e-test-site-data-analysis-template

August 28, 2024

[5]:
```
import os
import math
import datetime
import numpy as np
import pandas as pd
from math import pi
import scipy.integrate as spi
from scipy.integrate import quad
from matplotlib import pyplot as plt
from scipy.signal import savgol_filter
from scipy.ndimage import median_filter
from scipy.ndimage import gaussian_filter1d
from scipy.interpolate import UnivariateSpline
from sklearn.linear_model import LinearRegression
from matplotlib.ticker import (AutoMinorLocator, MultipleLocator)
```

0.1 1. Define Constants

Define the constants specific to this coldflow test

[ ]:
```
# nitrous constants
p_c = 7251                                                              # kPa
T_c = 309.57                                                            # Kelvin
rho_c = 452                                                             # kg/m^3

# 4.1 constants
b1 = -6.71893
b2 = 1.35966
b3 = -1.3779
b4 = -4.051

# 4.2 constants
b1_l = 1.72328
b2_l = -0.83950
b3_l = 0.51060
b4_l = -0.10412

# 4.3 constants
```

```
b1_g = -1.00900
b2_g = -6.28792
b3_g = 7.50332
b4_g = -7.90463
b5_g = 0.629427
```

[ ]:
```
# run tank dimensions
inner_diameter = 0.146304                                           # m
cyl_height = 0.912114                                               # m
V_total = 0.01533                                                   # m^3
base_area = np.pi * ((inner_diameter/2)**2)                         # m^2
```

[ ]:
```
# injector
hole_diameter = 2                                                   # mm
num_holes = ???
```

[ ]:
```
# nozzle
A = 0.0011
gamma = 1.2661
R = 375.5075                                                        # specific gas␣
↪constant of mixture
T_o = 2842                                                          # K
```

[ ]:
```
# fuel grain
m_fuel = ???                                                      # kg
m_fuel_uncertainty = 0.0005                                         # kg
```

[ ]:
```
# input filename
input_data = '???.txt'
```

[ ]:
```
# ambient temp
outdoor_temp = ???                                                   # Celcius
```

0.2 2. Define Functions

These are the functions required to run the code. Add any additional functions in this section.

[1]:
```
# get data in time range required
def get_data_range(data, time, start, end):
        return [d for t, d in zip(time, data) if start <= t <= end]
```

[2]:
```
# calculate total impulse by numerically integrating the thrust values with␣
↪respect to time
def calculate_total_impulse(time, thrust):
    def integrand(t):
        return np.interp(t, time, thrust)
    total_impulse = 0
```

```python
    for i in range(len(time) - 1):
        total_impulse += spi.quad(integrand, time[i], time[i+1])[0]
    return total_impulse
```

[3]:
```python
# determine nitrous vapour pressure, liquid density and gas density given␣
↪temperature
def nitrous_equations(temp_kelvin):
    T_r = temp_kelvin/T_c

    # 4.1 Vapour Pressure
    p = p_c * math.exp((1/T_r) * ((b1 * (1-T_r)) + (b2 * (1-T_r)**(3/2)) + (b3␣
↪* (1-T_r)**(5/2)) + (b4 * (1-T_r)**5)))

    # 4.2 Density of Saturated Liquid
    p_l = rho_c * math.exp((b1_l * (1-T_r)**(1/3)) + (b2_l * (1-T_r)**(2/3)) +␣
↪(b3_l * (1-T_r)) + (b4_l * (1-T_r)**(4/3)))

    # 4.3 Density of Saturated Gas
    p_g = rho_c * math.exp((b1_g * ((1/T_r)-1)**(1/3)) + (b2_g * ((1/
↪T_r)-1)**(2/3)) + (b3_g * ((1/T_r)-1)) + (b4_g * ((1/T_r)-1)**(4/3)) + (b5_g␣
↪* ((1/T_r)-1)**(5/3)))

    return p, p_l, p_g

# turn nitrous vapour pressure, liquid density and gas density into lists
def nitrous_equation_lists(temp_list):
    vap_list = []
    liquid_list = []
    gas_list = []
    for temp_kelvin in temp_list:
        p, p_l, p_g = nitrous_equations(temp_kelvin)
        vap_list.append(p)
        liquid_list.append(p_l)
        gas_list.append(p_g)

    return vap_list, liquid_list, gas_list
```

[4]:
```python
# Determine liquid density from temperature. Finds key from closest value in␣
↪dictionary using list comprehension
def get_liquid_density_from_pressure(temp_liquid_density_dict,␣
↪pressure_temp_dict, pressure_key):
    pressure_KPa = pressure_key * 6.89476
    temp_K = pressure_temp_dict.get(pressure_KPa) or␣
↪pressure_temp_dict[min(pressure_temp_dict.keys(), key = lambda key:␣
↪abs(key-pressure_KPa))]
```

```
    liquid_density = temp_liquid_density_dict.get(temp_K) or␣
↪temp_liquid_density_dict[min(temp_liquid_density_dict.keys(), key = lambda␣
↪key: abs(key-temp_K))]

    return liquid_density
```

[ ]:
```
# determine discharge coefficient given mass flow rate and combustion chamber␣
↪pressure
def calculate_discharge_coefficient(n_o, hole_diameter, num_holes, p_run_range,␣
↪run_temp_range, delta_p_run_cc, temp_liquid_density_dict):
    injection_area = (np.pi * ((hole_diameter/1000)/2)**2)
    discharge_coefficient = []
    for i in range(len(p_run_range)):
        liquid_density =␣
↪get_liquid_density_from_pressure(temp_liquid_density_dict,␣
↪pressure_temp_dict, p_run_range[i])
        discharge_coeff = n_o / (num_holes * injection_area * np.
↪sqrt(2*liquid_density*delta_p_run_cc[i]*6892.76))
        discharge_coefficient.append(discharge_coeff)

    return discharge_coefficient

def calculate_discharge_coefficient_uncertainty(discharge_coefficient,␣
↪hole_diameter, num_holes, p_run_range, delta_p_run_cc,␣
↪temp_liquid_density_dict, mass_flow_rate_std_error):
    injection_area = (np.pi * ((hole_diameter/1000)/2)**2)
    c_d_std_error = []
    for i in range(len(p_run_range)):
        liquid_density =␣
↪get_liquid_density_from_pressure(temp_liquid_density_dict,␣
↪pressure_temp_dict, p_run_range[i])
        discharge_coefficient_std_error = (1 / (num_holes * injection_area * np.
↪sqrt(2*liquid_density*abs(delta_p_run_cc[i])*6892.76))) *␣
↪mass_flow_rate_std_error
        c_d_std_error.append(discharge_coefficient_std_error)
    return c_d_std_error
```

[ ]:
```
def get_nitrous_mass_unknown_ullage(pressure, m_total):

    temp_K = pressure_temp_dict.get(pressure) or␣
↪pressure_temp_dict[min(pressure_temp_dict.keys(), key = lambda key:␣
↪abs(key-pressure))]
    p, p_liq, p_gas = nitrous_equations(temp_K)
    x = (V_total-((m_total/p_gas)))/((m_total/p_liq)-(m_total/p_gas))      # x␣
↪= mass of liquid / total mass
```

```
    m_liq = x * m_total                                                   #␣
↪get mass of liquid in tank
    m_gas = (1-x) * m_total                                               #␣
↪get mass of gas in tank
    V_liq = m_liq / p_liq                                                 #␣
↪get volume of liquid
    V_gas = m_gas / p_gas                                                 #␣
↪get volume of gas
    V_liq_height = V_liq / base_area                                      #␣
↪get height of liquid in tank
    dip_tube_length = cyl_height - V_liq_height                           #␣
↪get diptube length
    ullage_percentage = (dip_tube_length / cyl_height) * 100

    return m_liq, m_gas, ullage_percentage

# def get_nitrous_mass_known_ullage(outside_temp, ullage_percentage):

#     p, p_liq, p_gas = nitrous_equations(outside_temp + 273.15)            #␣
↪get pressure and density values given known temperature
#     V_liq_height = cyl_height * (1-(ullage_percentage / 100))             #␣
↪get height of liquid given fixed 10% ullage
#     V_liq = V_liq_height * base_area                                      #␣
↪get volume of liquid in tank using liquid height and cylinder area of base
#     m_liq = V_liq * p_liq                                                 #␣
↪get mass of liquid given volume and density at given temperature
#     V_gas_height = cyl_height * (ullage_percentage / 100)                 #␣
↪get height of gas (ullage)
#     V_gas = V_gas_height * base_area                                      #␣
↪get volume of gas in tank using ullage and cylinder area of base
#     m_gas = V_gas * p_gas                                                 #␣
↪get mass of gas given volume and density at given temperature
#     m_total = m_liq + m_gas                                               #␣
↪get total amount of nitrous
#     ullage_percentage = (V_gas_height / cyl_height) * 100                 #␣
↪calculate ullage percentage

#     return m_liq, m_gas, ullage_percentage
```

[ ]:
```
def nozzle_mass_flow_rate(time, pressure, A, gamma, R, T_o):
    pt = pressure * 6894.76  # Convert psi to Pascals (Pa)
    mdot = ((A * pt) / np.sqrt(T_o)) * np.sqrt(gamma / R) * (((gamma + 1) /␣
↪2)**(-((gamma + 1) / (2 * (gamma - 1)))))
    return mdot
```

0.3 3. Determine Start and End Times

This section is used to determine the correct start and end times for filling, burn and liquid phase. I had scripts that would do this automatically, but it was too unreliable with the noisy data. While it might take a bit of time to tweak, hardcoding it like this is much more accurate. It can be very quick after you've done it for the first time.

```
[ ]: column_names = ['time_ms', 'run_pressure_V', 'fill_pressure_V', 'purge_pressure_V', 'tank_pressure_V', 'tank_mass_V', 'thrust_V', 'cc_pressure_V', 'tank_temp_V', 'run_temp_V', 'vent_temp_V', 'garbage', 'run_pressure_sw', 'fill_pressure_sw', 'purge_pressure_sw', 'tank_pressure_sw', 'tank_mass_sw', 'thrust_sw', 'cc_pressure_sw', 'run_temp_sw', 'tank_temp_sw', 'ven_temp_sw']
df = pd.read_csv(input_data, sep='\t', names=column_names, index_col=False, dtype=np.float64, skip_blank_lines=True)
time, p_fill, p_tank, p_run, p_cc, tank_temp, run_temp, tank_mass, thrust = (df['time_ms'] / 1000, df['fill_pressure_sw'], df['tank_pressure_sw'], df['run_pressure_sw'], df['cc_pressure_sw'], df['tank_temp_sw'], df['run_temp_sw'], df['tank_mass_sw'], df['thrust_sw'])
```

```
[ ]: # play around with these values until you find the correct times
start_time_fill = ???
start_time = ???
end_time = ???
end_time_fill = start_time
start_time_liq_pre = ???
end_time_liq_pre = ???
start_time_liq = ???
end_time_liq = ???
```

```
[ ]: plt.figure(figsize=(10, 6))
plt.plot(time, p_tank)
plt.title('Tank Pressure')
plt.grid(True)
plt.xlim(start_time_fill, start_time)
plt.show()

plt.figure(figsize=(10, 6))
plt.plot(time, p_run)
plt.plot(time, p_cc)
plt.axvline(x= start_time_liq_pre, color='darkorange', linestyle='--', linewidth=2)
plt.axvline(x= end_time_liq_pre, color='blue', linestyle='--', linewidth=2)
plt.title('Run Pressure')
plt.grid(True)
plt.xlim(start_time, end_time)
plt.show()
```

```python
plt.figure(figsize=(10, 6))
plt.plot(time, thrust)
#plt.axvline(x= liq_start, color='darkorange', linestyle='--', linewidth=2)
#plt.axvline(x= liq_end, color='blue', linestyle='--', linewidth=2)
plt.title('Thrust')
plt.grid(True)
plt.xlim(start_time, end_time)
plt.show()
```

0.4 4. Extract Data

This section actually extracts the data from the txt file and gets the correct data ranges and pressure deltas.

[ ]:

```python
# create dictionary of nitrous liquid density and temperature values (as solving for liquid density from temp is too mathematically intensive)
temp_list = np.linspace(183, 309, num = 10000)
vap_list, liquid_list, gas_list = nitrous_equation_lists(temp_list)
temp_liquid_density_dict = dict(zip(temp_list, liquid_list))
pressure_temp_dict = dict(zip(vap_list, temp_list))

# get time ranges
time_range = [t - start_time for t in time[(time >= start_time) & (time <= end_time)]]
time_range_fill = [t - start_time_fill for t in time if start_time_fill <= t <= end_time_fill]
time_range_full = [t - start_time_fill for t in time[(time >= start_time_fill) & (time <= end_time)]]

# get data ranges in time frame
p_fill_range = get_data_range(p_fill, time, start_time_fill, end_time_fill)
p_tank_fill_range = get_data_range(p_tank, time, start_time_fill, end_time_fill)
p_tank_range = get_data_range(p_tank, time, start_time, end_time)
p_run_range = get_data_range(p_run, time, start_time, end_time)
p_cc_range = get_data_range(p_cc, time, start_time, end_time)
tank_temp_range = get_data_range(tank_temp, time, start_time, end_time)
run_temp_range = get_data_range(run_temp, time, start_time, end_time)
thrust_range = get_data_range(thrust, time, start_time, end_time)
mass_range = get_data_range(tank_mass, time, start_time_fill, start_time)
mass_run_range = get_data_range(tank_mass, time, start_time, end_time)

# calculate pressure deltas
delta_p_tank_fill = [pt - pf for pt, pf in zip(p_tank_fill_range, p_fill_range)]
delta_p_tank_run = [pt - pr for pt, pr in zip(p_tank_range, p_run_range)]
delta_p_run_cc = [pr - pc for pr, pc in zip(p_run_range, p_cc_range)]
```

0.5 5. Mass Data Fitting and Mass Estimates

This section fits the mass data and gives mass estimates. Ideally, the first method is used as it is automatic. But for hotfires the data is often so noisy that you occasionally have to use Method 2, which hard codes a fit.

[ ]:
```
# Method 1: Low Pass Filter and Gaussian Smoothing (ideal)
#smoothed_mass_savgol = savgol_filter(mass_run_range, window_length=100,␣
↪polyorder=1) # Savitzky-Golay filter (Low-Pass)
smoothed_mass_gaussian = gaussian_filter1d(mass_run_range, sigma=60) # Gaussian␣
↪Smoothing

plt.figure(figsize=(10, 6))
plt.plot(time_range, mass_run_range, label='Original Mass (kg)', alpha=0.7)
#plt.plot(time_range, smoothed_mass_savgol, label='Savitzky-Golay (Low-Pass)',␣
↪color='red')
plt.plot(time_range, smoothed_mass_gaussian, label='Gaussian Smoothing',␣
↪color='red')
plt.title('Tank Mass (Run)')
plt.ylim(0,15)
plt.grid(True)
plt.legend()
plt.show()
```

[ ]:
```
# Method 2: Manual Polynomial Fit (desperate for a fit if too many vibrations.␣
↪You have to determine the x and y coordinates yourself)
# manually define points
coordinates = np.array([[0, 11], [1, 10], [2.5, 9], [5, 6.9], [6, 6.8], [7.5, 6.
↪7], [9, 6.3], [10, 6], [12.5, 5.7], [13.5, 5.1], [15, 4.9], [17.5, 3.8], [19.
↪5, 3.2]])
x_coordinates = coordinates[:, 0]
y_coordinates = coordinates[:, 1]
degree = 7  # You can adjust the degree based on the complexity
coefficients = np.polyfit(x_coordinates, y_coordinates, degree)
poly_fit = np.poly1d(coefficients)
fit_line = np.linspace(min(x_coordinates), max(x_coordinates), 100)

plt.figure(figsize=(10, 6))
plt.plot(time_range, mass_run_range, label='Original Mass (kg)', alpha=0.7)
plt.plot(fit_line, poly_fit(fit_line), label='Polynomial Fit', color='red')
plt.title('Tank Mass (Run)')
plt.ylim(0,15)
plt.grid(True)
plt.legend()
plt.show()
```

[ ]:
```
# If using Method 2 for mass data fit
def integrand(x):
    return derivative_poly_fit(x)

derivative_poly_fit = poly_fit.deriv()
integrated_mass, _ = quad(integrand, min(x_coordinates), max(x_coordinates))
```

[ ]:
```
# Total Mass (end of fill)
m_total_start = 8

plt.figure(figsize=(10, 6))
plt.plot(time_range_fill, mass_range, label='Mass (kg)')
plt.axhline(y= m_total_start, color='red', linestyle='--', linewidth=2)
plt.title('Tank Mass Filling and Heating (Load Cell)')
#plt.xlim(1750,2000)
#plt.ylim(7, 9)
plt.grid(True)
plt.show()
```

[ ]:
```
# Mass from diptube length (ullage) estimate
# m_liq, m_gas, ullage_percentage =␣
 ↪get_carbon_dioxide_mass_known_ullage(outdoor_temp, 25)
# diptube_mass = m_liq + m_gas
```

[ ]:
```
# mass estimates
print('Load Cell Mass (end of fill) =', m_total_start, 'kg')
print('Integrated Mass Flow Rate Fit =', abs(integrated_mass), 'kg')
```

0.6 6. Calculate Important Parameters

This is where the important parameters are calculated. Copy over what is printed here to the LaTex reports.

[ ]:
```
# fill time
fill_time = start_time - start_time_fill
```

[ ]:
```
# peak tank pressure
peak_tank_pressure = max(p_tank_range)

# peak_tank_temperature
peak_tank_temp = max(tank_temp_range)

# peak run pressure
peak_run_pressure = max(p_run_range)

# peak combustion chamber pressure
peak_cc_pressure = max(p_cc_range)
```

[ ]:
```python
# ullage factor, liquid and gaseous nitrous mass
m_liq, m_gas, ullage_percentage = get_nitrous_mass_unknown_ullage(peak_tank_pressure, m_total_start)
#m_liq, m_gas, ullage_percentage = get_nitrous_mass_known_ullage(outdoor_temp)
```

[ ]:
```python
# mass flow rate
average_mass_flow_rate, _ = quad(lambda x: derivative_poly_fit(x), start_time_liq, end_time_liq)
average_mass_flow_rate = average_mass_flow_rate / (end_time_liq - start_time_liq)
peak_mass_flow_rate = minimize_scalar(lambda x: derivative_poly_fit(x), bounds=(start_time_liq, end_time_liq), method='bounded').fun

# discharge coefficient
average_c_d = calculate_discharge_coefficient(-average_mass_flow_rate, hole_diameter, num_holes, p_run_range, run_temp_range, delta_p_run_cc, temp_liquid_density_dict)
peak_c_d = calculate_discharge_coefficient(-peak_mass_flow_rate, hole_diameter, num_holes, p_run_range, run_temp_range, delta_p_run_cc, temp_liquid_density_dict)
```

[ ]:
```python
# burn time
burn_time = end_time - start_time

# peak thrust
peak_thrust = max(thrust_range)

# total impulse
total_impulse = calculate_total_impulse(time_range, thrust_range)
```

[ ]:
```python
# nozzle mass flow rate
nozzle_mass_flow_rate = [nozzle_mass_flow_rate(t, p, A, gamma, R, T_o) for t, p in zip(time_range, p_cc_range)]

## specific impulse

# Method 1: Use Nozzle Mass Flow Rate
isp = []
for thrust, mdot in zip(thrust_range, nozzle_mass_flow_rate):
    isp_value = thrust / (mdot * 9.81)
    isp.append(isp_value)

average_isp_value = np.mean([value for t, value in zip(time_range, isp) if start_time_liq <= t <= end_time_liq])

# Method 2: Total Impulse
```

```
total_impulse_isp = total_impulse / (((mean_mass + m_fuel) * 9.81))
```

[ ]:
```
# print values
print('Ullage Factor =', ullage_percentage, '%')
print('Fill Time =', fill_time, 's')
print('Peak Tank Pressure =', peak_tank_pressure, 'psi')
print('Peak Tank Temp =', peak_tank_temp, 'C')
print('Peak Run Pressure =', peak_run_pressure, 'psi')
print('Peak CC Pressure =', peak_cc_pressure, 'psi')
print('Peak Mass Flow Rate =', - peak_mass_flow_rate, 'kg/s')
print('Average Mass Flow Rate =', - average_mass_flow_rate, 'kg/s')
print('Mass of Liquid =', m_liq, 'kg')
print('Mass of Gas =', m_gas, 'kg')
print('Burn Time =', burn_time, 's')
print('Peak Thrust =', peak_thrust, 'N')
print('Total Impulse =', total_impulse, 'Ns')
print('Specific Impulse (nozzle mass flow rate) =', average_isp_value, 's')
print('Specific Impulse (total impulse) =', total_impulse_isp, 's')
```

0.7 7. Mass Error Propagation

This section propogates the error in the mass data to the mass flow rate and discharge coefficient

[ ]:
```
# Mass Error (standard deviation and standard error)
#mass_data = np.array([m_total_start, abs(integrated_mass), diptube_mass])
mass_data = np.array([m_total_start, abs(integrated_mass)])
mean_mass = np.mean(mass_data)
std_dev = np.std(mass_data, ddof=1)
std_error = std_dev / np.sqrt(len(mass_data))

print("Mean Mass:", mean_mass, "kg")
print("Standard Deviation:", std_dev, "kg")
print("Standard Error:", std_error, "kg")
```

[ ]:
```
# Mass Flow Rate Error
mass_flow_rate_std_error = np.abs(average_mass_flow_rate) * std_error
```

[ ]:
```
# Discharge Coefficient Error
average_c_d_std_error =␣
 ↪calculate_discharge_coefficient_uncertainty(average_c_d, hole_diameter,␣
 ↪num_holes, p_run_range, delta_p_run_cc, temp_liquid_density_dict,␣
 ↪mass_flow_rate_std_error)
peak_c_d_std_error = calculate_discharge_coefficient_uncertainty(peak_c_d,␣
 ↪hole_diameter, num_holes, p_run_range, delta_p_run_cc,␣
 ↪temp_liquid_density_dict, mass_flow_rate_std_error)
```

```
[ ]: # Total Impulse Error
upper_mean_mass = mean_mass + std_error
lower_mean_mass = mean_mass - std_error

upper_m_fuel = m_fuel + m_fuel_uncertainty
lower_m_fuel = m_fuel - m_fuel_uncertainty

# Calculate upper and lower bounds for total_impulse_isp
upper_total_impulse_isp = (total_impulse) / (((lower_mean_mass + upper_m_fuel)␣
 ↪* 9.8))
lower_total_impulse_isp = (total_impulse) / (((upper_mean_mass + lower_m_fuel)␣
 ↪* 9.8))
```

0.8 8. Plots

Plot the results

```
[ ]: # Fill Pressure
plt.figure(figsize=(10, 6))
plt.plot(time_range_fill, p_fill_range, label='Fill Pressure')
plt.plot(time_range_fill, p_tank_fill_range, label='Tank Pressure')
plt.xlabel('Time (s)', fontsize = 15)
plt.ylabel('Pressure (psi)', fontsize = 15)
plt.title('Fill Pressure', fontsize = 15)
plt.grid(True)
plt.legend(fontsize = 12)
plt.gca().set_facecolor('white')
plt.savefig('fill_pressure.png', facecolor='white')
plt.show()

# Fill Pressure Zoomed
plt.figure(figsize=(10, 6))
plt.plot(time_range_fill, p_fill_range, label='Fill Pressure')
plt.plot(time_range_fill, p_tank_fill_range, label='Tank Pressure')
plt.xlabel('Time (s)', fontsize = 15)
plt.ylabel('Pressure (psi)', fontsize = 15)
plt.title('Fill Pressure', fontsize = 15)
plt.grid(True)
plt.legend(fontsize = 15)
plt.xlim(0,30)
#plt.ylim(0,500)
plt.gca().set_facecolor('white')
plt.savefig('fill_pressure_zoomed.png', facecolor='white')
plt.show()
```

```
[ ]: # Temperature
plt.figure(figsize=(10, 6))
```

```python
plt.plot(time_range, run_temp_range, label='Run Temperature (C)')
plt.plot(time_range, tank_temp_range, label='Tank Temperature (C)')
plt.xlabel('Time (s)', fontsize = 15)
plt.ylabel('Temperature (C)', fontsize = 15)
plt.title('Tank Emptying Temperature', fontsize = 15)
plt.grid(True)
plt.legend(fontsize = 15)
plt.gca().set_facecolor('white')
plt.savefig('tank_emptying_temp.png', facecolor='white')
plt.show()
```

[ ]:
```python
# Tank/Run Pressure
plt.figure(figsize=(10, 6))
plt.plot(time_range, p_tank_range, label='Tank Pressure')
plt.plot(time_range, p_run_range, label='Run Line Pressure')
plt.plot(time_range, delta_p_tank_run, linestyle='--', label='Delta_P')
plt.xlabel('Time (s)', fontsize = 15)
plt.ylabel('Pressure (psi)', fontsize = 15)
plt.title('Tank Emptying Pressure Drop', fontsize = 15)
plt.grid(True)
plt.legend(fontsize = 12)
plt.gca().set_facecolor('white')
plt.yticks(np.arange(0, max(p_tank_range) + 100, 100))
plt.savefig('tank_emptying_pressure.png', facecolor='white')
plt.show()

# Injector Pressure Drop
plt.figure(figsize=(10, 6))
plt.plot(time_range, p_run_range, label='Run Line Pressure')
plt.plot(time_range, p_cc_range, label='Combustion Chamber Pressure')
plt.plot(time_range, delta_p_run_cc, linestyle='--', label='Delta_P')
plt.xlabel('Time (s)', fontsize = 15)
plt.ylabel('Pressure (psi)', fontsize = 15)
plt.title('Injector Pressure Drop', fontsize = 15)
plt.grid(True)
plt.legend(fontsize = 12)
plt.gca().set_facecolor('white')
plt.savefig('injector_pressure_drop.png', facecolor='white')
plt.show()
```

[ ]:
```python
# Thrust
plt.figure(figsize=(10, 6))
plt.plot(time_range, thrust_range, label='Thrust (N)')
plt.xlabel('Time (s)', fontsize = 15)
plt.ylabel('Thrust (N)', fontsize = 15)
plt.title('Thrust', fontsize = 15)
plt.grid(True)
```

```
plt.gca().set_facecolor('white')
plt.savefig('thrust.png', facecolor='white')
plt.show()
```

[ ]:
```
# Mass
plt.figure(figsize=(10, 6))
plt.plot(time_range_fill, mass_range, label='Mass (kg)')
plt.xlabel('Time (s)', fontsize = 15)
plt.ylabel('Mass (kg)', fontsize = 15)
plt.title('Tank Mass Filling and Heating (Load Cell)', fontsize = 15)
plt.grid(True)
plt.gca().set_facecolor('white')
plt.savefig('mass_fill.png', facecolor='white')
plt.show()


# 2.9 Mass (Run)
plt.figure(figsize=(10, 6))
plt.plot(time_range, mass_run_range, label='Original Mass (kg)', alpha=0.7)
plt.plot(fit_line, poly_fit(fit_line), label='Polynomial Fit', color='red')
upper_bound = poly_fit(fit_line) + std_error
lower_bound = poly_fit(fit_line) - std_error
plt.fill_between(fit_line, upper_bound, lower_bound, color='red', alpha=0.3,␣
 ↪label='Uncertainty Bounds')
plt.title('Tank Mass (Run)', fontsize = 15)
plt.xlabel('Time (s)', fontsize = 15)
plt.ylabel('Mass (kg)', fontsize = 15)
plt.grid(True)
plt.ylim(0,15)
plt.gca().set_facecolor('white')
plt.legend(fontsize = 12)
plt.savefig('mass_tank_emptying.png', facecolor='white')
plt.show()

# Mass Flow Rate
derivative_values = -derivative_poly_fit(fit_line)

# Calculate the upper and lower bounds using the standard error
upper_bound = derivative_values + mass_flow_rate_std_error
lower_bound = derivative_values - mass_flow_rate_std_error

# Create the plot
plt.figure(figsize=(10, 6))
plt.plot(fit_line, derivative_values, label='Time Derivative of Liquid Phase␣
 ↪Mass (ṁ)', color='teal')
plt.axhline(y= - average_mass_flow_rate, color='blue', linestyle='--',␣
 ↪label="Average ṁ (Liquid Phase) = 0.77 kg/s", linewidth=2)
```

```python
plt.axhline(y= - peak_mass_flow_rate, color='darkorange', linestyle='--', label="Peak ṁ = 0.97 kg/s", linewidth=2)
plt.fill_between(fit_line, upper_bound, lower_bound, color='teal', alpha=0.3, label='Uncertainty Bounds')
plt.title('Mass Flow Rate', fontsize=15)
plt.xlabel('Time (s)', fontsize=15)
plt.ylabel('Mass Flow Rate (kg/s)', fontsize=15)
plt.grid(True)
plt.legend(fontsize=12)
plt.xlim(start_time_liq, end_time_liq)
plt.ylim(0, 2)
plt.gca().set_facecolor('white')
plt.savefig('injector_mass_flow_rate.png', facecolor='white')
plt.show()

average_c_d = np.array(average_c_d)
average_c_d_std_error = np.array(average_c_d_std_error)
peak_c_d = np.array(peak_c_d)
peak_c_d_std_error = np.array(peak_c_d_std_error)

# Discharge Coefficient Estimation
plt.figure(figsize=(10, 6))
plt.plot(time_range, average_c_d, label='Average ṁ = 0.77 kg/s (CO2)', color = 'blue')
plt.plot(time_range, peak_c_d, label='Peak ṁ = 0.97 kg/s (CO2)', color = 'darkorange')
upper_bound_avg = average_c_d + average_c_d_std_error
lower_bound_avg = average_c_d - average_c_d_std_error
upper_bound_peak = peak_c_d + peak_c_d_std_error
lower_bound_peak = peak_c_d - peak_c_d_std_error
plt.fill_between(time_range, upper_bound_avg, lower_bound_avg, color='blue', alpha=0.3, label='Uncertainty Bounds')
plt.fill_between(time_range, upper_bound_peak, lower_bound_peak, color='darkorange', alpha=0.3, label='Uncertainty Bounds')
plt.title('Discharge Coefficient Estimation', fontsize = 15)
plt.xlabel('Time (s)', fontsize = 15)
plt.ylabel('Discharge Coefficient', fontsize = 15)
plt.grid(True)
plt.legend(fontsize = 12)
plt.xlim(start_time_liq, end_time_liq)
plt.ylim(0,0.6)
plt.gca().set_facecolor('white')
plt.savefig('discharge_coefficient.png', facecolor='white')
plt.show()
```

[ ]:
```python
# Nozzle Mass Flow Rate
plt.figure(figsize=(10, 6))
plt.plot(time_range, nozzle_mass_flow_rate)
plt.xlabel("Time (s)", fontsize = 15)
plt.ylabel("Mass Flow Rate (kg/s)", fontsize = 15)
plt.title("Estimated Nozzle Mass Flow Rate", fontsize = 15)
plt.grid(True)
plt.gca().set_facecolor('white')
plt.savefig('nozzle_mass_flow_rate.png', facecolor='white')
plt.show()

#upper_mean_mass = mean_mass + std_error
#lower_mean_mass = mean_mass - std_error

#upper_m_fuel = m_fuel + m_fuel_uncertainty
#lower_m_fuel = m_fuel - m_fuel_uncertainty

# Calculate upper and lower bounds for total_impulse_isp
#upper_total_impulse_isp = (total_impulse) / (((lower_mean_mass + upper_m_fuel) * 9.8))
#lower_total_impulse_isp = (total_impulse) / (((upper_mean_mass + lower_m_fuel) * 9.8))
#print(upper_total_impulse_isp)
#print(lower_total_impulse_isp)

# Specific Impulse (Estimated)
plt.figure(figsize=(10, 6))
plt.plot(time_range, isp, label="Nozzle Mass Fow Rate ISP estimate ")
plt.axhline(y=average_isp_value, color='r', linestyle='--', label="Nozzle Mass Fow Rate ISP average estimate = 228 s")
plt.axhline(y=total_impulse_isp, color='g', linestyle='--', label="Total Impulse ISP estimate = 232 s")
plt.fill_between(time_range, upper_total_impulse_isp, lower_total_impulse_isp, color='g', alpha=0.2)
plt.xlabel("Time (s)", fontsize = 15)
plt.ylabel("Specific Impulse (s)", fontsize = 15)
plt.title("Estimated Specific Impulse (liquid phase)", fontsize = 15)
plt.grid(True)
plt.legend(fontsize = 12)
plt.xlim(start_time_liq, end_time_liq)
plt.ylim(150,300)
plt.gca().set_facecolor('white')
plt.savefig('isp.png', facecolor='white')
plt.show()
```