ME131 HW2

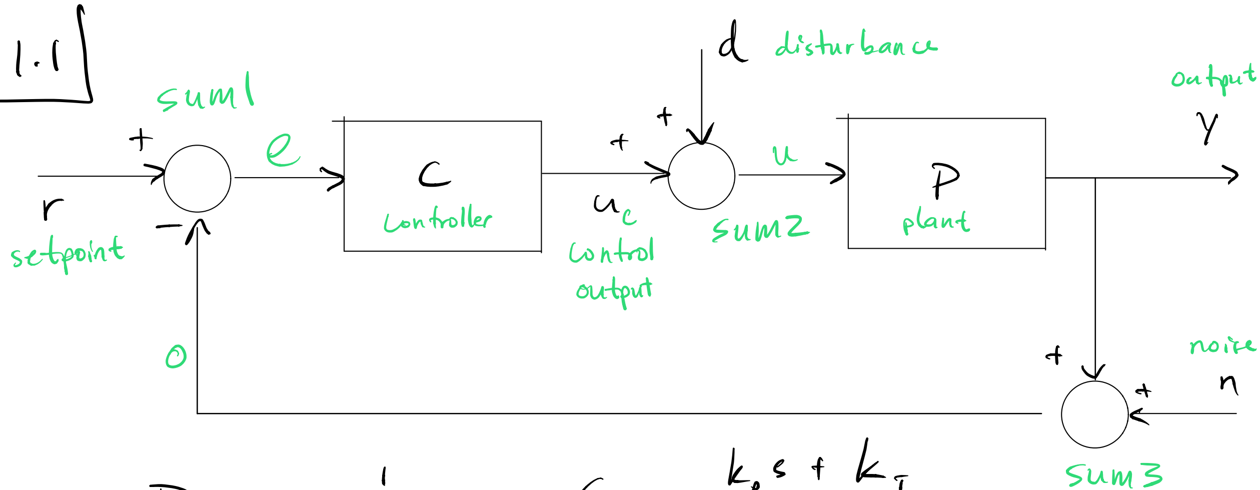
Due: 2/13/18

Trey Fortmuller
26037758

Problem 1: PI Controller Design

(the accompanying MATLAB scripts are available at github.com/treyfortmuller/me131)

1.1



$$P(s) = \frac{1}{s-1} \qquad C(s) = \frac{k_p s + k_I}{s}$$

let $k_p = 2$
$k_I = 1$

- first enter the controller & plant transfer functions in a MATLAB workspace.
- then build the model in Simulink using step, sine, LTI, summation, and scope blocks
- Then set the parameters of input, noise, and disturbance as given & run the simulation.

Steady state response of CL system output (w/ $\mu(t)$ the unit step input)

inputs			output y_{ss}
$r(t)$	$d(t)$	$n(t)$	$y_{ss}(t)$
$2\mu(t)$	$\mu(t)$	$3\mu(t)$	$y_{ss}(t) = -1$
$2\sin(5t)$	$\sin(t)$	$3\mu(t)$	$y_{ss}(t)$ is oscillatory about -3 $\forall\ t$

(step response scopes and simulink block diagram below).

1.2 Design k_p, k_I s.t. unit step forced response $G_{r \to y}(s)$ satisfies

- $\%OS \approx 10\%$
- 5% settling time $t_s \leq 20$ s
- Rise time $5 \leq t_r \leq 10$s

↳ use 2nd order approximation

Typical second order system: $$G(s) = \frac{\omega_n^2}{s^2 + 2\xi\omega_n s + \omega_n^2}$$

using MATLAB's "feeback" we get the CL transfer function $$H(s) = \frac{2s+1}{s^2+5+1} = \frac{k_p s + k_I}{s^2 + (k_p - 1)s + k_I}$$

$$H(s) \approx G(s) \text{ with } k_I = \omega_n^2 \quad k_p - 1 = 2\xi\omega_n$$

formulas

% overshoot: $$\%OS = 100\left(\frac{y_{peak} - y_{ss}}{y_{ss}}\right) \quad (1)$$

$$\xi = \frac{-\ln(\%OS/100)}{\sqrt{\pi^2 + \ln^2(\%OS/100)}} \quad (2)$$

settling time: $$T_s = -\frac{\ln\left(0.05\sqrt{1-\xi^2}\right)}{\xi\,\omega_n} \qquad (3)$$

rise time: $$T_r \approx \frac{0.8 + 1.1\xi + 1.4\xi^2}{\omega_n} \qquad (4)$$

we'll solve (2) for a desirable ξ:

$$\xi = \frac{-\ln\left(\frac{1}{10}\right)}{\sqrt{\pi^2 + \ln^2\left(\frac{1}{10}\right)}} = \boxed{0.59}$$

next we'll pick a rise time of 7.5s & find ω_n

$$\omega_n \approx \frac{0.8 + 1.1(0.59) + 1.4(0.59)}{7.5} = \boxed{0.303}$$

Now we'll just ensure these parameters satisfy the settling time constraint.

$$T_s = -\frac{\ln\left(0.05\sqrt{1-(0.59)^2}\right)}{(0.59)(0.303)} = 18 \text{ seconds}$$

so the settling time constraint is satisfied.

$$k_I = \omega_n^2 \quad k_p = 1 + 2\xi\omega_n$$

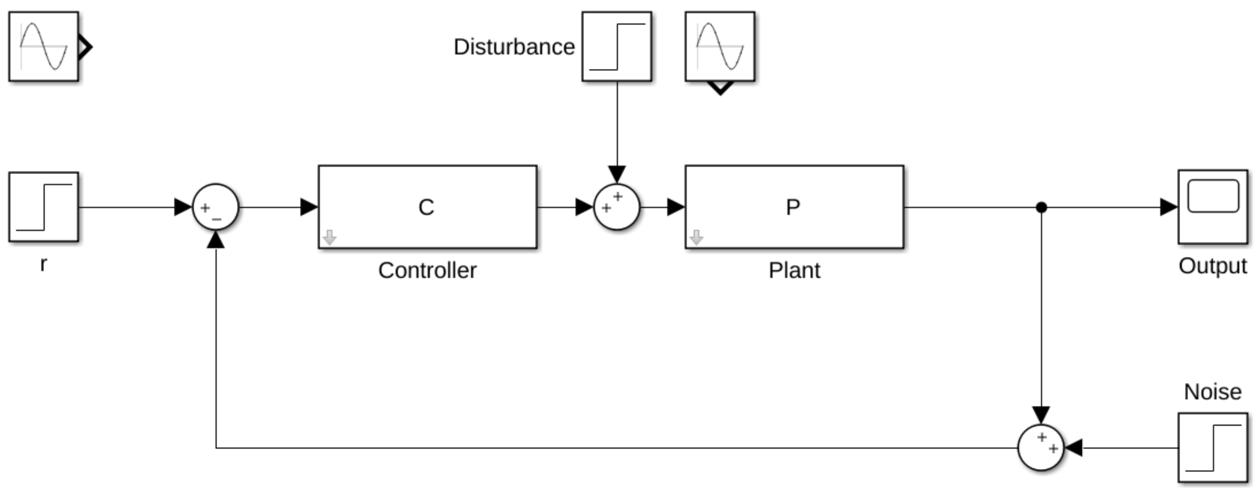
$$\Rightarrow \boxed{\begin{aligned} k_I &= 0.09 \\ k_p &= 1.36 \end{aligned}}$$

1.3) Write a MATLAB function implementing the controller in discrete time.

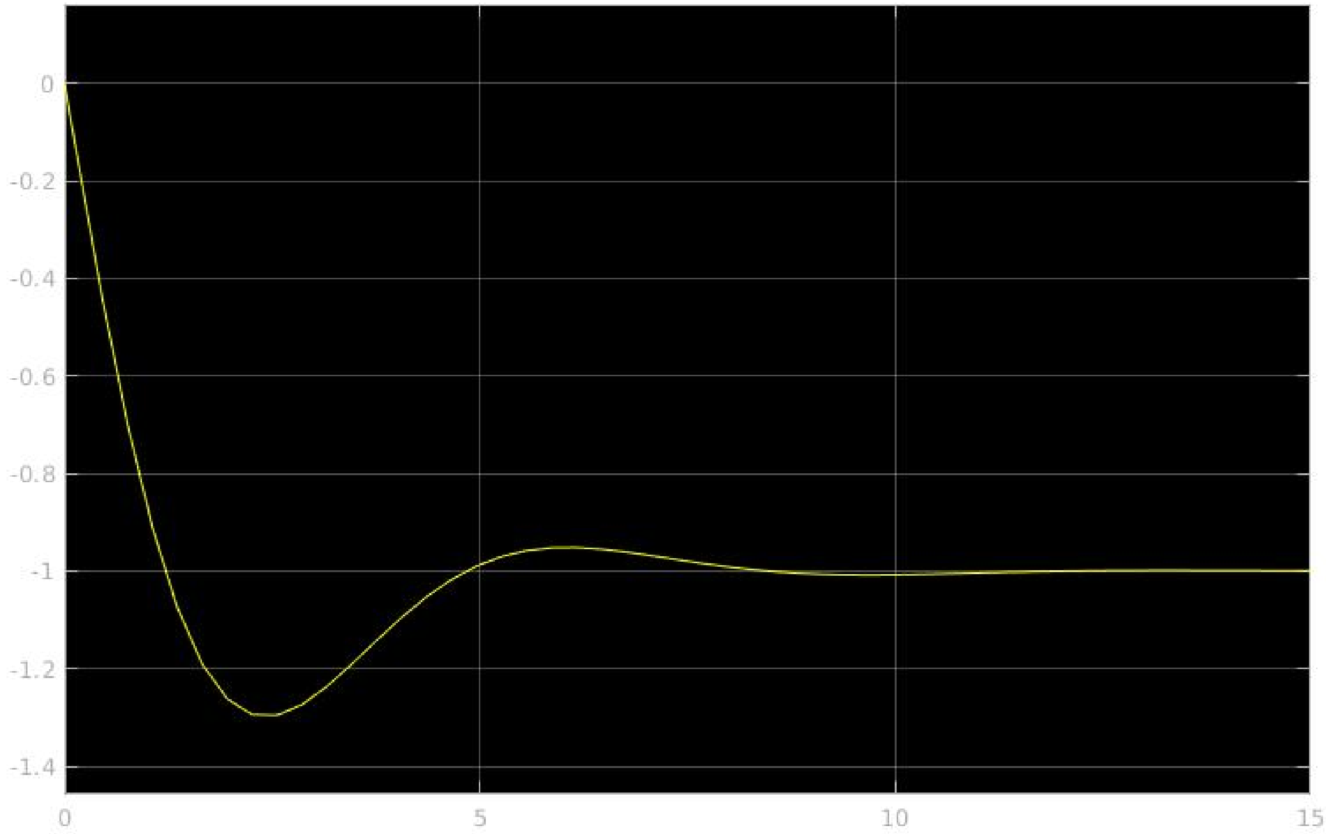
$$u = \text{controller}(e, \Delta t)$$

(submit the file)

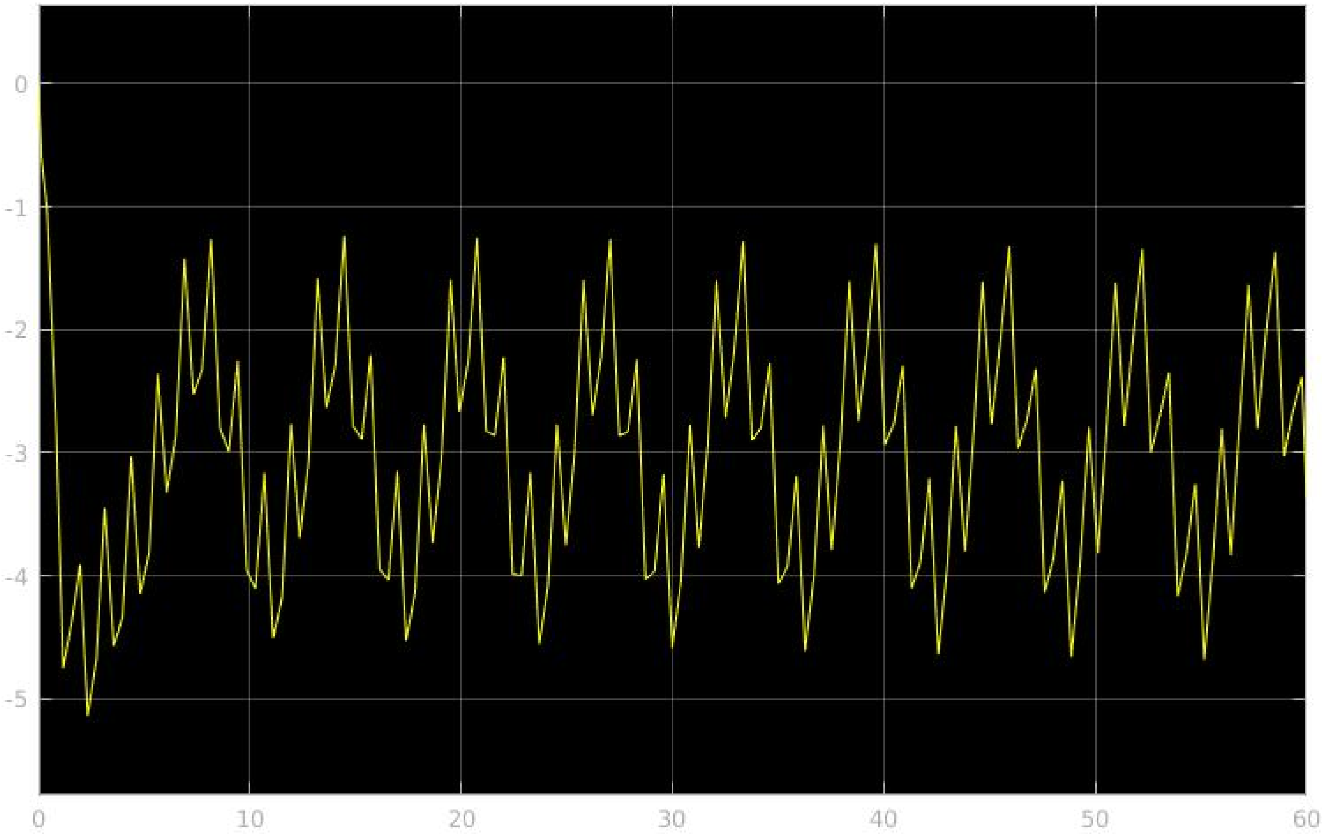
Simulink Block Diagram



Input case 1



Input case 2



Discrete PI Controller Function

`controller.m`

```
function u = controller(e, d_t)

    % control gains
    kp = 2;
    ki = 1;

    persistent integral
    if isempty(integral)
        integral = 0;
    end
    integral = integral + e*d_t;

    u = kp*e + ki*integral;
end
```

ME131 Lab 2 Deliverables

Simulating Kinematic Models in ROS

- Task 2 (c)
    - `bike_model.py` with the discretized forward Euler kinematic bicycle model implemented.

```
from numpy import sin, cos, tan, arctan, array, dot
from numpy import sign, argmin, sqrt, abs, pi
import rospy

def bikeFE(x, y, psi, v, a, d_f, a0,m, Ff, theta, ts):
    """
    process model
    """
    # external parameters
    l_f = 1.5
    l_r = 1.5
    g = 9.81

    # incline_rad = (theta*pi)/180

    # external forces calculation
    Fg = m*g*sin(theta)
    Fd = a0 * v**2
    F_ext = -Ff-Fd-Fg

    # compute slip angle
    beta = arctan((l_r / (l_r + l_f))*tan(d_f))

    # compute next state
    x_next = x + ts*(v*cos(psi + beta))
    y_next = y + ts*(v*sin(psi + beta))
    psi_next = (v / l_r)*sin(beta)
    v_next = v + ts*(a + (F_ext / m))

    return array([x_next, y_next, psi_next, v_next])
```

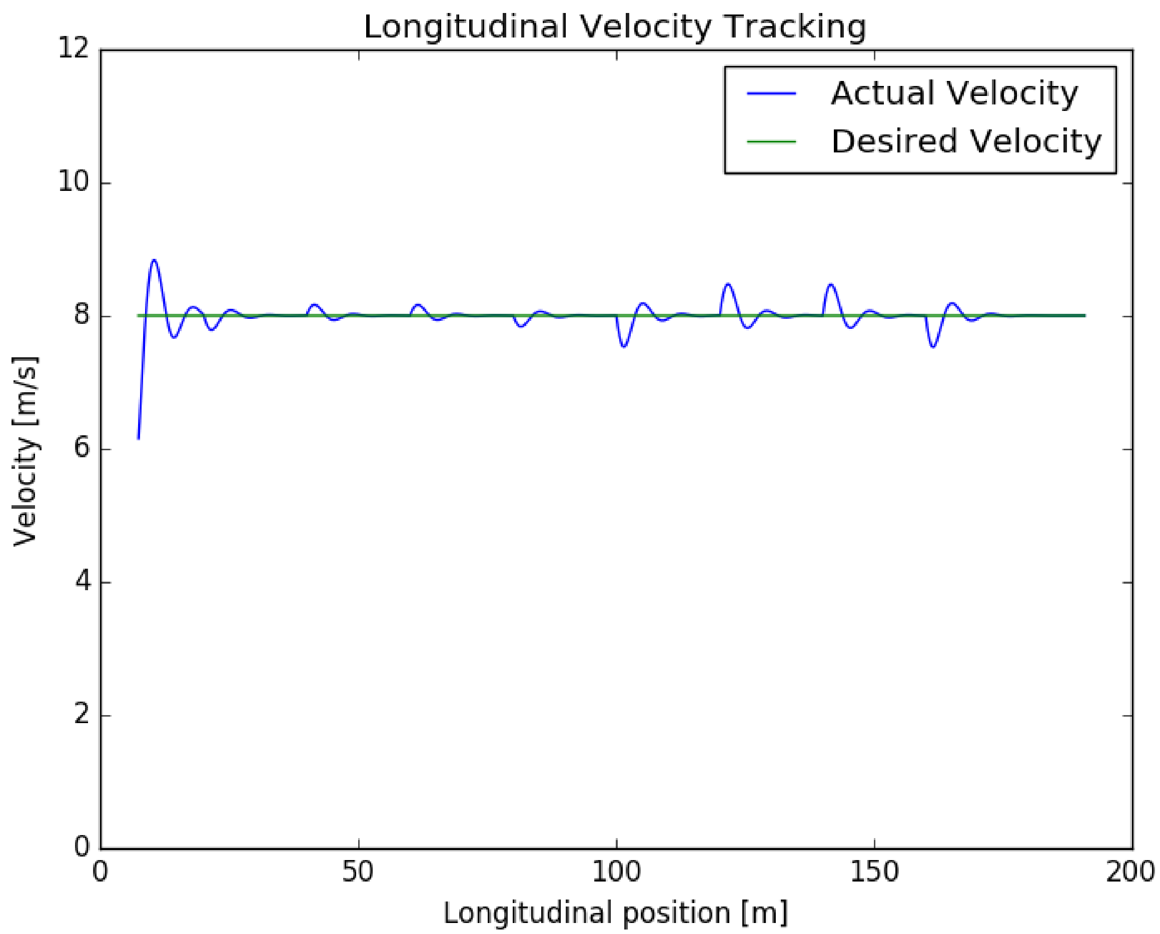
- Task 2 (d)
    - `controller.py` PID controller and its tuned gains.

```
#!/usr/bin/env python

import rospy
import time
from barc.msg import ECU
from labs.msg import Z_DynBkMdl
```

```
# initialize state
x = 0
y = 0
v_x = 0
v_y = 0

# ecu command update
def measurements_callback(data):
    global x, y, psi, v_x, v_y, psi_dot
    x = data.x
    y = data.y
    psi = data.psi
    v_x = data.v_x
    v_y = data.v_y
    psi_dot = data.psi_dot

# Insert your PID longitudinal controller here: since you are asked to do longitudinal
control,  the steering angle d_f can always be set to zero. Therefore, the control output
of your controller is essentially longitudinal acceleration acc.
# ==========PID longitudinal controller=========#
class PID():
    def __init__(self, kp=1, ki=1, kd=1):
        self.kp = kp
        self.ki = ki
        self.kd = kd
        self.integrator = 0
        self.error_prev = 0

    def acc_calculate(self, speed_reference, speed_current):

        self.integrator += self.error_prev
        error_current = speed_reference-speed_current

        acc = self.kp*(error_current) + self.ki*(self.integrator) + self.kd*
(error_current - self.error_prev)

        self.error_prev = error_current

        return acc
# ==========end of the controller==============#

# controller node
def controller():
    # initialize node
    rospy.init_node('controller', anonymous=True)

    # topic subscriptions / publications
    rospy.Subscriber('z_vhcl', Z_DynBkMdl, measurements_callback)
    state_pub = rospy.Publisher('ecu', ECU, queue_size = 10)

    # set node rate
    loop_rate = 50
```

```
    dt = 1.0 / loop_rate
    rate = rospy.Rate(loop_rate)
    t0 = time.time()

    # set initial conditions
    d_f = 0
    acc = 0

    # reference speed
    v_ref = 8 # reference speed is 8 m/s

    # Initialize the PID controller with your tuned gains
    PID_control = PID(kp=5, ki=1, kd=3)

    while not rospy.is_shutdown():
        # acceleration calculated from PID controller.
        acc = PID_control.acc_calculate(v_ref, v_x)

        # steering angle
        d_f = 0.0

        # publish information
        state_pub.publish( ECU(acc, d_f) )

        # wait
        rate.sleep()

if __name__ == '__main__':
    try:
        controller()
    except rospy.ROSInterruptException:
        pass
```

- Task 2 (j)
    - Submit a plot using the `plot.py` script after tuning the gains in the cruise controller



- Teleoperation via Keyboard
  - Keyboard control of the simulated vehicle:
    - https://drive.google.com/open?id=1UDEUumB0B6LlSTTGFk-k7HKpjTBvON5S
- `rqt_graph` during teleoperation

I used the teleop_twist_keyboard ROS package recommended in the lab instructions to interpret keyboard input, by default this node publishes to `/cmd_vel`. I left that node publishing to `/cmd_vel` and created my own `/barc_teleop` node which subscribes to `/cmd_vel`, takes the pertinent inputs, and maps then to the `/ecu` topic. The provided simulator node takes these messages and again remaps them appropriately for car viewer GUI.

- Teleoperation launch file
  - Note this launch file does not include the teleoperation node as we'll need that in its own terminal for keyboard input, I run `rosrun teleop_twist_keyboard teleop_twist_keyboard.py` to initialize the teleop_twist node.
  - It also does not include the car trajectory viewer as it requires python2 and I have configured my system to employ python3 by default, I run that node separately with `python2 view_car_trajectory.py`.

```
<launch>
    <!-- SYSTEM MODEL -->
```

```xml
    <!-- vehicle parameters -->
    <param name="mass" type="double" value="2000" />

    <!-- control parameters -->
    <param name="air_drag_coeff" type="double" value="0.01308" />
    <param name="friction" type="double" value="0.01711" />

    <!-- Keyboard Relay -->
    <node pkg="labs" type="car_teleop.py" name="barc_teleop" />

    <!-- Simulator -->
    <node pkg="labs" type="vehicle_simulator.py" name="simulator" />

</launch>
```