Estabilidade

November 3, 2021

[ ]:
```
import numpy as np
from numpy import genfromtxt
from rocketpy import Function
from data import *
from control.matlab import *
plt.style.use('seaborn')
```

0.1 Calculo de propriedades

[ ]:
```
class Canard:
    def __init__(self, n, span, rootChord, tipChord, radius, airfoil):
        self.n = n
        self.span = span
        self.rootChord = rootChord
        self.tipChord = tipChord
        self.radius = radius
        self.airfoil = airfoil

        self.area = np.pi * radius**2
        self.addFins()

    def addFins(self):
        # Retrieve parameters for calculations
        Af = (self.rootChord + self.tipChord) * self.span / 2  # fin area
        AR = 2 * (self.span ** 2) / Af  # Aspeself.tipChord ratio
        gamac = np.arctan((self.rootChord - self.tipChord) / (2 * self.span)) ␣
↪# mid chord angle

        # Import the lift curve as a funself.tipChordion of lift values by␣
↪attack angle
        read = genfromtxt(self.airfoil, delimiter=",")
        cnalfa0 = Function(read, extrapolation="natural").differentiate(0,␣
↪1e-01)

        # Calculate clalpha
        FD = 2 * np.pi * AR / (cnalfa0 * np.cos(gamac))
        clalpha = (
```

```
            cnalfa0
            * FD
            * (Af / self.area)
            * np.cos(gamac)
            / (2 + FD * (1 + (4 / FD ** 2)) ** 0.5)
        )

        # Aplies number of fins to lift coefficient
        clalpha *= self.n / 2

        # Fin-body interference correself.tipChordion
        clalpha *= 1 + self.radius / (self.span + self.radius)

        # self.rootChordeate a funself.tipChordion of lift values by attack␣
↪angle
        cldata = Function(
            lambda x: clalpha * x, "Alpha (rad)", "Lift coeficient (Cl)",␣
↪interpolation="linear"
        )

        # Save cldata
        self.cldata = cldata

        # Calculate roll forcing properties
        Ymac = self.radius + self.span / 3 * (self.rootChord + 2 * self.
↪tipChord) / (self.rootChord + self.tipChord)
        clf_delta = clalpha * Ymac / (2 * self.radius)

        clfdata = Function(
            lambda delta: clf_delta * delta, "Delta (rad)", "Roll forcing␣
↪coeficient (Clf)", interpolation="linear"
        )

        # Save clfdata
        self.clfdata = clfdata

        # Calculate roll damping properties
        b1 = (Ymac /  2) * (self.radius**2) * self.span
        b2 = ((self.rootChord + 2 * self.tipChord) / 3) * self.radius * (self.
↪span**2)
        b3 = ((self.rootChord + 3 * self.tipChord) / 12) * (self.span**3)
        trapezoidal_constant_fins = b1 + b2 + b3

        cld_wv = self.n * cnalfa0 * trapezoidal_constant_fins / (self.area * 2␣
↪* self.radius)
        self.cld_data = lambda w, v: cld_wv * w / v
```

0.2 Definindo aletas e canards

[ ]:
```
# Dados das aletas
n = 3
span = 0.077
rootChord = 0.058
tipChord = 0.018
airfoil = 'NACA0012 curva Completa.txt'
angulo_maximo_de_abertura = 3 * np.pi / 180

# Dados das canards
n_canard = 2
span_canard = 0.06
rootChord_canard = 0.035
tipChord_canard = 0.035
airfoil_canard = 'NACA0012 curva Completa.txt'
angulo_maximo_de_abertura_canard = 7 * np.pi / 180

# Dados do foguete
radius = 80.9/2000
J = 0.007
Ar = np.pi * radius**2 # Área de referencia
Lr = 2 * radius # Comprimento de referencia

# Dados do ambiente ao redor
rho = 1.06 # air density
velocidade = 0.3 * 343 # Mach 0.5
DynamicPressure = velocidade**2 * rho / 2

# Criando objeto correspondente à aleta
aleta = Canard(n, span, rootChord, tipChord, radius, airfoil)
canard = Canard(n_canard, span_canard, rootChord_canard, tipChord_canard,␣
↪radius, airfoil)
```

0.3 Criando as funções de transferência

[ ]:
```
# Parametros das aletas
forcing_aletas_coef = aleta.clfdata.differentiate(0)
damping_aletas_coef = aleta.cld_data(1, velocidade) - aleta.cld_data(0,␣
↪velocidade)

# Parametros das canards
forcing_canard_coef = canard.clfdata.differentiate(0)
damping_canard_coef = canard.cld_data(1, velocidade) - canard.cld_data(0,␣
↪velocidade)

s = tf([1, 0], 1)
```

```python
# Planta
Gp = 1 / (J * s + (damping_aletas_coef + damping_canard_coef) * DynamicPressure␣
↪* Ar * Lr)

# Servo
tau_s = 0.07 / (np.pi/3)
Gs = 1 / (tau_s * s + 1)

# Controlador
Kp = 3
Ki = 1
Kd = 0.01
Gc = Kp + Ki / s + Kd * s

# Sensoreamento
H = 1

# Função de transferência total
G = feedback(Gc * Gs * Gp * forcing_canard_coef, H)
G
```

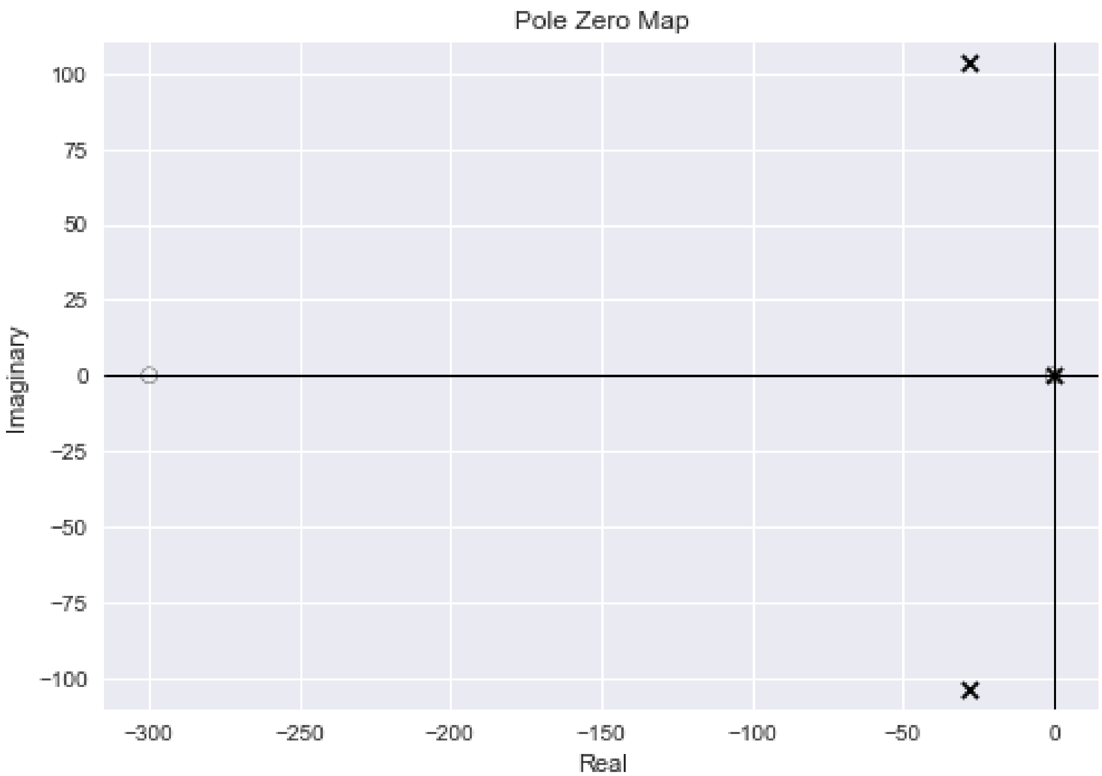
[ ]:

$$\frac{0.01805s^2 + 5.414s + 1.805}{0.0004679s^3 + 0.02668s^2 + 5.438s + 1.805}$$

0.3.1 Mapa de Polos e Zeros

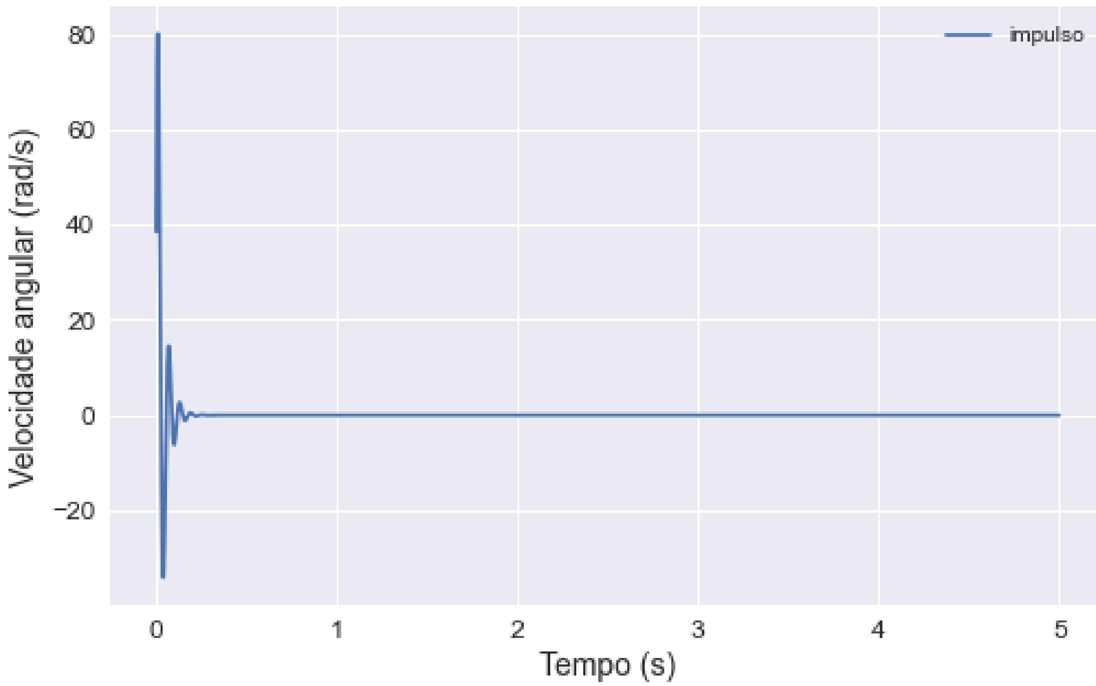
[ ]:
```python
pzmap(G)
```

[ ]:
```
(array([-28.34521+103.92353j, -28.34521-103.92353j,  -0.33237  +0.j      ]),
 array([-299.6663,   -0.3337]))
```



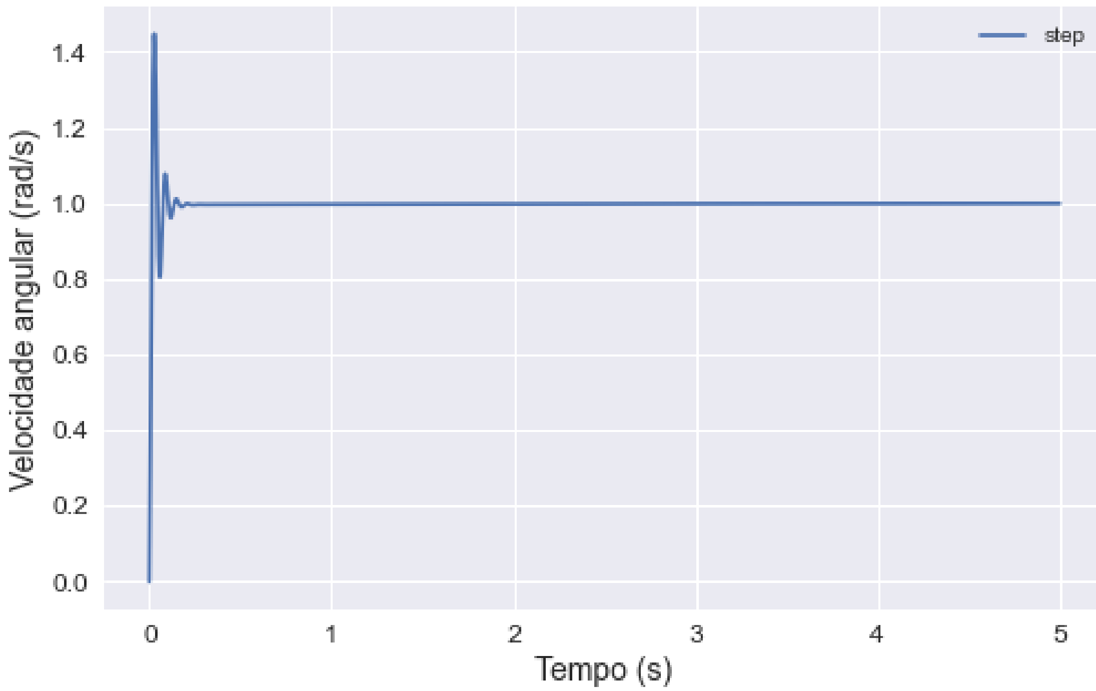
0.3.2 Resposta à uma entrada impulso

[ ]:
```
t = np.linspace(0, 5, 1001)
yi, xi = impulse(G, t)
fi = Data(xi, yi, 'Tempo (s)', 'Velocidade angular (rad/s)', 'impulso',␣
 ↪method='cubicSpline')
fi.plot2D(style='matplotlib')
```



0.3.3 Resposta à uma entrada degrau

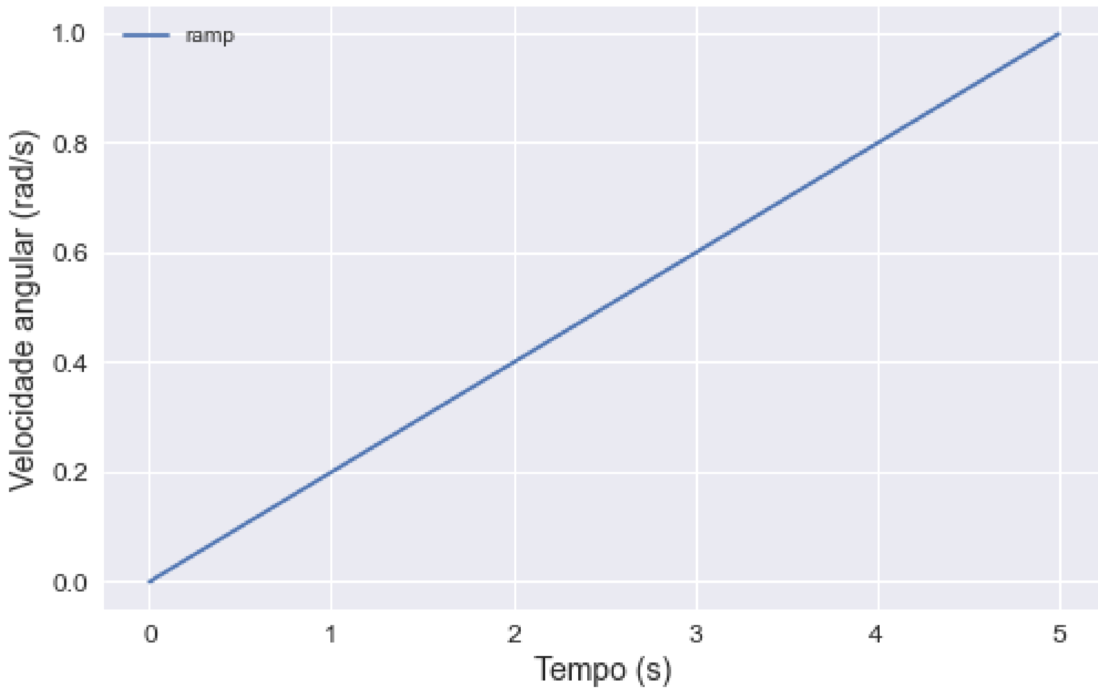
[ ]:
```
ys, xs = step(G, t)
fs = Data(xs, ys, 'Tempo (s)', 'Velocidade angular (rad/s)', 'step',␣
↪method='cubicSpline')
fs.plot2D(style='matplotlib')
```



0.3.4 Resposta à uma entrada rampa

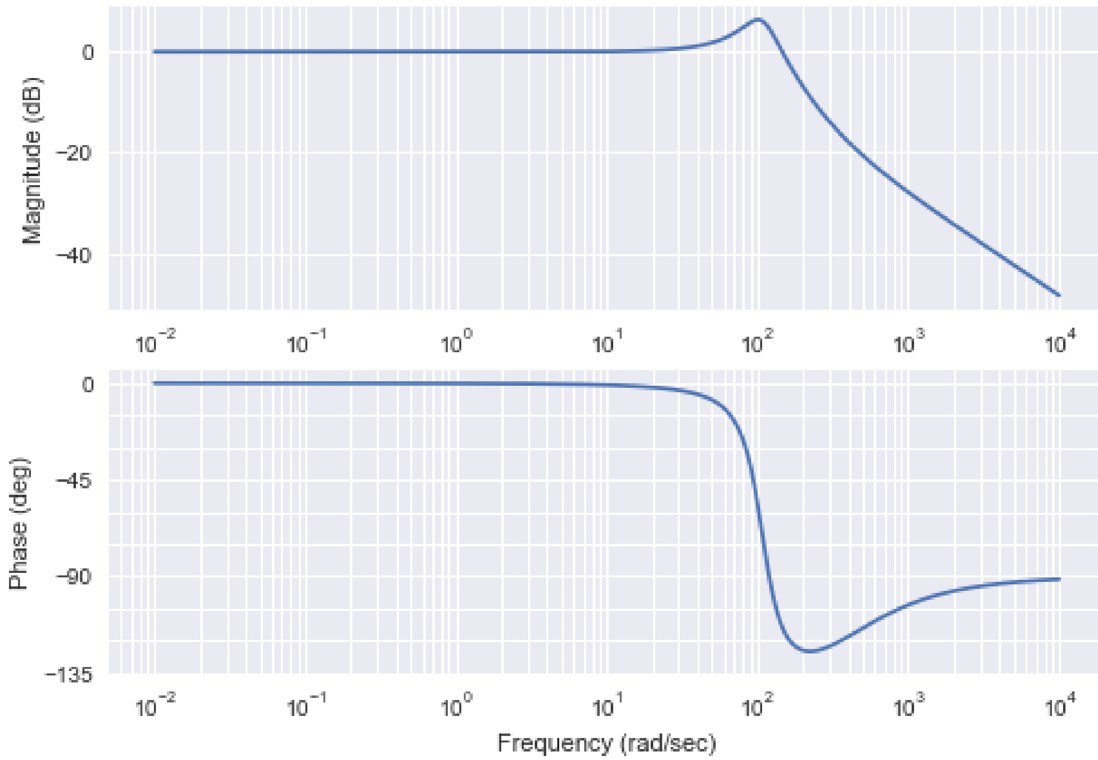
[ ]:
```
ramp = np.array([t[i] / max(t) for i in range(len(t))])
yr = lsim(G, ramp, t)[0]
fr = Data(t, yr, 'Tempo (s)', 'Velocidade angular (rad/s)', 'ramp',␣
↪method='cubicSpline')
fr.plot2D(style='matplotlib')
```

```
C:\Users\bruno\anaconda3\lib\site-packages\control\timeresp.py:293: UserWarning:
return_x specified for a transfer function system. Internal conversion to state
space used; results may meaningless.
  warnings.warn(
```



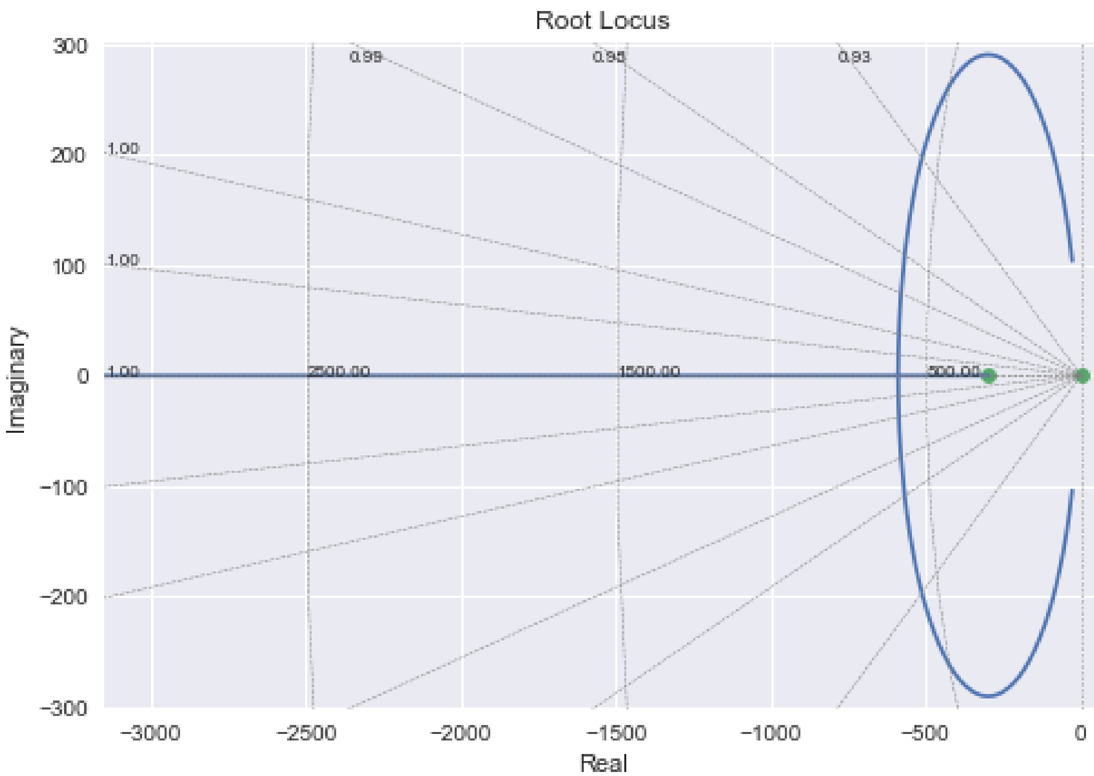
0.4 Bode

[ ]:
```
mag, phase, omega = bode(G)
```



0.5 Root Locus

[ ]:
```
r, k = rlocus(G)
```


Root Locus
Imaginary
Real
300
200
100
0
−100
−200
−300
−3000
−2500
−2000
−1500
−1000
−500
0
0.99
0.95
0.93
1.00
1.00
1.00
2500.00
1500.00
500.00

[ ]: