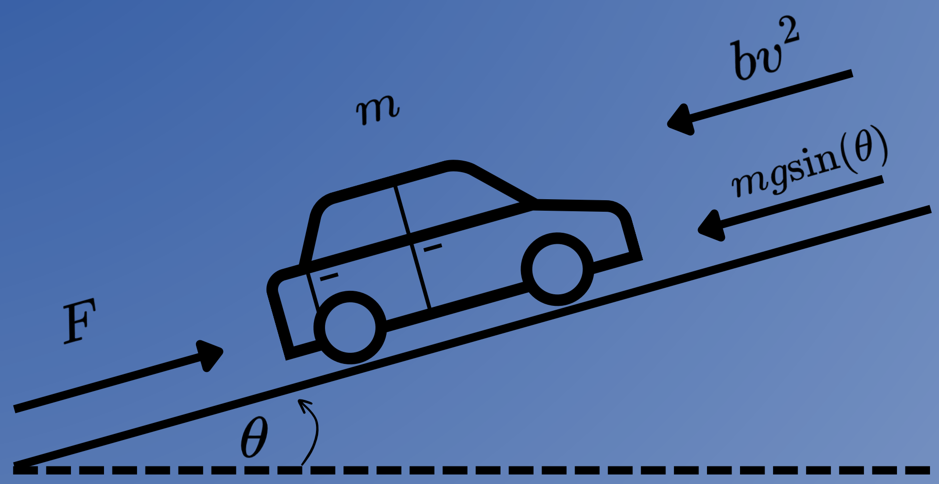
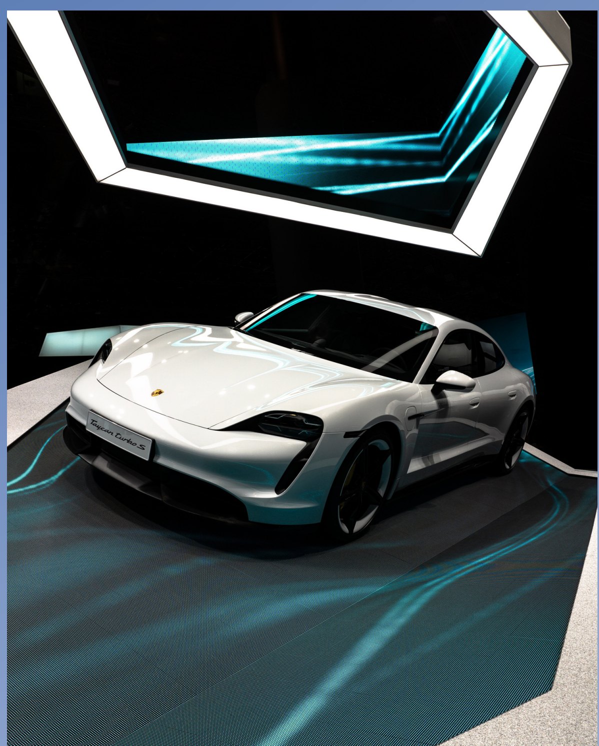
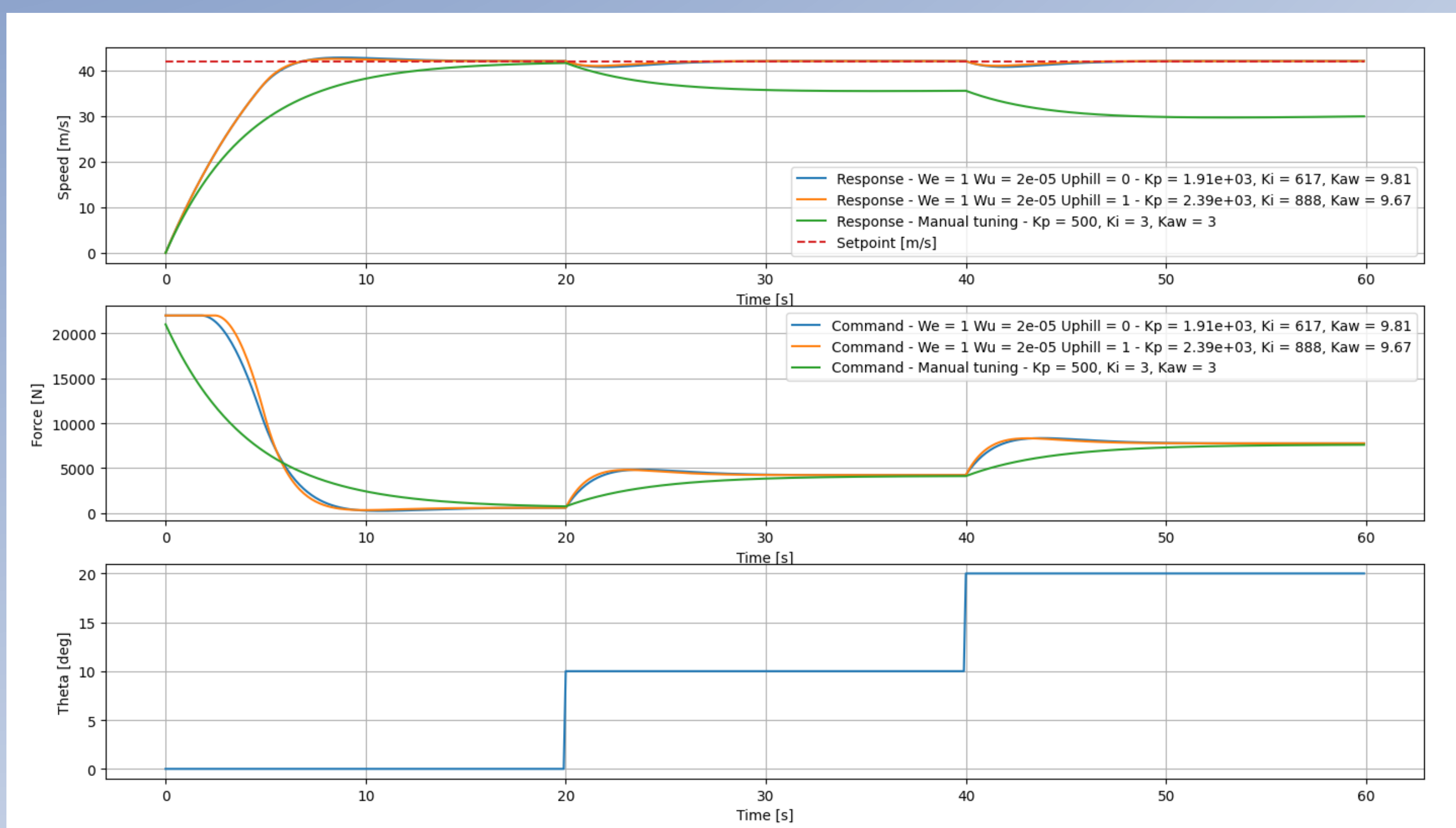
Speed control optimisation on a slope





```python
def Simulation(x, time_step, end_time, m, b, F_max_0, F_max_max, v_max):

    """ Simulate the PID control of a car with given parameters.

        Returns:
        (t, stp, z, command): arrays of time, setpoints, positions, and commands
    """

    length = round(end_time/time_step)

    t = np.zeros(length)
    stp = np.zeros(length)
    v = np.zeros(length)
    command = np.zeros(length)

    # A PI controller is considered - Kd and T_C are set = 0 - this is based on the knowledge that
    # for this problem a PI is sufficient
    [Kp, Ki, Kaw] = x
    Kd = 0
    T_C = 0

    # Initialize PID controller
    pid = PID(Kp, Ki, Kd, Kaw, T_C, time_step, F_max_0, 0, 300000)

    # Initialize car with given parameters
    car = Car(m, b, F_max_0, F_max_max, v_max, time_step)

    # Iterate through time steps
    for idx in range(0, length):
        t[idx] = idx*time_step
        # Set setpoint
        stp[idx] = 42

        # Execute the control loop
        v[idx] = car.v
        pid.Step(v[idx], stp[idx])
        command[idx] = pid.command_sat
        car.Step(command[idx])

    return (t, stp, v, command)

def Cost(x, time_step, end_time, m, b, F_max_0, F_max_max, v_max, We, Wu):
    """ Calculate the cost function for a given set of parameters.

        Inputs:
        x: PID parameters [Kp, Ki, Kd, Kaw, T_C]
        We: weight on control error
        Wu: weight on control effort

        Returns:
        cost: scalar value representing the total cost
    """

    # Simulate
    (t, stp, v, command) = Simulation(x, time_step, end_time, m, b, F_max_0, F_max_max, v_max)

    # Cost function
    # J = sum((stp[i] - v[i])^2*t[i])*We + sum((command[i+1] - command[i])^2)*Wu + command[0]^2*Wu
    cost = np.sum(np.square(stp - v)*t)*We + np.sum(np.square(np.diff(command)))*Wu + command[0]*command[0]*Wu

    return cost
```



Simone Bertoni

Code
https://github.com/simorxb/PID-tuning-opt-car-slope

Idea

The equation of a PID controller with back-calculation anti-windup is:

$$U(s) = K_pE(s) + (K_iE(s) + K_{aw}(U_{sat}(s) - U(s)))\frac{1}{s} + K_dE(s)\frac{s}{T_cs+1}$$

where $U_{sat}(s)$ is the command $U(s)$ saturated (always necessary to match the actuator's capability) and $E(s) = R(s) - Y(s)$.

The idea is to find the optimal combination of the PID parameters ($K_p, K_i, K_d, K_{aw}, T_c$) that minimises a cost function:

$$J = \sum_{i=0}^{N-1} W_e(r_i - y_i)^2 + \sum_{i=0}^{N-2} W_u(u_{sat_{i+1}} - u_{sat_i})^2 + W_u u_{sat_0}^2$$

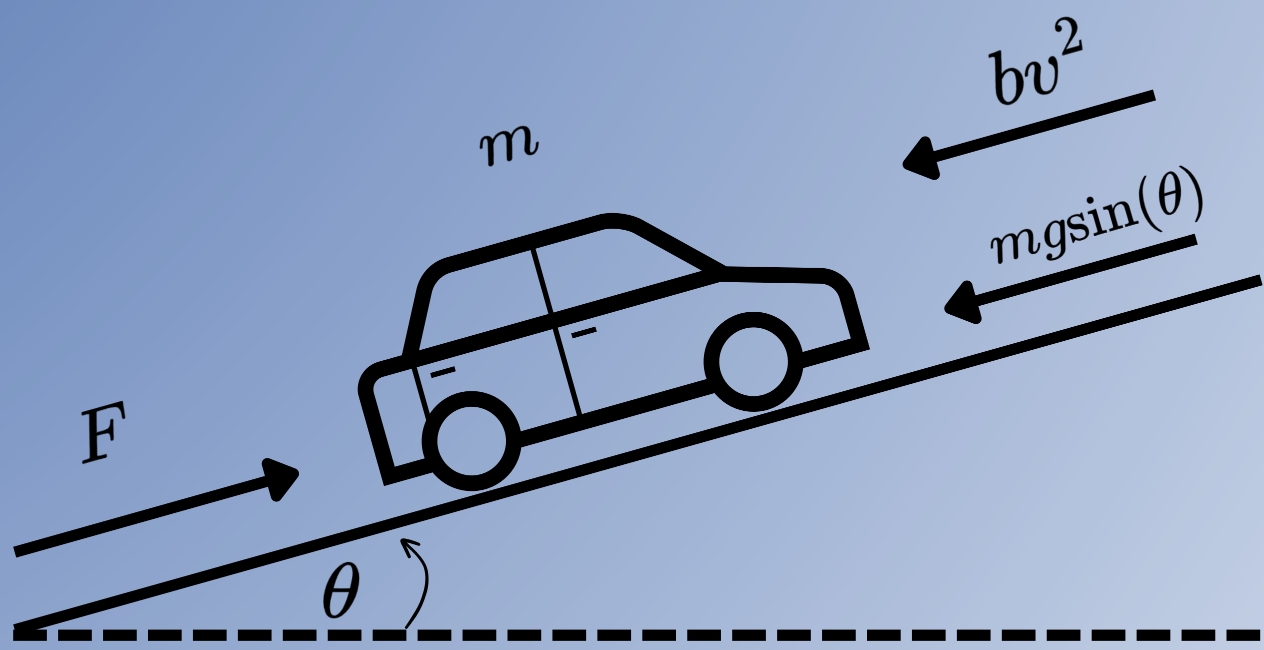
where:

r: setpoint
y: controlled variable
u_{sat}: saturated command
W_e: weighting coefficient related to the control error
W_u: weighting coefficient related to u_{sat}

Simone Bertoni

Car – model

The model is a car of mass m, pushed by a force F and subject to aerodynamic drag $-\frac{1}{2}C_dA\rho v^2$ and resistance force due to the slope. Where:



C_d: drag coefficient
A: reference area
ρ: air density
θ: slope angle
v: speed

The dynamic equation is:

$$m\frac{dv}{dt} = F - mg\sin(\theta) - \frac{1}{2}C_dA\rho v^2$$

The parameters used are taken from the Porsche Taycan Turbo (Wikipedia):

$$m = 2140\ kg$$
$$C_dA = 0.513\ m^2$$
$$\rho = 1.293\ \frac{kg}{m^3}$$

For convenience $\frac{1}{2}C_dA\rho$ is called b in the model.

Simone Bertoni

Car – max force

For the Porsche Taycan Turbo Wikipedia reports:

Max speed: 260 km/h
0 - 100 km/h: 3.2 s

Based on this info we can assume (approximation) that the maximum force that the electric powertrain can develop depends on the speed (intuitive).

We assume that the dependence is linear as shown in the figure.

From the max speed we have:

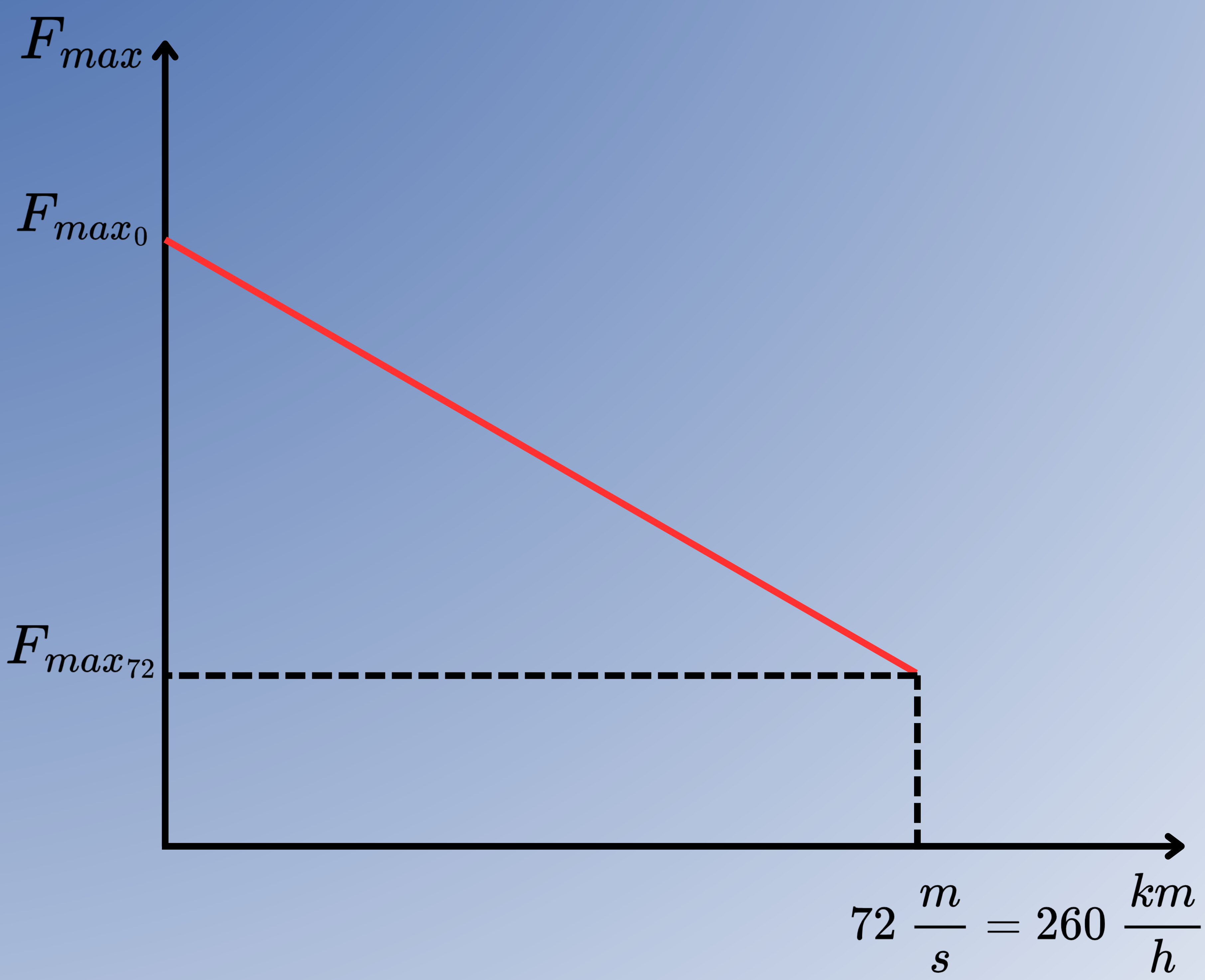
$$F_{max_{72}} = \frac{1}{2} C_d A \rho v_{max}^2 = 1710\ N$$

where $v_{max} = 72\ \frac{m}{s} = 260\ \frac{km}{h}$

Then we can simulate the model in Collimator to find F_{max_0} that gives 0-100 km/h in 3.2 s. This results in $F_{max_0} = 22000\ N$.

Simone Bertoni

Car - max force - plot



Simone Bertoni

Python – car model

```python
class Car:

    """ This class represents a car moving in 1D, subject to a throttle force F, with mass m,
        aerodynamic drag coefficient b, F_max/F_min forces, and time step T.
    """

    def __init__(self, m, b, F_max_0, F_max_max, v_max, g, T):
        self.m = m                      # Mass of the car
        self.b = b                      # Aerodynamic drag coefficient
        self.F_max_0 = F_max_0          # Max force applied to the car by the powertrain at 0 speed
        self.F_max_max = F_max_max      # Max force applied to the car by the powertrain at max speed
        self.v_max = v_max              # Max speed (m/s)
        self.T = T                      # Time step
        self.v = 0                      # Speed of the car
        self.g = g                      # Gravity (m/s^2)

    def Step(self, F, theta):

        """ Update the speed of the car based on the applied force F and the slope angle theta
        """
        # Max force applied by the powertrain depends on the speed
        v_to_F_max_x_axis = [0, self.v_max]
        F_max_y_axis = [self.F_max_0, self.F_max_max]

        if self.v < v_to_F_max_x_axis[0]:
            F_max = F_max_y_axis[0]
        elif self.v > v_to_F_max_x_axis[-1]:
            F_max = F_max_y_axis[-1]
        else:
            F_max = np.interp(self.v, v_to_F_max_x_axis, F_max_y_axis)

        # Saturate input force
        if F > F_max:
            F_sat = F_max

        elif F < 0:
            F_sat = 0
        else:
            F_sat = F

        # Calculate the derivative dv/dt using the input force and the car's speed and properties
        dv_dt = (F_sat - self.b*self.v*self.v - self.g*self.m*math.sin(theta))/self.m

        # Update the speed by integrating the derivative using the time step T
        self.v += dv_dt*self.T
```

Simone Bertoni

Python – PID

```python
class PID:

    """ This class implements a PID controller.
    """

    def __init__(self, Kp, Ki, Kd, Kaw, T_C, T, max, min, max_rate):
        self.Kp = Kp                # Proportional gain
        self.Ki = Ki                # Integral gain
        self.Kd = Kd                # Derivative gain
        self.Kaw = Kaw              # Anti-windup gain
        self.T_C = T_C              # Time constant for derivative filtering
        self.T = T                  # Time step
        self.max = max              # Maximum command
        self.min = min              # Minimum command
        self.max_rate = max_rate    # Maximum rate of change of the command
        self.integral = 0           # Integral term
        self.err_prev = 0           # Previous error
        self.deriv_prev = 0         # Previous derivative
        self.command_sat_prev = 0   # Previous saturated command
        self.command_prev = 0       # Previous command
        self.command_sat = 0        # Current saturated command

    def Step(self, measurement, setpoint):
        """ Execute a step of the PID controller.

        Inputs:
            measurement: current measurement of the process variable
            setpoint: desired value of the process variable
        """

        # Calculate error
        err = setpoint - measurement

        # Update integral term with anti-windup
        self.integral += self.Ki*err*self.T + self.Kaw*(self.command_sat_prev - self.command_prev)*self.T

        # Calculate filtered derivative
        deriv_filt = (err - self.err_prev + self.T_C*self.deriv_prev)/(self.T + self.T_C)
        self.err_prev = err
        self.deriv_prev = deriv_filt

        # Calculate command using PID equation
        command = self.Kp*err + self.integral + self.Kd*deriv_filt

        # Store previous command
        self.command_prev = command

        # Saturate command
        if command > self.max:
            self.command_sat = self.max
        elif command < self.min:
            self.command_sat = self.min
        else:
            self.command_sat = command

        # Apply rate limiter
        if self.command_sat > self.command_sat_prev + self.max_rate*self.T:
            self.command_sat = self.command_sat_prev + self.max_rate*self.T
        elif self.command_sat < self.command_sat_prev - self.max_rate*self.T:
            self.command_sat = self.command_sat_prev - self.max_rate*self.T

        # Store previous saturated command
        self.command_sat_prev = self.command_sat
```

Simone Bertoni

Python - Simulation & Cost function

```python
def Simulation(x, time_step, end_time, m, b, F_max_0, F_max_max, v_max, uphill):

    """ Simulate the PID control of a car with given parameters.

        Returns:
        (t, stp, z, command, theta): arrays of time, setpoints, positions, commands and slope angle
    """

    length = round(end_time/time_step)

    t = np.zeros(length)
    stp = np.zeros(length)
    v = np.zeros(length)
    command = np.zeros(length)
    theta = np.zeros(length)

    # A PI controller is considered - Kd and T_C are set = 0 - this is based on the knowledge that
    # for this problem a PI is sufficient
    [Kp, Ki, Kaw] = x
    Kd = 0
    T_C = 0

    # Initialize PID controller
    pid = PID(Kp, Ki, Kd, Kaw, T_C, time_step, F_max_0, 0, 300000)

    # Initialize car with given parameters
    car = Car(m, b, F_max_0, F_max_max, v_max, 9.81, time_step)

    # Iterate through time steps
    for idx in range(0, length):
        t[idx] = idx*time_step
        # Set setpoint
        stp[idx] = 42

        if t[idx] < end_time/3 or uphill == 0:
            theta[idx] = 0
        elif t[idx] < end_time*2/3:
            theta[idx] = 10*math.pi/180
        else:
            theta[idx] = 20*math.pi/180

        # Execute the control loop
        v[idx] = car.v
        pid.Step(v[idx], stp[idx])
        command[idx] = pid.command_sat
        car.Step(command[idx], theta[idx])

    return (t, stp, v, command, theta)

def Cost(x, time_step, end_time, m, b, F_max_0, F_max_max, v_max, We, Wu, uphill):
    """ Calculate the cost function for a given set of parameters.

        Inputs:
        x: PID parameters [Kp, Ki, Kd, Kaw, T_C]
        We: weight on control error
        Wu: weight on control effort

        Returns:
        cost: scalar value representing the total cost
    """

    # Simulate
    (t, stp, v, command, theta) = Simulation(x, time_step, end_time, m, b, F_max_0, F_max_max, v_max, uphill)

    # Cost function
    # J = sum((stp[i] - v[i])^2*t[i])*We + sum((command[i+1] - command[i])^2)*Wu + command[0]^2*Wu
    cost = np.sum(np.square(stp - v))*We + np.sum(np.square(np.diff(command)))*Wu + command[0]*command[0]*Wu

    return cost
```

Simone Bertoni

Python – Main – Optimisation

```python
def main():
    # -------- Configuration --------

    # Simulation parameters

    time_step = 0.1
    end_time = 60
    length = round(end_time/time_step)

    # Car parameters

    m = 2140
    b = 0.33
    F_max_0 = 22000
    F_max_max = 1710
    v_max = 72

    # Optimization weights for cost function

    We = [1, 1]
    Wu = [0.00002, 0.00002]
    uphill = [0, 1]

    # Initialize arrays for storing results

    t = np.zeros((length, len(We)+1))
    stp = np.zeros((length, len(We)+1))
    command = np.zeros((length, len(We)+1))
    theta = np.zeros((length, len(We)+1))
    v = np.zeros((length, len(We)+1))
    result = []

    # Perform minimization for each couple of We and Wu weights

    for idx in range(0, len(We)):
        bounds = ((0, None), (0, None), (0, None))
        r = minimize(Cost, [500, 3, 3], args=(time_step, end_time, m, b, F_max_0, F_max_max, v_max, We[idx], Wu[idx], uphill[idx]), bounds=bounds)
        result.append(r)

        # Print optimization results

        print("We = " + "{:.3g}".format(We[idx]) + " Wu = " + "{:.3g}".format(Wu[idx])  + " Uphill = " + "{:.0g}".format(uphill[idx]) + " Kp = " + "{:.3g}".format(result[idx].x[0])
            + " Ki = " + "{:.3g}".format(result[idx].x[1])  + " Kaw = " + "{:.3g}".format(result[idx].x[2]))
        print("Success: " + str(r.success))

        # Run simulation with optimized parameters
        (t[:, idx], stp[:, idx], v[:, idx], command[:, idx], theta[:, idx]) = Simulation(r.x, time_step, end_time, m, b, F_max_0, F_max_max, v_max, 1)

    # Run simulation with manual tuning
    x_man = [500, 3, 3]
    (t[:, idx+1], stp[:, idx+1], v[:, idx+1], command[:, idx+1], theta[:, idx+1]) = Simulation(x_man, time_step, end_time, m, b, F_max_0, F_max_max, v_max, 1)
```

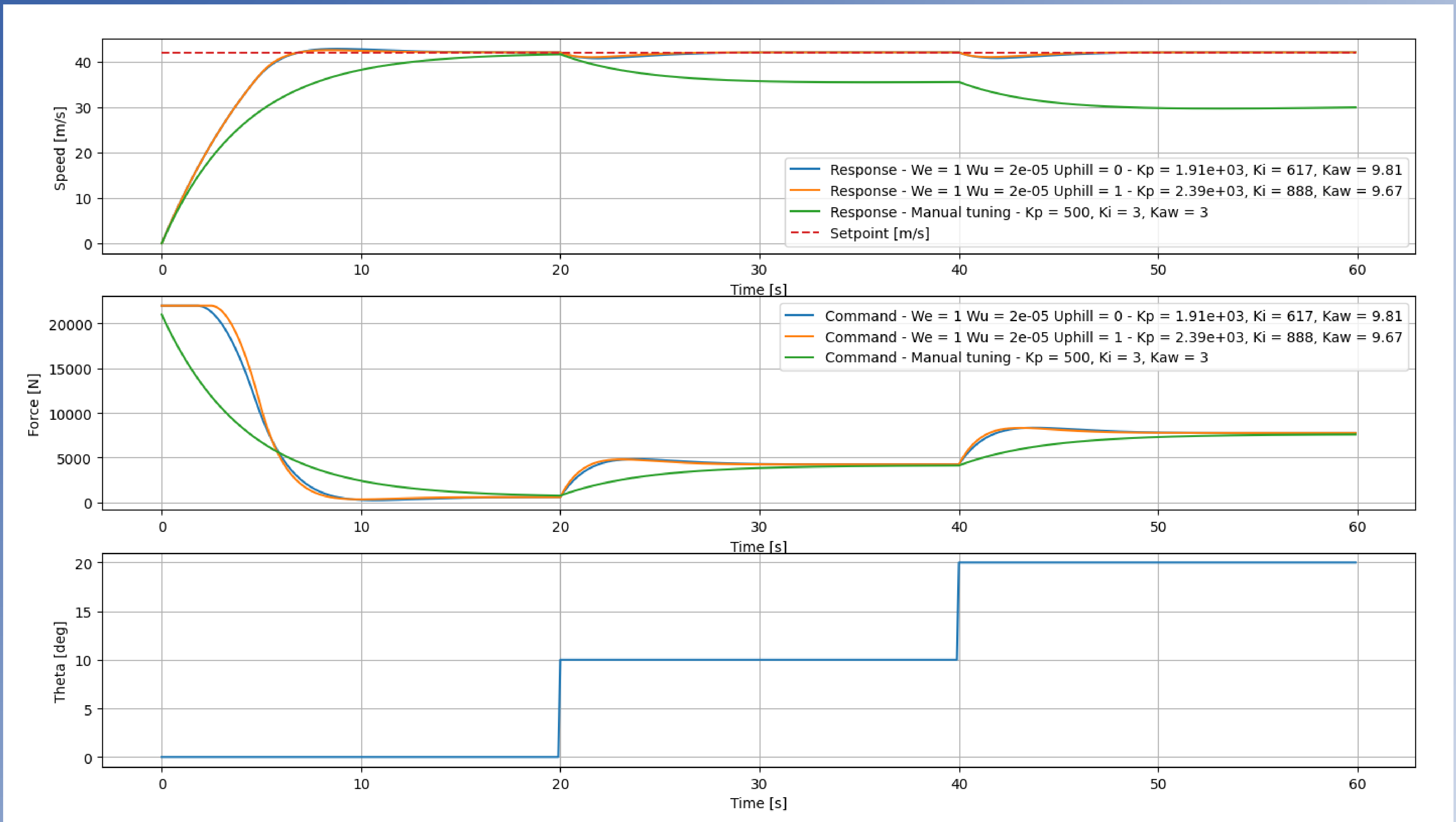
Simone Bertoni

Python – Main – Plot results

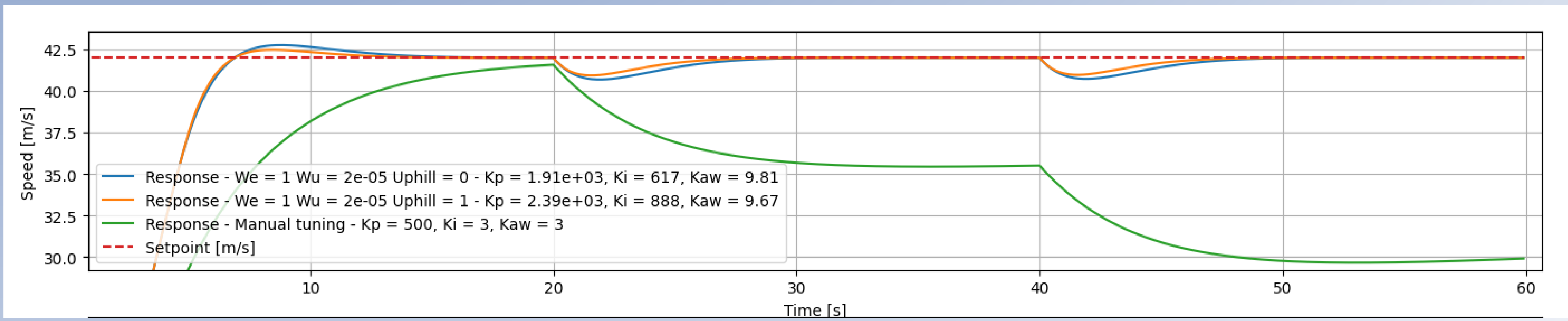
```python
# Plot speed response

plt.subplot(3, 1, 1)
for idx in range(0, len(We)):
    plt.plot(t[:,idx], v[:,idx], label="Response - We = " + "{:.3g}".format(We[idx]) + " Wu = " + "{:.3g}".format(Wu[idx])
            + " Uphill = " + "{:.0g}".format(uphill[idx]) + " - Kp = " + "{:.3g}".format(result[idx].x[0])
            + ", Ki = " + "{:.3g}".format(result[idx].x[1])  + ", Kaw = " + "{:.3g}".format(result[idx].x[2]))
plt.plot(t[:,idx+1], v[:,idx+1], label="Response - Manual tuning" + " - Kp = " + "{:.3g}".format(x_man[0])   + ", Ki = "
        + "{:.3g}".format(x_man[1])  + ", Kaw = " + "{:.3g}".format(x_man[2]))
plt.plot(t[:,0], stp[:,0], '--', label="Setpoint [m/s]")
plt.xlabel("Time [s]")
plt.ylabel("Speed [m/s]")
plt.legend()
plt.grid()

# Plot command force

plt.subplot(3, 1, 2)
for idx in range(0, len(We)):
    plt.plot(t[:,idx], command[:,idx], label="Command - We = " + "{:.3g}".format(We[idx]) + " Wu = " + "{:.3g}".format(Wu[idx])
            + " Uphill = " + "{:.0g}".format(uphill[idx]) + " - Kp = " + "{:.3g}".format(result[idx].x[0])
            + ", Ki = " + "{:.3g}".format(result[idx].x[1])  + ", Kaw = " + "{:.3g}".format(result[idx].x[2]))
plt.plot(t[:,idx+1], command[:,idx+1], label="Command - Manual tuning" + " - Kp = " + "{:.3g}".format(x_man[0])   + ", Ki = "
        + "{:.3g}".format(x_man[1])  + ", Kaw = " + "{:.3g}".format(x_man[2]))
plt.xlabel("Time [s]")
plt.ylabel("Force [N]")
plt.legend()
plt.grid()

# Plot theta

plt.subplot(3, 1, 3)
plt.plot(t[:,idx+1], theta[:,idx+1]*180/math.pi)
plt.xlabel("Time [s]")
plt.ylabel("Theta [deg]")
plt.grid()

# Display the plots

plt.show()
```

Simone Bertoni

Simulation result



Zoom

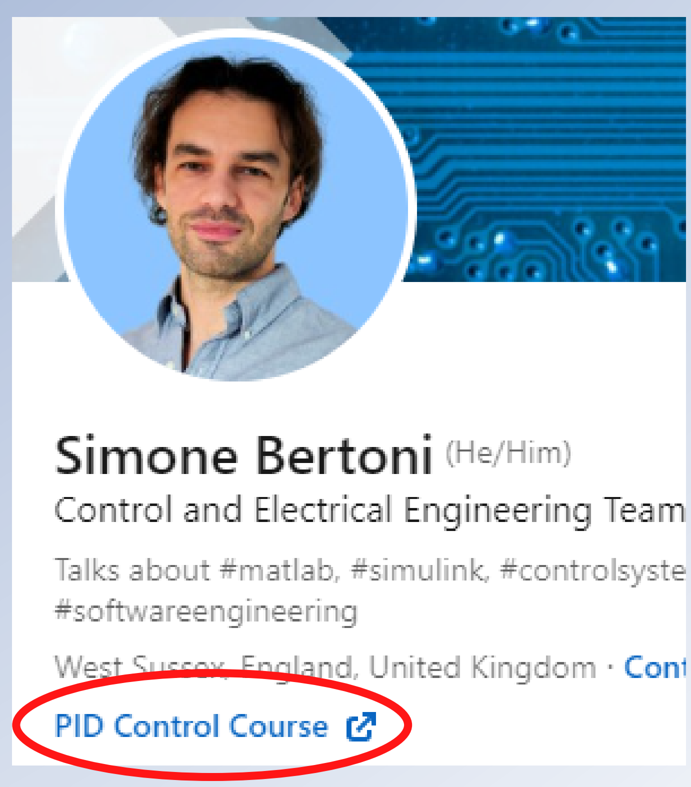


Simone Bertoni

PID Control

Interested in PID Control? Check out my digital course:

Find the link here!



Simone Bertoni