UNIVERSIDADE FEDERAL DE UBERLÂNDIA-UFU
FACULDADE DE ENGENHARIA MECÂNICA - FEMEC
ENGENHARIA MECATRÔNICA

SISTEMAS DIGITAIS PARA MECATRÔNICA

Trabalho Final 01 – Simulação de Drone 2D

Professor: Éder Alves de Moura

Engenharia Mecatrônica

GRUPO: MURILO MARCHI PEREIRA 11521EMT005
YURI LIMA ALMEIDA 11621EMT022

UBERLÂNDIA
2021

Sumário

1. Objetivos

Este trabalho tem como objetivo de implementar uma simulação de um sistema de controle em duas dimensões para um drone, utilizando a modelagem cinemática e dinâmica desenvolvida ao longo do semestre. O sistema deve conter dois tipos de ações: movimentação por *waypoints* e movimentação pelo teclado.

2. Introdução

Atualmente na engenharia é muito presente a necessidade de se controlar sistemas, desde sistemas simples, até sistemas complexos como em casos de máquinas e robôs utilizados nos diversos sistemas industriais, portanto é de extrema importância para os engenheiros estudar as várias teorias e métodos de controle de sistemas e como implementa-los em sistemas embarcados afim de criar equipamentos cada vez mais eficientes e autônomos.

Dentro das diversas formas de controle, uma das mais comuns é o controlador do tipo Proporcional-Integral-Derivativo (PID), que são muito utilizados devido à sua alta aplicabilidade na maioria dos sistemas de controle, simplicidade e, possibilidade de ser ajustado com técnicas baseadas na resposta experimental do sistema.

A lei de controle de um PID é dada por:

$$u(t) = K_p\left(e(t) + \frac{1}{T_i}\int_0^t e(t)dt + T_d\frac{de(t)}{dt}\right)$$

Onde *e(t)* é o erro de rastreamento (igual a diferença entre a entrada de referência e a saída do sistema); K_p é o ganho proporcional; T_i é o tempo integrativo e T_d é o tempo derivativo. Aplicando a transformada de Laplace na equação acima, obtém-se a seguinte função de transferência para o controlador:

$$C_{PID}(s) = \frac{U(s)}{E(s)} = K_p\left(1 + \frac{1}{T_i s} + T_d s\right)$$

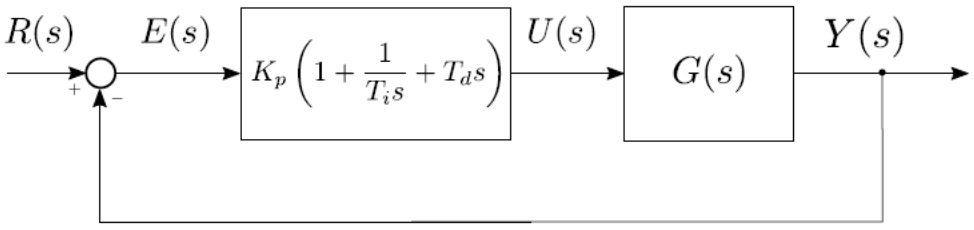

Figura 1: Diagrama de blocos de um sistema com controlador PID.

Para implementar um controlador PID em um sistema digital é preciso converte-lo do tempo continuo para o tempo discreto. Essas discretização pode ser feita por três métodos diferentes:

$$Backward\ Euler: s = \frac{1 - z^{-1}}{T_s} \equiv \frac{1}{T_s}\frac{z-1}{z}$$

$$Forward\ Euler: s = \frac{1}{T_s}\frac{1 - z^{-1}}{z^{-1}} \equiv \frac{z-1}{T_s}$$

$$Tusting\ (ou\ Bilinear): s = \frac{1}{T_s}\frac{1 - z^{-1}}{1 + z^{-1}} \equiv \frac{1}{T_s}\frac{z-1}{z+1}$$

Onde T_s é o período de amostragem do sistema. Utilizando o método de Tusting obtém-se a seguinte lei de controle do sistema a tempo discreto:

$$u(k) = \left[K_p e(k)\right] + \left[u(k-1) + K_i\frac{T}{2}(e(k) + e(k-1))\right] + \left[-u(k-1) + K_d\frac{2}{T}(e(k) - e(k-1))\right]$$

Um controlador PID possui três ações distintas: ação proporcional (K_p), ação integrativa (T_i) e, ação derivativa (T_d). Tipicamente se apenas a ação proporcional for utilizada, haverá um erro em regime estacionário que pode ser diminuto aumentando-se o ganho proporcional, porém conforme K_p é aumentado, o sistema tende a ficar cada vez mais oscilatório. Aplicando-se ação integral no controlador, o erro em regime estacionário para uma recendência do tipo degrau é eliminado, porém conforme T_i

diminui, o sistema tende a ficar oscilatório. Por fim, a ação derivativa permite aumentar o amortecimento do sistema, contudo o sistema tende a ficar oscilatório também de T_d for excessivamente elevado. Combinando essas três ações pode-se obter diferentes tipos de controladores: Proporcional (P); Proporcional-Integral (PI); Proporcional-Derivativo (PD); e o Proporcional-Integral-Derivativo (PID). A escolha do controlador deve ser tal que os requisitos de funcionamento do sistema sejam atingidos.

Este trabalho apresentará a implementação de um controlador PID na simulação em duas dimensões de voo de um drone do tipo *quadcopter*. O drone utilizado como modelo para a simulação é o DJI Mini 2, conforme foi modelado durante as aulas práticas da disciplina.

Figura 2: DJI Mini 2.

3. Desenvolvimento

A princípio foi escolhida pelo grupo como plataforma de desenvolvimento para o simulador o motor de jogo *Godot* devido à sua simplicidade, sua facilidade de integração entre os códigos e a interface gráfica do simulador, similaridade com Python, que já uma linguagem bastante conhecida e utilizada e, devido ao interesse dos membros do grupo de aprender o uso de uma nova ferramenta. Porém logo tornou-se aparente que implementar um controlador PID em *Godot* não seria tão trivial quanto inicialmente foi imaginada, portanto o grupo decidiu pela mudança da plataforma para o Python, que é uma linguagem a qual o grupo estava mais familiarizado.

O projeto é constituído por dois arquivos: *s_classes.py* e *simulator.py*. O arquivo *simulator.py* utiliza as seguintes bibliotecas: *pygame*; *datetime* e *math*. *Pygame* é uma biblioteca de jogos multiplataforma feita para ser utilizada em conjunto com a linguagem de programação Python e, no projeto é utilizada como motor gráfico para simulação do modelo. A biblioteca *datetime*, que possui ferramentas para se trabalhar com datas e horários, é utilizada para contar o tempo passado na simulação e saber quando será realizado o próximo calculo, ou seja, a atualização da atitude da aeronave bem como quando será o próximo frame e a biblioteca *math* é usada para fazer cálculos trigonométricos na simulação. O arquivo *s_classes.py* utiliza as mesas bibliotecas citadas anteriormente e também a biblioteca *random*, que contém diversas função para o trabalho com números aleatório, possibilitando o uso do o método Gauss para gerar uma distribuição gaussiana na soma com o erro da leitura do sensor.

No arquivo *s_classes.py* são definidas as classes do programa que posteriormente serão chamadas pelo arquivo *simulator.py*. São essas as classes: *Controller, Sensors; dSensor; drone; Ground, PID e AutoController.*

Figura 3: Classes do arquivo *s_classes.py*.

A classe *Controller* é utilizada para atualizar os coeficientes do controlador. A classe *Sensors* leva em consideração as posições nos eixos X, Y e a angulação do drone simulando sensores que estariam presentes no equipamento real. A classe *dSensor* calcula a diferença entre a leitura atual dos sensores e a leitura anterior A classe *drone* possui os métodos de aceleração, forças de empuxo e as características do veículo, sendo essas: sua massa, velocidade inicial, posição inicial e suas dimensões. A classe *Ground* define e desenha um chão para o ambiente da simulação. A classe *PID* aplica os coeficientes das ações proporcional, integral e derivativa, na lei de controle do sistema. A classe *AutoController* é onde são aplicadas as leis de controle do PID no modelo definido pela classe *drone* e também onde são mapeados a entrada dos comando do teclado. Abaixo é mostrado o código do arquivo *s_classes.py*:

```
import datetime
import pygame
import random
import math

# Classe para o controlador
class Controller(object):
    def update(self, drone):
        pass

# Classe para o sensor
class Sensor(object):
    def __init__(self, value, error=0):
        self.value = value
        self.error = error

    def set(self, value):
        self.value = value + random.gauss(0, self.error)

    def update(self, latest_measurement):
        pass

    def get(self):
        return self.value

# Classe que calcula a diferença de leitura dos sensores
class dSensor(Sensor):
    def __init__(self, value, error=0):
        Sensor.__init__(self, value, error)
        self.last = value

    def update(self, latest_measurement):
        self.set(latest_measurement - self.last)
        self.last = latest_measurement

# Classe que define os sensores do drone para leitura da velocidade e
rotação
class Sensors(object):
    def __init__(self, drone, base_error=0):
        self.drone = drone
        self.x_vel = dSensor(drone.pos[0], base_error)
        self.y_vel = dSensor(drone.pos[1], base_error)
        self.rot = dSensor(drone.rot, base_error)

    def update(self):
        self.x_vel.update(self.drone.pos[0])
        self.y_vel.update(self.drone.pos[1])
        self.rot.update(self.drone.rot)

# Classe que define o objeto drone
class drone(object):
    def __init__(self, pos, world, controller,
sensor_interface=Sensors, base_error=0):
        self.controller = controller
        self.world = world
        self.pos = pos
        self.rot = -math.pi / 2
        self.l_thrust = 0
        self.r_thrust = 0
        self.vel = [0, 0]
```

```
        self.max_thrust = 100
        self.min_thrust = 0
        self.mass = 1E3
        self.arm_length = 25
        self.sensors = sensor_interface(self, base_error)

    def set_thrust(self, left, right):
        self.l_thrust = max(min(left, self.max_thrust),
self.min_thrust)
        self.r_thrust = max(min(right, self.max_thrust),
self.min_thrust)

    def total_thrust(self):
        return [math.cos(self.rot) * (self.l_thrust + self.r_thrust),
                math.sin(self.rot) * (self.l_thrust + self.r_thrust)]

    def update(self):
        self.accelerate(self.total_thrust())
        nx, ny = self.pos[0] + self.vel[0], self.pos[1] + self.vel[1]
        if not self.world.check((nx, ny)):
            self.pos[0] = nx
            self.pos[1] = ny
        else:
            self.vel[0] *= 0.8
            self.vel[1] *= -0.5

        net_rot_thrust = (self.l_thrust - self.r_thrust) * 0.001
        ground_rot_thrust = 0.01
        if self.world.check(self.r_thruster()):
            net_rot_thrust -= ground_rot_thrust
        if self.world.check(self.l_thruster()):
            net_rot_thrust += ground_rot_thrust
        self.rot += net_rot_thrust

        self.sensors.update()
        self.controller.update(self)

    # Força no braço direito
    def r_thruster(self):
        cx, cy = self.pos
        cx += math.cos(self.rot + (math.pi / 2)) * self.arm_length
        cy += math.sin(self.rot + (math.pi / 2)) * self.arm_length
        return (cx, cy)

    # Força no braço esquerdo
    def l_thruster(self):
        cx, cy = self.pos
        cx += math.cos(self.rot - (math.pi / 2)) * self.arm_length
        cy += math.sin(self.rot - (math.pi / 2)) * self.arm_length
        return (cx, cy)

    # Altera as velocidades mediante as forças
    def accelerate(self, force):
        self.vel[0] += force[0] / self.mass
        self.vel[1] += force[1] / self.mass

    # Desenha o Drone
    def draw(self, dest):
        x = int(self.pos[0])
        y = int(self.pos[1])
```

```
        lx, ly = self.l_thruster()
        rx, ry = self.r_thruster()
        lx, ly = int(lx), int(ly)
        rx, ry = int(rx), int(ry)
        pygame.draw.line(dest, (255, 255, 255), (lx, ly), (rx, ry))
        pygame.draw.circle(dest, (255, 255, 255), (x, y), 4)

        self.controller.draw(dest)

# Classe para definir o chão
class Ground(object):
    def __init__(self):
        self.depth = 790

    def check(self, pos):
        return pos[1] >= self.depth

    def get_height(self, pos):
        return max(self.depth - pos[1], -1)

    def draw(self, dest):
        pygame.draw.line(dest, (255,255,255), (0, self.depth),
(dest.get_size()[0], self.depth), 2)

# Classe para o PID
class PID(object):
    def __init__(self, p, i, d):
        self.params = (p, i, d)
        self.last = 0
        self.integral = 0
        self.output = 0

    # Faz o update com os coeficientes
    def update(self, error):
        p, i, d = self.params
        self.integral += error
        delta = error - self.last
        self.last = error
        self.output = error * p + self.integral * i + delta * d
        return self.output

    # Calcula o erro de rastreamento
    def update_auto(self, actual, desired=0):
        error = desired - actual
        return self.update(error)

# Classe que define a ação de controle do PID
class AutoController(Controller):
    def __init__(self):
        Controller.__init__(self)
        self.desired_height = 100
        self.desired_x = -100

        self.estimated_x = 0
        self.height_estimate = 0
        self.rotation_estimate = -math.pi / 2

        self.drone = None

        self.height_pid = PID(0.05, 0, 3.5)
```

```
        self.x_pid = PID(0.05, 0, 2)

        self.yvel_pid = PID(10000, 0, 0)
        self.xvel_pid = PID(0.2, 0, 0)
        self.rot_pid = PID(50, 0, 0)

    def update(self, drone):
        self.drone = drone

        target_delta = 1
        # Mapeamento das teclas para controle manual do drone
        keys = pygame.key.get_pressed()
        if keys[pygame.K_a] or keys[pygame.K_LEFT]:
            self.desired x -= target delta
        if keys[pygame.K_d] or keys[pygame.K_RIGHT]:
            self.desired_x += target_delta
        if keys[pygame.K_w] or keys[pygame.K_UP]:
            self.desired_height += target_delta
        if keys[pygame.K_s] or keys[pygame.K_DOWN]:
            self.desired_height -= target_delta

        # Coleta informações dos sensores
        sensors = drone.sensors
        self.height_estimate -= sensors.y_vel.get() # velocidade
Vertical
        self.rotation_estimate += sensors.rot.get() # rotacao
        self.estimated_x += sensors.x_vel.get() # Velocidade
horizontal

        # Calcula o erro para x
        x_error = self.estimated_x - self.desired_x
        desired_xvel = -self.x_pid.update(x_error)

        # Calcula o erro para y
        height_error = self.height_estimate - self.desired_height
        desired_yvel = self.height_pid.update(height_error)

        # Calcula o erro para velocidade em x
        yvel_error = sensors.y_vel.get() - desired_yvel
        base_thrust = self.yvel_pid.update(yvel_error)

        # Define minimos e maximos para forças
        thrust_min = drone.max_thrust * 0.1
        thrust_max = drone.max_thrust * 0.8
        base_thrust = min(max(base_thrust, thrust_min), thrust_max)

        # Erro da velocidade x
        xvel_error = sensors.x_vel.get() - desired_xvel

        # Define minimos e maximos para rotacao
        desired_rot = (-math.pi / 2) -
self.xvel_pid.update(xvel_error)
        rot_min = (-math.pi / 2) - (math.pi / 4)
        rot max = (-math.pi / 2) + (math.pi / 4)
        desired_rot = min(max(desired_rot, rot_min), rot_max)
        rot_error = self.rotation_estimate - desired_rot
        left_thrust = -self.rot_pid.update(rot_error)
        right_thrust = -left_thrust

        l = base_thrust + left_thrust
```

```
        r = base_thrust + right_thrust

        drone.set_thrust(l, r)

    def draw(self, dest):
        x, y = int(self.drone.pos[0]), int(self.drone.pos[1])
        tx = int(dest.get_size()[0] / 2 + self.desired_x)
        ty = int(self.drone.world.depth - self.desired_height)
        r = 20
        colour = (0, 0, 0)

        pygame.draw.line(dest, colour, (tx-r, ty), (tx+r, ty))
        pygame.draw.line(dest, colour, (tx, ty-r), (tx, ty+r))
```

O arquivo *simulator.py* chama sua função main no início do programa onde são definidos o tamanho da tela, a quantidade de quadros e laços por segundo e são criados os objetos *controller*, *world*, *Drone* e, também é definida a fonte para o que for escrito na tela. Então é iniciado o laço que faz os cálculos da posição do drone de acordo com os comandos fornecidos pelo usuário. A partir desses comandos, o programa refaz os cálculos da posição e atualiza a tela do simulador de acordo, então o laço se repete enquanto o programa estiver aberto. Abaixo é mostrado o código do arquivo *simulator.py*.

```
import pygame
import datetime
import math


from s_classes import *

def main():

    # Inicia o pygame
    pygame.init()
    xl = 1200
    yl = 800
    # Define o tamanho da tela
    screen = pygame.display.set_mode((xl, yl))

    # Define quantidades de loops e frames por segundo
    lps = 100.0
    fps = 60.0

    # Define variáveis para periodo de calculo e periodo de frame
    calc_period = datetime.timedelta(seconds=1.0 / lps)
    next_calc = datetime.datetime.now()
    draw_period = datetime.timedelta(seconds=1.0 / fps)
    next_draw = datetime.datetime.now()

    gravity = 9.81

    # Cria objetos para controlador, ambiente de simulação e drone
```

```
    controller = AutoController()
    world = Ground()
    Drone = drone([400, world.depth], world, controller,
base_error=0.001)

    # define fonte como default para plot de dados na tela
    font = pygame.font.Font(None, 30)

    count = 0
    running = True
    # inicia o loop
    while running:
        # Lida com eventos de quit e clique do mouse
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                running = False

            if event.type == pygame.MOUSEBUTTONDOWN:
                # para o click do mouse pega a posição de x e y define
como posição desejada
                controller.desired_height = world.depth - event.pos[1]
                controller.desired_x = event.pos[0] -
(screen.get_size()[0] / 2)

        now = datetime.datetime.now()

        # Refaz o calculo para a posição
        if now >= next_calc:
            next_calc += calc_period

            Drone.vel[1] += 9.91 / lps
            Drone.update()


        # Replota tudo na tela
        if now >= next_draw:
            next_draw += draw_period

            text1 = f'''Velocidade: {Drone.vel[0]:.3f}, {-
Drone.vel[1]:.3f} [m/s]'''
            output1 = font.render(text1, True, (255, 255, 255))
            text2 = f'''Posição: {Drone.pos[0]:.3f}, {(yl -
Drone.pos[1]):.3f} [m]'''
            output2 = font.render(text2, True, (255, 255, 255))
            text4 = f'''Angulação: {Drone.rot*180/math.pi*(-1):.3f}
[graus]'''
            output4 = font.render(text4, True, (255, 255, 255))

            screen.fill((0,0,0))
            world.draw(screen)
            Drone.draw(screen)
            screen.blit(output1,(10 , 10))
            screen.blit(output2,(10 , 35))
            screen.blit(output4,(10 , 60))

            pygame.display.flip()

    pygame.quit()
```

```
if __name__ == "__main__":
    main()
```

O grupo tentou também implementar uma função para mostrar na tela gráficos referentes à posição e velocidade do drone em comparação com os comandos fornecidos ao sistema, porém notou-se que ao implementar essa função o simulador começou a apresentar atrasos muito altos, portanto essa função foi removida do programa.

4. Conclusão

No desenvolvimento deste trabalho ficou bastante clara a importância de se estudar teorias de controle de sistemas, como o PID e como implementa-las em sistemas embarcados. Também se destaca a versatilidade da linguagem de programação Python, que foi utilizada no projeto, mostrando sua capacidade e aplicabilidade em diferentes áreas distintas.

5. Referências bibliográficas

[1] - OGATA, K. ***Discrete-time Control Systems.*** 2a ed. Upper Saddle River: Prentice-Hall, 1994.

[2] - OGATA, K. **Engenharia de controle moderno.** 5a ed. São Paulo: Pearson, 2011. 824 p.

[3] - https://arxiv.org/pdf/2106.15134.pdf

[4] - https://github.com/tristeng/control/blob/master/notebooks/quadcopter-2d.ipynb