VISUALIZATION PERFORMANCE PARAMETERS HIGH RISE STEEL BUILDINGS

COMPARISON BETWEEN BRACED FRAMED TUBE AND OUTRIGGER SYSTEM

Visualization of the embodied carbon and costs based on the given data sets of the both systems

Emma Potijk (4715802) \ MSc Thesis Structural Engineering \ University: TU Delft \ Company: Buro Happold \ Period: 08/2023 - 04/2024

Import standard functions

In [ ]:
```
import pandas as pd
import numpy as np
from sklearn.model_selection import KFold
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_squared_error
from sklearn.neural_network import MLPRegressor
from sklearn.model_selection import train_test_split

import matplotlib.pyplot as plt
import seaborn as sns
sns.set(style="whitegrid")

import ipywidgets as widgets
from IPython.display import display
from ipywidgets import interact
import copy
import warnings
from mpl_toolkits.mplot3d import Axes3D

import matplotlib.pyplot as plt
import matplotlib.colors as mcolors
```

In [ ]:
```
#pip install geneticalgorithm
```

In [ ]:
```
warnings.filterwarnings(action='ignore', category=UserWarning)
```

In [ ]:
```
color_orange = '#FF914D'
color_lightblue = '#38B6FF'
color_green = '#589D62'
color_purple = '#CB6CE6'
color_darkblue = '#417DB9'
color_yellow = '#FFF3C2'
color_grey = '#A6A6A6'
color_lightgreen = '#8BE198'
```

--- CONTENT ---

1) Data preprocessing

a) Import data
b) Converting input features

2) Data visualization

a) Scatter plots of mass
b) Scatter plots of costs
c) Scatter plots of embodied carbon
d) Scatter plots of volume
e) Pairplot of costs to embodied carbon
f) Visualization with one constant variable

3) Comparison of computational time

4) Determination of impact stability framework

a) Contribution of stability framework vs gravity framework
    i) Braced framed tube
    ii) Outrigger

--- CODE ---

1) DATA PREPROCESSING

S1 = Braced Framed Tube System (grid size = 3.6 m)

S2 = Outrigger System (grid size = 1.8 m)

i) IMPORT DATA

S1: Braced Framed Tube

In [ ]:
```
file_path = r'C:\Users\epotijk\OneDrive - Buro Happold\MSc Thesis\Data Sets GH\Colibri_Braced_Framed_Tube\data.csv'
dataset_S1 = pd.read_csv(file_path, delimiter = ',')
```

```
dataset_S1 = dataset_S1.drop(columns = ['img'])
dataset_S1 = dataset_S1.loc[(dataset_S1 >= 0).all(axis=1)]
```

In [ ]:
```
ranges = [(2, 8), (8, 14), (14, 20), (20, 26)]

def check_descending(row):
    for start, end in ranges:
        for i in range(end-1, start-1, -1):  # Iterate in reverse order
            if i > start and row[i] > row[i-1]:  # Check if values are not descending
                row[i-1] = row[i]  # Replace smaller value with larger one
    return row

dataset_S1 = dataset_S1.copy()
dataset_S1 = dataset_S1.apply(check_descending, axis=1)
dataset_S1
```

S2: Outrigger

In [ ]:
```
file_path = r'C:\Users\epotijk\OneDrive - Buro Happold\MSc Thesis\Data Sets GH\Colibri_Outrigger_DivisionColumns_v3\data.csv'
dataset_S2 = pd.read_csv(file_path, delimiter = ',')

#dataset_S2 = dataset_S2.drop(columns = ['img'])
#dataset_S2 = dataset_S2.loc[(dataset_S2 >= 0).all(axis=1)]
```

In [ ]:
```
#filtered_rows = dataset_S2[dataset_S2['in:Number of grids'] == 8]
#filtered_rows
```

In [ ]:
```
pd.set_option('display.max_rows', 20)  # Replace None with the exact number of rows if known.
pd.set_option('display.max_columns', 20)  # Replace None with the exact number of columns if known.
#filtered_rows
```

Filtering the data set so that the maximum allowed displacement is not exceeded

In [ ]:
```
mask = dataset_S2['out:Displacement Framework [cm]'] > dataset_S2['out:Max allowed displacement [cm]']
dataset_S2 = dataset_S2[~mask]
```

In [ ]:
```
dataset_S2.replace(-999.000000, 0.000000)
indices_to_drop6 = dataset_S2[dataset_S2['in:Number of grids'] == 6].index
indices_to_drop8 = dataset_S2[dataset_S2['in:Number of grids'] == 8].index

dataset_S2.drop(indices_to_drop6, inplace=True)
dataset_S2.drop(indices_to_drop8, inplace=True)
```

ii) CONVERTING INPUT FEATURES

Geometry Variables

In [ ]:
```
# Ensure dataset_S2 is not a view or a copy of another DataFrame
dataset_S2 = dataset_S2.copy()
dataset_S1 = dataset_S1.copy()

grid_size_S1 = 3.6
grid_size_S2 = 1.8
floor_to_floor = 3.0

dataset_S1['in:Number of grids'] = dataset_S1['in:Number of grids'] * grid_size_S1
dataset_S1['in:Number of floors'] = dataset_S1['in:Number of floors'] * floor_to_floor

dataset_S2['in:Number of grids'] = dataset_S2.iloc[:,0] * grid_size_S2
dataset_S2['in:Number of floors'] = dataset_S2.iloc[:,1] * floor_to_floor

dataset_S1 = dataset_S1.rename(columns={'in:Number of grids' : 'in:Length [m]'})
dataset_S1 = dataset_S1.rename(columns={'in:Number of floors' : 'in:Height [m]'})

dataset_S2 = dataset_S2.rename(columns={'in:Number of grids' : 'in:Length [m]'})
dataset_S2 = dataset_S2.rename(columns={'in:Number of floors' : 'in:Height [m]'})
```

In [ ]:
```
mass_S1 = dataset_S1.copy()
mass_S2 = dataset_S2.copy()
```

Determination Height Zones

In [ ]:
```
X_S1 = mass_S1[['in:Length [m]', 'in:Height [m]']]
X_S2 = mass_S2[['in:Length [m]', 'in:Height [m]']]
```

In [ ]:
```
def determine_wind_zones(X):
    if isinstance(X, pd.DataFrame):
        b = X.iloc[:, 0].values
        h = X.iloc[:, 1].values
    else:
        b = X[:, 0]
        h = X[:, 1]

    wind_zones = np.zeros_like(b, dtype=int)

    for i in range(len(b)):
        if h[i] <= b[i]:
            wind_zones[i] = 1
        elif b[i] <= h[i] < 2 * b[i]:
            wind_zones[i] = 2
        elif h[i] > 2 * b[i]:
            wind_zones[i] = 3

    return wind_zones

def determine_h_wind_zones(wind_zones, X):
    if isinstance(X, pd.DataFrame):
        b = X.iloc[:, 0].values
        h = X.iloc[:, 1].values
```

```
    else:
        b = X[:, 0]
        h = X[:, 1]

    z_e1 = np.zeros_like(wind_zones, dtype=float)
    z_e2 = np.zeros_like(wind_zones, dtype=float)
    z_e3 = np.zeros_like(wind_zones, dtype=float)

    floor_to_floor = 3

    for i, zone in enumerate(wind_zones):
        if zone == 1:
            z_e1[i] = h[i]
            z_e2[i] = 0
            z_e3[i] = 0
        elif zone == 2:
            z_e1[i] = b[i]
            z_e2[i] = h[i]
            z_e3[i] = 0
        elif zone == 3:
            z_e1[i] = (b[i] // floor_to_floor) * floor_to_floor
            z_e2[i] = h[i] - ((b[i] // floor_to_floor) * floor_to_floor)
            z_e3[i] = h[i]

    z_e1 = np.round(z_e1, 1)
    z_e2 = np.round(z_e2, 1)
    z_e3 = np.round(z_e3, 1)


    height_wind_zones = np.column_stack((z_e1, z_e2, z_e3))
    return height_wind_zones

def determine_height_zones(wind_zones, X, height_wind_zones):
    num_zones = np.zeros_like(wind_zones, dtype=float)

    def distribute_floors(total_floors, num_zones):
        if num_zones == 0:
            return [], 0

        base_floors_per_zone = total_floors // num_zones
        remaining_floors = total_floors % num_zones

        floors_per_zone = [base_floors_per_zone] * num_zones

        for i in range(int(remaining_floors)):
            floors_per_zone[i] += 1

        return floors_per_zone, len(floors_per_zone)

    for i in range(len(wind_zones)):
        if wind_zones[i] == 1:
            num_zones[i] = 1

        elif wind_zones[i] == 2:
            num_zones[i] = 2

        elif wind_zones[i] == 3:
            floors_2 = height_wind_zones[i, 1] - height_wind_zones[i, 0]
            ratio = (height_wind_zones[i, 1] - height_wind_zones[i, 0]) / height_wind_zones[i, 0]

            if ratio == 1.5:
                ratio = 2
            if ratio == 3.5:
                ratio = 4
            if ratio == 2.5:
                ratio = 3
            else:
                ratio = round(ratio)
                if ratio > 4:
                    ratio = 4
                else:
                    floors_distribution_windzone2, outcome_length = distribute_floors(floors_2, int(ratio))

            if ratio <= 1:
                num_zones[i] = 3

            elif ratio > 1:
                additional_zones = ratio
                additional_zones = min(additional_zones, 4)
                num_zones[i] = additional_zones + 2

    return num_zones
```

In [ ]:
```
wind_zones_S2 = determine_wind_zones(X_S2)
height_wind_zones_S2 = determine_h_wind_zones(wind_zones_S2, X_S2)
num_zones_S2 = mass_S2.iloc[:, 47]

wind_zones_S1 = determine_wind_zones(X_S1)
height_wind_zones_S1 = determine_h_wind_zones(wind_zones_S1, X_S1)
num_zones_S1 = mass_S1.iloc[:, 33]
```

In [ ]:
```
def determine_height_zones_length(height_wind_zones, num_zones):
    height_zones_length = []
    for i in range(len(num_zones)):
        if num_zones.iloc[i] == 2:
            l1 = height_wind_zones[i][0]
            l2 = height_wind_zones[i][1] - height_wind_zones[i][0]
            height_zones_length.append([l1, l2, 0, 0, 0, 0])
        if num_zones.iloc[i] == 3:
            l1 = height_wind_zones[i][0]
            l3 = height_wind_zones[i][2] - height_wind_zones[i][1]
            l2 = height_wind_zones[i][2] - l3 - l1
            height_zones_length.append([l1, l2, l3, 0, 0, 0])
        if num_zones.iloc[i] == 4:
            l1 = height_wind_zones[i][0]
            l4 = height_wind_zones[i][2] - height_wind_zones[i][1]
```

```
            l23 = height_wind_zones[i][2] - l4 - l1
            l2 = l23 / 2
            l3 = l23 / 2
            height_zones_length.append([l1, l2, l3, l4, 0, 0])
        if num_zones.iloc[i] == 5:
            l1 = height_wind_zones[i][0]
            l5 = height_wind_zones[i][2] - height_wind_zones[i][1]
            l234 = height_wind_zones[i][2] - l5 - l1
            l2 = l234 / 3
            l3 = l234 / 3
            l4 = l234 / 3
            height_zones_length.append([l1, l2, l3, l4, l5, 0])
        if num_zones.iloc[i] == 6:
            l1 = height_wind_zones[i][0]
            l6 = height_wind_zones[i][2] - height_wind_zones[i][1]
            l12345 = height_wind_zones[i][2] - l6 - l1
            l2 = l12345 / 4
            l3 = l12345 / 4
            l4 = l12345 / 4
            l5 = l12345 / 4
            height_zones_length.append([l1, l2, l3, l4, l5, l6])
    return height_zones_length
```

In [ ]:
```
height_zones_length_S2 = determine_height_zones_length(height_wind_zones_S2, num_zones_S2)
height_zones_length_S1 = determine_height_zones_length(height_wind_zones_S1, num_zones_S1)
```

In [ ]:
```
height_zones_length_S2[100]
```

In [ ]:
```
#height_zones_length_S2[106]
index_to_delete = 100
if 0 <= index_to_delete < len(height_zones_length_S2):
    del height_zones_length_S2[index_to_delete]
```

In [ ]:
```
index_label = mass_S2.iloc[100].name
mass_S2.drop(index_label, inplace=True)
```

In [ ]:
```
y = len(height_zones_length_S2)
x = len(mass_S2)
print(y)
print(x)
```

Determination Average Mass per GFA over total Height of Structure

Braced Framed Tube

In [ ]:
```
#OUTER BEAMS
#Outer beams: average value over total structure height
mass_final_OB_S1 = []

for i in range(len(mass_S1)):
    sum_of_heights = sum(height_zones_length_S1[i][0:6])
    mass_final_OB_value = (
        mass_S1.iloc[i, 8] * height_zones_length_S1[i][0]
        + mass_S1.iloc[i, 9] * height_zones_length_S1[i][1]
        + mass_S1.iloc[i, 10] * height_zones_length_S1[i][2]
        + mass_S1.iloc[i, 11] * height_zones_length_S1[i][3]
        + mass_S1.iloc[i, 12] * height_zones_length_S1[i][4]
        + mass_S1.iloc[i, 13] * height_zones_length_S1[i][5]
    ) / sum_of_heights
    mass_final_OB_S1.append(mass_final_OB_value)


#OUTER COLUMNS
#Outer columns: average value over total structure height
mass_final_OC_S1 = []  # Initialize an empty list

for i in range(len(mass_S1)):
    sum_of_heights = sum(height_zones_length_S1[i][0:6])
    mass_final_OC_value = (
        mass_S1.iloc[i, 14] * height_zones_length_S1[i][0]
        + mass_S1.iloc[i, 15] * height_zones_length_S1[i][1]
        + mass_S1.iloc[i, 16] * height_zones_length_S1[i][2]
        + mass_S1.iloc[i, 17] * height_zones_length_S1[i][3]
        + mass_S1.iloc[i, 18] * height_zones_length_S1[i][4]
        + mass_S1.iloc[i, 19] * height_zones_length_S1[i][5]
    ) / sum_of_heights
    mass_final_OC_S1.append(mass_final_OC_value)


#FLOOR BEAMS
#Floor Beams: average value over total structure height
mass_final_FB_S1 = []  # Initialize an empty list

for i in range(len(mass_S1)):
    sum_of_heights = sum(height_zones_length_S1[i][0:6])
    mass_final_FB_value = (
        mass_S1.iloc[i, 20] * height_zones_length_S1[i][0]
        + mass_S1.iloc[i, 21] * height_zones_length_S1[i][1]
        + mass_S1.iloc[i, 22] * height_zones_length_S1[i][2]
        + mass_S1.iloc[i, 23] * height_zones_length_S1[i][3]
        + mass_S1.iloc[i, 24] * height_zones_length_S1[i][4]
        + mass_S1.iloc[i, 25] * height_zones_length_S1[i][5]
    ) / sum_of_heights
    mass_final_FB_S1.append(mass_final_FB_value)


#INNER COLUMNS
#Inner Columns: average value over total structure height
mass_final_IC_S1 = []  # Initialize an empty list

for i in range(len(mass_S1)):
    sum_of_heights = sum(height_zones_length_S1[i][0:6])
    mass_final_IC_value = (
        mass_S1.iloc[i, 26] * height_zones_length_S1[i][0]
```

```
        + mass_S1.iloc[i, 27] * height_zones_length_S1[i][1]
        + mass_S1.iloc[i, 28] * height_zones_length_S1[i][2]
        + mass_S1.iloc[i, 29] * height_zones_length_S1[i][3]
        + mass_S1.iloc[i, 30] * height_zones_length_S1[i][4]
        + mass_S1.iloc[i, 31] * height_zones_length_S1[i][5]
    ) / sum_of_heights
    mass_final_IC_S1.append(mass_final_IC_value)
```

In [ ]:
```
mass_S1['Mass Final OB'] = mass_final_OB_S1
mass_S1['Mass Final OC'] = mass_final_OC_S1
mass_S1['Mass Final FB'] = mass_final_FB_S1
mass_S1['Mass Final IC'] = mass_final_IC_S1
```

In [ ]:
```
#Material Split
mass_S1['Mass Total Steel'] = (mass_S1['Mass Final OB'] + mass_S1['Mass Final OC']
                               + mass_S1['Mass Final FB'] + mass_S1['Mass Final IC']
                               + mass_S1.iloc[:, 32])

mass_S1['Mass Total Floors'] = mass_S1.iloc[:, 2:8].sum(axis=1) / mass_S1.iloc[:, 33]
mass_S1['Mass Total'] = mass_S1['Mass Total Steel']
```

Outrigger System

In [ ]:
```
#OUTER BEAMS
#Outer beams: average value over total structure height
mass_final_OB_S2 = []

for i in range(len(mass_S2)):
    sum_of_heights = sum(height_zones_length_S2[i][0:6])
    if sum_of_heights == 0 or np.isnan(sum_of_heights):
        mass_final_OB_value = np.nan
    else:
        mass_final_OB_value = (
            mass_S2.iloc[i, 8] * height_zones_length_S2[i][0]
            + mass_S2.iloc[i, 9] * height_zones_length_S2[i][1]
            + mass_S2.iloc[i, 10] * height_zones_length_S2[i][2]
            + mass_S2.iloc[i, 11] * height_zones_length_S2[i][3]
            + mass_S2.iloc[i, 12] * height_zones_length_S2[i][4]
            + mass_S2.iloc[i, 13] * height_zones_length_S2[i][5]
        ) / sum_of_heights
        mass_final_OB_S2.append(mass_final_OB_value)


#OUTER COLUMNS CONNECTED TO OUTRIGGER
#Outer columns connected to Outrigger: average value over total structure height
mass_final_OC_Out_S2 = []  # Initialize an empty list

for i in range(len(mass_S2)):
    sum_of_heights = sum(height_zones_length_S2[i][0:6])
    if sum_of_heights == 0 or np.isnan(sum_of_heights):
        mass_final_OC_value = np.nan
    else:
        mass_final_OC_Out_value = (
            mass_S2.iloc[i, 14] * height_zones_length_S2[i][0]
            + mass_S2.iloc[i, 15] * height_zones_length_S2[i][1]
            + mass_S2.iloc[i, 16] * height_zones_length_S2[i][2]
            + mass_S2.iloc[i, 17] * height_zones_length_S2[i][3]
            + mass_S2.iloc[i, 18] * height_zones_length_S2[i][4]
            + mass_S2.iloc[i, 19] * height_zones_length_S2[i][5]
        ) / sum_of_heights
        mass_final_OC_Out_S2.append(mass_final_OC_Out_value)


#OUTER COLUMNS NOT CONNECTED TO OUTRIGGER
#Outer columns NOT connected to Outrigger: average value over total structure height
mass_final_OC_noOut_S2 = []  # Initialize an empty list

for i in range(len(mass_S2)):
    sum_of_heights = sum(height_zones_length_S2[i][0:6])
    if sum_of_heights == 0 or np.isnan(sum_of_heights):
        mass_final_OC_noOut_value = np.nan
    else:
        mass_final_OC_noOut_value = (
            mass_S2.iloc[i, 20] * height_zones_length_S2[i][0]
            + mass_S2.iloc[i, 21] * height_zones_length_S2[i][1]
            + mass_S2.iloc[i, 22] * height_zones_length_S2[i][2]
            + mass_S2.iloc[i, 23] * height_zones_length_S2[i][3]
            + mass_S2.iloc[i, 24] * height_zones_length_S2[i][4]
            + mass_S2.iloc[i, 25] * height_zones_length_S2[i][5]
        ) / sum_of_heights
        mass_final_OC_noOut_S2.append(mass_final_OC_noOut_value)


#FLOOR BEAMS
#Floor Beams: average value over total structure height
mass_final_FB_S2 = []  # Initialize an empty list

for i in range(len(mass_S2)):
    sum_of_heights = sum(height_zones_length_S2[i][0:6])
    if sum_of_heights == 0 or np.isnan(sum_of_heights):
        mass_final_FB_value = np.nan
    else:
        mass_final_FB_value = (
            mass_S2.iloc[i, 26] * height_zones_length_S2[i][0]
            + mass_S2.iloc[i, 27] * height_zones_length_S2[i][1]
            + mass_S2.iloc[i, 28] * height_zones_length_S2[i][2]
            + mass_S2.iloc[i, 29] * height_zones_length_S2[i][3]
            + mass_S2.iloc[i, 30] * height_zones_length_S2[i][4]
            + mass_S2.iloc[i, 31] * height_zones_length_S2[i][5]
        ) / sum_of_heights
        mass_final_FB_S2.append(mass_final_FB_value)


#INNER COLUMNS CONNECTED TO OUTRIGGER
#Inner Columns: average value over total structure height
```

```
mass_final_IC_Out_S2 = []  # Initialize an empty list

for i in range(len(mass_S2)):
    sum_of_heights = sum(height_zones_length_S2[i][0:6])
    if sum_of_heights == 0 or np.isnan(sum_of_heights):
        mass_final_IC_Out_value = np.nan
    else:
        mass_final_IC_Out_value = (
            mass_S2.iloc[i, 32] * height_zones_length_S2[i][0]
            + mass_S2.iloc[i, 33] * height_zones_length_S2[i][1]
            + mass_S2.iloc[i, 34] * height_zones_length_S2[i][2]
            + mass_S2.iloc[i, 35] * height_zones_length_S2[i][3]
            + mass_S2.iloc[i, 36] * height_zones_length_S2[i][4]
            + mass_S2.iloc[i, 37] * height_zones_length_S2[i][5]
        ) / sum_of_heights
        mass_final_IC_Out_S2.append(mass_final_IC_Out_value)


#INNER COLUMNS NOT CONNECTED TO OUTRIGGER
#Inner Columns: average value over total structure height
mass_final_IC_noOut_S2 = []  # Initialize an empty list

for i in range(len(mass_S2)):
    sum_of_heights = sum(height_zones_length_S2[i][0:6])
    if sum_of_heights == 0 or np.isnan(sum_of_heights):
        mass_final_IC_noOut_value = np.nan
    else:
        mass_final_IC_noOut_value = (
            mass_S2.iloc[i, 38] * height_zones_length_S2[i][0]
            + mass_S2.iloc[i, 39] * height_zones_length_S2[i][1]
            + mass_S2.iloc[i, 40] * height_zones_length_S2[i][2]
            + mass_S2.iloc[i, 41] * height_zones_length_S2[i][3]
            + mass_S2.iloc[i, 42] * height_zones_length_S2[i][4]
            + mass_S2.iloc[i, 43] * height_zones_length_S2[i][5]
        ) / sum_of_heights
        mass_final_IC_noOut_S2.append(mass_final_IC_noOut_value)
```

In [ ]:
```
mass_S2['Mass Final OB'] = mass_final_OB_S2
mass_S2['Mass Final OC Out'] = mass_final_OC_Out_S2
mass_S2['Mass Final OC no Out'] = mass_final_OC_noOut_S2
mass_S2['Mass Final FB'] = mass_final_FB_S2
mass_S2['Mass Final IC Out'] = mass_final_IC_Out_S2
mass_S2['Mass Final IC no Out'] = mass_final_IC_noOut_S2
```

Material Split

In [ ]:
```
mass_S2['Mass Total Steel'] = (mass_S2['Mass Final OB'] + mass_S2['Mass Final OC Out'] + mass_S2['Mass Final OC no Out']
                               + mass_S2['Mass Final FB'] + mass_S2['Mass Final IC Out'] + mass_S2['Mass Final IC no Out']
                               + mass_S2['out:Average Total Mass Outrigger [kg/m2GFA]'])

mass_S2['Mass Total Floors'] = mass_S2.iloc[:, 2:8].sum(axis=1) / mass_S2.iloc[:, 47]
mass_S2['Mass Total Concrete Core'] = mass_S2.iloc[:, 45]
mass_S2['Mass Total'] = mass_S2['Mass Total Steel'] + mass_S2['Mass Total Concrete Core']
```

Performance Parameters Variables

i) Costs

In [ ]:
```
#Prefab Concrete Reinforced
pc_rc_price_area = 850                          # concrete price [€/m3]
density_pc_rc = 2500                            # density prefab reinforced concrete [kg/m3]
pc_rc_price = pc_rc_price_area / density_pc_rc  # concrete price [€/kg]

#Hollow Core
hc_price_area = 110                             # hollow core slab price (h=320mm) [€/m2]
density_hc = 429                                # hollow core slab mass per area (h=320mm) [kg/m2]
hc_price = hc_price_area / density_hc           # hollow core slab price (h=320mm) [€/kg]

#Steel
s_price = 3.5                                   # steel price S355 [€/kg]


print('Price: prefab reinforced concrete =', round(pc_rc_price,3), '€/kg')
print('Price: hollow core concrete =', round(hc_price,3), '€/kg')
print('Price: steel =', round(s_price,3), '€/kg')
```

i) Embodied Carbon

In [ ]:
```
CO2_sprice = 0.05         # shaduw price CO2 eq [€/kgCO2eq]

pc_rc_EmCO2 = 0.238       # EmCO2 precast reinforced [kgCO2eq/kg]
hc_EmCO2 = 0.155          # EmCO2 hollow core [kgCO2eq/kg]
s_EmCO2 = 1.12            # EmCO2 steel S355 [kgCO2eq/kg]

print('Embodied Carbon: prefab reinforced concrete =', round(pc_rc_EmCO2,3), 'kgCO2eq/kg')
print('Embodied Carbon: hollow core concrete =', round(hc_EmCO2,3), 'kgCO2eq/kg')
print('Embodied Carbon: steel =', round(s_EmCO2,3), 'kgCO2eq/kg')
```

2) DATA VISUALIZATION

SCATTER PLOTS OF MASS

In [ ]:
```
# Define the color map
cmap = plt.get_cmap('Greens')

# Define the ranges and corresponding colors
ranges = [(0, 10), (10, 20), (20, 30), (30, 40), (40, 50), (50, 60), (60, 70), (70, 80)]
colors = np.linspace(0.1, 1, len(ranges))
```

```
# Create the scatter plot
plt.figure(figsize=(9,5))
for (start, end), color in zip(ranges, colors):
    mask = (mass_S1.iloc[:, 0] >= start) & (mass_S1.iloc[:, 0] < end)
    plt.scatter(mass_S1.loc[mask, mass_S1.columns[1]], mass_S1.loc[mask, 'Mass Total Steel'], color=cmap(color), label=f'{start}-{end}', marker='o')

plt.title('Scatter Plot of Mass Braced Framed Tube (S1): Strenght + Stiffness Optimization')
plt.xlabel('Height of the structure [m]')
plt.ylabel('Mass of structure [kg/m2GFA]')
plt.legend(bbox_to_anchor=(1.02, 1), loc='upper left', borderaxespad=0, title='Width of the structure [m]')
plt.xlim(0, 320)
plt.ylim(-250, 2500)

# Show the plot
plt.show()
```

In [ ]:
```
# Define the color map
cmap = plt.get_cmap('Blues')

# Create the scatter plot
plt.figure(figsize=(9,5))
for (start, end), color in zip(ranges, colors):
    mask = (mass_S2.iloc[:, 0] >= start) & (mass_S2.iloc[:, 0] < end)
    plt.scatter(mass_S2.loc[mask, mass_S2.columns[1]], mass_S2.loc[mask, 'Mass Total'], color=cmap(color), label=f'{start}-{end}', marker='o')

plt.title('Scatter Plot of Mass Outrigger (S2): Strenght + Stiffness Optimization')
plt.xlabel('Height of the structure [m]')
plt.ylabel('Mass of structure [kg/m2GFA]')
plt.legend(bbox_to_anchor=(1.02, 1), loc='upper left', borderaxespad=0, title='Width of the structure [m]')
plt.xlim(0, 320)
plt.ylim(-250, 2500)

# Show the plot
plt.show()
```

SCATTER PLOTS OF PERFORMANCE

Conversion of the data

In [ ]:
```
mass_S1['Total Costs'] = mass_S1['Mass Total Steel'] * s_price
mass_S1['Total EmCO2'] = mass_S1['Mass Total Steel'] * s_EmCO2*CO2_sprice
```

In [ ]:
```
mass_S2['Total Costs'] = mass_S2['Mass Total Steel'] * s_price + mass_S2['Mass Total Concrete Core'] * pc_rc_price
mass_S2['Total EmCO2'] = mass_S2['Mass Total Steel'] * s_EmCO2*CO2_sprice + mass_S2['Mass Total Concrete Core'] * pc_rc_EmCO2*CO2_sprice
```

Scatter plots of costs

Total scatter plots of costs for both S1 and S2

In [ ]:
```
plt.figure(figsize=(9,5))

cmap = plt.get_cmap('Greens')
for (start, end), color in zip(ranges, colors):
    mask = (mass_S1.iloc[:, 0] >= start) & (mass_S1.iloc[:, 0] < end)
    plt.scatter(mass_S1.loc[mask, mass_S1.columns[1]], mass_S1.loc[mask, 'Total Costs'], color=cmap(color), label=f'S1: {start}-{end}', marker='o')


cmap = plt.get_cmap('Blues')
for (start, end), color in zip(ranges, colors):
    mask = (mass_S2.iloc[:, 0] >= start) & (mass_S2.iloc[:, 0] < end)
    plt.scatter(mass_S2.loc[mask, mass_S2.columns[1]], mass_S2.loc[mask, 'Total Costs'], color=cmap(color), label=f'S2: {start}-{end}', marker='o')


plt.title('Scatter Plot of Structural Costs of Braced Framed Tube (S1) and Outrigger (S2)')
plt.xlabel('Height of the structure [m]')
plt.ylabel('Total Structural Costs [€/m2GFA]')
plt.legend(bbox_to_anchor=(1.02, 1), loc='upper left', borderaxespad=0, title='Width of the structure [m]')
#plt.xlim(0, 320)
plt.ylim(0, 25000)

# Show the plot
plt.show()
```

Zoomed-In View of scatterplot of costs

In [ ]:
```
plt.figure(figsize=(9,5))

cmap = plt.get_cmap('Greens')
for (start, end), color in zip(ranges, colors):
    mask = (mass_S1.iloc[:, 0] >= start) & (mass_S1.iloc[:, 0] < end)
    plt.scatter(mass_S1.loc[mask, mass_S1.columns[1]], mass_S1.loc[mask, 'Total Costs'], color=cmap(color), label=f'S1: {start}-{end}', marker='o')


cmap = plt.get_cmap('Blues')
for (start, end), color in zip(ranges, colors):
    mask = (mass_S2.iloc[:, 0] >= start) & (mass_S2.iloc[:, 0] < end)
    plt.scatter(mass_S2.loc[mask, mass_S2.columns[1]], mass_S2.loc[mask, 'Total Costs'], color=cmap(color), label=f'S2: {start}-{end}', marker='o')


plt.title('Zoomed-In: Scatter Plot of Structural Costs of Braced Framed Tube (S1) and Outrigger (S2)')
plt.xlabel('Height of the structure [m]')
plt.ylabel('Total Structural Costs [€/m2GFA]')
plt.legend(bbox_to_anchor=(1.02, 1), loc='upper left', borderaxespad=0, title='Width of the structure [m]')
#plt.xlim(0, 320)
plt.ylim(0, 2000)

# Show the plot
plt.show()
```

Scatter plot of costs for only S1

In [ ]:
```
plt.figure(figsize=(9,5))

#cmap = plt.get_cmap('Greens')
#for (start, end), color in zip(ranges, colors):
#    mask = (mass_S1.iloc[:, 0] >= start) & (mass_S1.iloc[:, 0] < end)
#    plt.scatter(mass_S1.loc[mask, mass_S1.columns[1]], mass_S1.loc[mask, 'Total Costs'], color=cmap(color), label=f'S1: {start}-{end}', marker='o')


cmap = plt.get_cmap('Blues')
for (start, end), color in zip(ranges, colors):
    mask = (mass_S2.iloc[:, 0] >= start) & (mass_S2.iloc[:, 0] < end)
    plt.scatter(mass_S2.loc[mask, mass_S2.columns[1]], mass_S2.loc[mask, 'Total Costs'], color=cmap(color), label=f'S2: {start}-{end}', marker='o')


plt.title('Scatter Plot of Structural Costs of Outrigger System (S2)')
plt.xlabel('Height of the structure [m]')
plt.ylabel('Total Structural Costs [€/m2GFA]')
plt.legend(bbox_to_anchor=(1.02, 1), loc='upper left', borderaxespad=0, title='Width of the structure [m]')
plt.xlim(0, 320)
plt.ylim(0, 6000)

# Show the plot
plt.show()
```

Scatter plots of embodied carbon

Total scatter plot of embodied carbon of only S1

In [ ]:
```
import matplotlib.pyplot as plt

plt.figure(figsize=(9,5))

cmap = plt.get_cmap('GnBu')
for (start, end), color in zip(ranges, colors):
    mask = (mass_S1.iloc[:, 0] >= start) & (mass_S1.iloc[:, 0] < end)
    width_count = mask.sum()  # Calculate the count of widths falling within the range
    plt.scatter(mass_S1.loc[mask, mass_S1.columns[1]], mass_S1.loc[mask, 'Total EmCO2'], color=cmap(color), label=f'S1: {start}-{end} (n = {width_cour

#cmap = plt.get_cmap('RdPu')
#for (start, end), color in zip(ranges, colors):
#    mask = (mass_S2.iloc[:, 0] >= start) & (mass_S2.iloc[:, 0] < end)
#    width_count = mask.sum()  # Calculate the count of widths falling within the range
#    plt.scatter(mass_S2.loc[mask, mass_S2.columns[1]], mass_S2.loc[mask, 'Total EmCO2'], color=cmap(color), label=f'S2: {start}-{end}', marker='o')

plt.title('Scatter Plot of Environmental Costs of Braced Framed Tube (S1)')
plt.xlabel('Height of the structure [m]')
plt.ylabel('Total Environmental Costs [€/m2GFA]')
plt.legend(bbox_to_anchor=(1.02, 1), loc='upper left', borderaxespad=0, title='Width of the structure [m]')
plt.xlim(0)
plt.ylim(0,100)
plt.show()
```

Scatterplot over volume

In [ ]:
```
X_S1 = mass_S1[['in:Length [m]', 'in:Height [m]']]
y_S1 = mass_S1[['Total Costs', 'Total EmCO2']]

X_S2 = mass_S2[['in:Length [m]', 'in:Height [m]']]
y_S2 = mass_S2[['Total Costs', 'Total EmCO2']]
```

In [ ]:
```
X_S1 = X_S1.rename(columns={'in:Length [m]':'Width of floorplan [m]', 'in:Height [m]':'Height of structure [m]'})
X_S2 = X_S2.rename(columns={'in:Length [m]':'Width of floorplan [m]', 'in:Height [m]':'Height of structure [m]'})
```

Pairplot of total costs per GFA for the volume of the Structure [m3]

In [ ]:
```
X_S1_combined = X_S1['Width of floorplan [m]'] * X_S1['Width of floorplan [m]'] * X_S1['Height of structure [m]']
X_S2_combined = X_S2['Width of floorplan [m]'] * X_S2['Width of floorplan [m]'] * X_S2['Height of structure [m]']

# Create subplots
fig, axes = plt.subplots(1, 2, figsize=(15, 5))

# Plot for Total Costs per GFA
axes[0].scatter(X_S1_combined, y_S1['Total Costs'], color=color_lightgreen, alpha = 0.7, label='S1: input data')
axes[0].scatter(X_S2_combined, y_S2['Total Costs'], color=color_lightblue, alpha = 0.7, label='S2: input data')

# Plot polynomial regression line
x_values_S1_combined = np.linspace(min(X_S1_combined), max(X_S1_combined), 100)
x_values_S2_combined = np.linspace(min(X_S2_combined), max(X_S2_combined), 100)

#axes[0].plot(x_values_S1_combined, poly_function_costs_S1(x_values_S1_combined), color=color_green, label=f'S1: Braced Framed Tube (Degree {degree})
#axes[0].plot(x_values_S2_combined, poly_function_costs_S2(x_values_S2_combined), color=color_darkblue, label=f'S2: Outrigger (Degree {degree})')

axes[0].set_title('Zoomed-In Volume of Structure [m3] vs Total Structural Costs [€/m2GFA]')
axes[0].set_xlabel('Volume of the Structure [m3]')
axes[0].set_ylabel('Total Structural Costs [€/m2GFA]')
axes[0].set_ylim(0, 7000)
axes[0].set_xlim(0)
axes[0].legend()

# Plot for Total Embodied Carbon per GFA
axes[1].scatter(X_S1_combined, y_S1['Total EmCO2'], color=color_lightgreen, alpha = 0.7, label='S1: input data')
axes[1].scatter(X_S2_combined, y_S2['Total EmCO2'], color=color_lightblue, alpha = 0.7, label='S2: input data')

# Plot polynomial regression line
#axes[1].plot(x_values_S1_combined, poly_function_EmCO2_S1(x_values_S1_combined), color=color_green, label=f'S1: Braced Framed Tube (Degree {degree})
#axes[1].plot(x_values_S2_combined, poly_function_EmCO2_S2(x_values_S2_combined), color=color_darkblue, label=f'S2: Outrigger (Degree {degree})')

axes[1].set_title('Zoomed-In: Volume of Structure [m3] vs Total Environmental Costs [€/m2GFA]')
```

```
axes[1].set_xlabel('Volume of the Structure [m3]')
axes[1].set_ylabel('Total Environmental Costs [€/m2GFA]')
axes[1].set_ylim(0, 130)
axes[1].set_xlim(0)
axes[1].legend()


# Adjust Layout
plt.tight_layout()
plt.show()
```

In [ ]:
```
plt.figure(figsize=(9,5))
plt.scatter(X_S1_combined, y_S1['Total Costs'], color=color_lightgreen, alpha = 0.7, label='S1: input data')
plt.scatter(X_S2_combined, y_S2['Total Costs'], color=color_lightblue, alpha = 0.7, label='S2: input data')

plt.title('Zoomed-In: Volume of Structure [m3] vs Total Structural Costs [€/m2GFA]')
plt.xlabel('Volume of the Structure [m3]')
plt.ylabel('Total Structural Costs [€/m2GFA]')
plt.ylim(0, 1250)
plt.xlim(0)
plt.legend()

# Show the plot
plt.show()
```

PAIRPLOT OF COSTS AGAINST EMBODIED CARBON

In [ ]:
```
# Extract the relevant columns
fig, axes = plt.subplots(1, 2, figsize=(15, 5))
costs_S1 = y_S1['Total Costs']
EmCO2_S1 = y_S1['Total EmCO2']

costs_S2 = y_S2['Total Costs']
EmCO2_S2 = y_S2['Total EmCO2']

volumne_S1 = X_S1_combined
volumne_S2 = X_S2_combined

max_x_S1 = max(EmCO2_S1)
max_x_S2 = max(EmCO2_S2)
max_x = max(max_x_S1, max_x_S2)

max_y_S1 = max(costs_S1)
max_y_S2 = max(costs_S2)
max_y = max(max_y_S1, max_y_S2)

min_x_S1 = min(EmCO2_S1)
min_x_S2 = min(EmCO2_S2)
min_x = min(max_x_S1, max_x_S2)

x_S1 = np.linspace(0, max_x_S1, 1000)
x_S2 = np.linspace(0, max_x_S2, 1000)

# Fit polynomial regression lines
degree = 1  # the degree of the polynomial can be adjusted

# Braced Framed Tube
coefficients_S1 = np.polyfit(EmCO2_S1, costs_S1, degree)
slope_S1 = coefficients_S1[0]  # slope coefficient
poly_function_S1 = np.poly1d(coefficients_S1)

# Outrigger
coefficients_S2 = np.polyfit(EmCO2_S2, costs_S2, degree)
slope_S2 = coefficients_S2[0]  # slope coefficient
poly_function_S2 = np.poly1d(coefficients_S2)

# Create subplots
# Plot for Total Costs per GFA
axes[0].scatter(EmCO2_S1, costs_S1, color=color_lightgreen, alpha=0.5)
axes[0].plot(x_S1, poly_function_S1(x_S1), color=color_green, label=f'S1: Braced Framed Tube')

axes[0].set_title(f'Structural Costs versus Environmental Costs for Braced Framed Tube (S1)')
axes[0].set_xlabel('Total Environmental Costs [€/mGFA2]')
axes[0].set_ylabel('Total Structural Costs [€/m2GFA]')
axes[0].set_ylim(0, max_y_S1)
axes[0].set_xlim(0, max_x_S1)
axes[0].annotate(f'Slope coefficient: {slope_S1:.2f}', xy=(0.02, 0.85), xycoords='axes fraction', fontsize=12)
axes[0].legend(loc = 'upper left')

# Plot for Total Embodied Carbon per GFA
axes[1].scatter(EmCO2_S2, costs_S2, color=color_lightblue, alpha=0.5)
axes[1].plot(x_S2, poly_function_S2(x_S2), color=color_darkblue, label=f'S2: Outrigger')


axes[1].set_title(f'Structural Costs versus Environmental Costs for Outrigger (S2)')
axes[1].set_xlabel('Total Environmental Costs [€/m2GFA]')
axes[1].set_ylabel('Total Structural Costs [€/m2GFA]')
axes[1].set_ylim(0, max_y_S2)
axes[1].set_xlim(0, max_x_S2)
axes[1].annotate(f'Slope coefficient: {slope_S2:.2f}', xy=(0.02, 0.85), xycoords='axes fraction', fontsize=12)
axes[1].legend(loc='upper left')

# Adjust Layout
plt.tight_layout()
plt.show()
```

In [ ]:
```
print('The function of the total costs vs the total embodied carbon for S1: Braced Tube is:', poly_function_S1)
print('')
print('The function of the total costs vs the total embodied carbon for S2: Outrigger is:', poly_function_S2)
```

VISUALIZATION WITH ONE VARIABLE CONSTANT

i) Visualization with width of the structure constant

In [ ]:
```
constant_variable = 'in:Length [m]'
constant_value = (43.2) # Set the constant value you want to use

# Filter data for the specified constant value
filtered_data_S1 = mass_S1[mass_S1[constant_variable] == constant_value]
filtered_data_S2 = mass_S2[mass_S2[constant_variable] == constant_value]
```

In [ ]:
```
# Extract the relevant columns
X_filtered_S1 = filtered_data_S1['in:Height [m]']
y_costs_S1 = filtered_data_S1['Total Costs']
y_carbon_S1 = filtered_data_S1['Total EmCO2']

X_filtered_S2 = filtered_data_S2['in:Height [m]']
y_costs_S2 = filtered_data_S2['Total Costs']
y_carbon_S2 = filtered_data_S2['Total EmCO2']


# Fit polynomial regression lines
degree = 2

# Braced Framed Tube
coefficients_costs_S1 = np.polyfit(X_filtered_S1, y_costs_S1, degree)
poly_function_costs_S1 = np.poly1d(coefficients_costs_S1)
coefficients_carbon_S1 = np.polyfit(X_filtered_S1, y_carbon_S1, degree)
poly_function_carbon_S1 = np.poly1d(coefficients_carbon_S1)

# Outrigger
coefficients_costs_S2 = np.polyfit(X_filtered_S2, y_costs_S2, degree)
poly_function_costs_S2 = np.poly1d(coefficients_costs_S2)
coefficients_carbon_S2 = np.polyfit(X_filtered_S2, y_carbon_S2, degree)
poly_function_carbon_S2 = np.poly1d(coefficients_carbon_S2)


# Create subplots
plt.figure(figsize=(15, 5))

# Plot for Total Costs per GFA
plt.subplot(1, 2, 1)

plt.scatter(X_filtered_S1, y_costs_S1, color=color_lightgreen, label='S1: Braced Framed Tube', alpha=0.7)
plt.scatter(X_filtered_S2, y_costs_S2, color=color_lightblue, label='S2: Outrigger', alpha=0.7)

#plt.plot(X_filtered_S1, poly_function_costs_S1(X_filtered_S1), color=color_green, label=f'S1: Regression (Degree {degree})')
#plt.plot(X_filtered_S2, poly_function_costs_S2(X_filtered_S2), color=color_darkblue, label=f'S2: Regression (Degree {degree})')

plt.xlim(0)
plt.ylim(0, 3000)

plt.title(f'Total Structural Costs per m2GFA for Floor Plan Length of {constant_value} m')
plt.xlabel('Height of structure [m]')
plt.ylabel('Total Structural Costs per m2GFA [€/m2]')
plt.legend()

# Plot for Total Embodied Carbon per GFA
plt.subplot(1, 2, 2)

plt.scatter(X_filtered_S1, y_carbon_S1, color=color_lightgreen, label='S1: Braced Framed Tube', alpha=0.7)
plt.scatter(X_filtered_S2, y_carbon_S2, color=color_lightblue, label='S2: Outrigger', alpha=0.7)

#plt.plot(X_filtered_S1, poly_function_carbon_S1(X_filtered_S1), color=color_green, label=f'S1: Regression (Degree {degree})')
#plt.plot(X_filtered_S2, poly_function_carbon_S2(X_filtered_S2), color=color_darkblue, label=f'S2: Regression (Degree {degree})')

plt.xlim(0)
plt.ylim(0)

plt.title(f'Total Environmental Costs per m2GFA for Floor Plan Length of {constant_value} m')
plt.xlabel('Height of structure [m]')
plt.ylabel('Total Environmental Costs per GFA [€/m2]')
plt.legend()


# Adjust layout
plt.tight_layout()
plt.show()
```

ii) Visualization with height of the structure constant

In [ ]:
```
constant_variable = 'in:Height [m]'
constant_value = 150  # Set the constant value you want to use

# Filter data for the specified constant value
filtered_data_S1 = mass_S1[mass_S1[constant_variable] == constant_value]
filtered_data_S2 = mass_S2[mass_S2[constant_variable] == constant_value]
```

In [ ]:
```
# Extract the relevant columns
X_filtered_S1 = filtered_data_S1['in:Length [m]']
y_costs_S1 = filtered_data_S1['Total Costs']
y_carbon_S1 = filtered_data_S1['Total EmCO2']

X_filtered_S2 = filtered_data_S2['in:Length [m]']
y_costs_S2 = filtered_data_S2['Total Costs']
y_carbon_S2 = filtered_data_S2['Total EmCO2']


# Create subplots
plt.figure(figsize=(15, 5))

# Plot for Total Costs per GFA
plt.subplot(1, 2, 1)

plt.scatter(X_filtered_S1, y_costs_S1, color=color_lightgreen, label='S1: Braced Framed Tube', alpha=0.7)
plt.scatter(X_filtered_S2, y_costs_S2, color=color_lightblue, label='S2: Outrigger', alpha=0.7)
```

```
#plt.plot(X_filtered_S1, poly_function_costs_S1(X_filtered_S1), color=color_green, label=f'S1: Regression (Degree {degree})')
#plt.plot(X_filtered_S2, poly_function_costs_S2(X_filtered_S2), color=color_darkblue, label=f'S2: Regression (Degree {degree})')

#plt.xlim(0, max(X_filtered_S1, X_filtered_S2) + 20)
plt.ylim(0, 1250)

plt.title(f'Total Structural Costs per m2GFA for Height of {constant_value} m')
plt.xlabel('Length of the floorplan [m]')
plt.ylabel('Total Structural Costs per m2GFA [€/m2]')
plt.legend()

# Plot for Total Embodied Carbon per GFA
plt.subplot(1, 2, 2)

plt.scatter(X_filtered_S1, y_carbon_S1, color=color_lightgreen, label='S1: Braced Framed Tube', alpha=0.7)
plt.scatter(X_filtered_S2, y_carbon_S2, color=color_lightblue, label='S2: Outrigger', alpha=0.7)

#plt.plot(X_filtered_S1, poly_function_carbon_S1(X_filtered_S1), color=color_green, label=f'S1: Regression (Degree {degree})')
#plt.plot(X_filtered_S2, poly_function_carbon_S2(X_filtered_S2), color=color_darkblue, label=f'S2: Regression (Degree {degree})')

#plt.xlim(0, max(X_filtered_S1, X_filtered_S2) + 20)
#plt.ylim(0, max(y_costs_S1, y_costs_S2) + 2500)

plt.title(f'Total Environmental Costs per m2GFA for Height of {constant_value} m')
plt.xlabel('Length of the floorplan [m]')
plt.ylabel('Total Environmental Costs per m2GFA [€/m2]')
plt.legend()

# Adjust layout
plt.tight_layout()
plt.show()
```

In [ ]:
```
plt.figure(figsize=(9,5))
plt.scatter(X_filtered_S1, y_costs_S1, color=color_lightgreen, label='S1: Braced Framed Tube', alpha=0.7)
plt.scatter(X_filtered_S2, y_costs_S2, color=color_lightblue, label='S2: Outrigger', alpha=0.7)

plt.title('Zoomed-In: Total Costs per GFA for Height of 150 m')
plt.xlabel('Width of the floor plan [m]')
plt.ylabel('Total Costs per GFA [€/m2]')
plt.ylim(0, 12000)
plt.xlim()
plt.legend()

# Show the plot
plt.show()
```

3) COMPARISON OF COMPUTATIONAL TIME

In [ ]:
```
methods = ['Analytical', 'Parametric', 'ML (ANN)']

x = np.linspace(0, 500, 500)
analytical = 20*x
parametric = 80 + 1/3*x
ann = 80 + 40 + 1/3600*x

# Create subplots
plt.figure(figsize=(5, 5))

# Plot for Total Costs per GFA
x_equal = np.argmin(np.abs(parametric - ann))


plt.plot(x, analytical, label = 'Analytical', color =color_lightblue)
plt.plot(x, parametric, label = 'Parametric', color =color_green)
plt.plot(x, ann, label = 'ANN', color=color_purple)
plt.scatter(x[x_equal], parametric[x_equal], color=color_orange, label=f'x = {np.round(x[x_equal])}')

plt.xlabel('Iterations')
plt.ylabel('Time (hours)')
plt.title('Building and Running Times for Different Methods')
plt.xlim(0, 500)
plt.ylim(0, 500)
plt.legend(loc = 'upper right')
plt.grid(True)
plt.show()

print(f"The x value where Parametric and ANN are equal: {x[x_equal]}")
```

4) DETERMINATION OF FORCE DISTRIBUTION : STABILITY FRAMEWORK VS FLOOR SYSTEM

i) S1: Braced Framed Tube

In [ ]:
```
mass_S1['Impact Stability'] = (mass_S1['Mass Final OC'] + mass_S1.iloc[:, 32]) * s_EmCO2
mass_S1['Impact Vertical'] = ((mass_S1['Mass Final IC'] + mass_S1['Mass Final FB'] + mass_S1['Mass Final OB']) * s_EmCO2) + (mass_S1['Mass Total Floor

mass_S1['Impact Total'] = mass_S1['Impact Vertical'] + mass_S1['Impact Stability']
```

In [ ]:
```
impact_S1_h100 = mass_S1[(mass_S1['in:Height [m]'] <= 100)]
impact_S1_h150 = mass_S1[(100 < mass_S1['in:Height [m]']) & (mass_S1['in:Height [m]'] <= 150)]
impact_S1_h200 = mass_S1[(150 < mass_S1['in:Height [m]']) & (mass_S1['in:Height [m]'] <= 200)]
impact_S1_h250 = mass_S1[(200 < mass_S1['in:Height [m]']) & (mass_S1['in:Height [m]'] <= 250)]
impact_S1_h300 = mass_S1[(250 < mass_S1['in:Height [m]']) & (mass_S1['in:Height [m]'] <= 300)]
```

In [ ]:
```
share_S1_range1 = []
for i in range(len(impact_S1_h100)):
    relative_share_S1 = impact_S1_h100['Impact Stability'] / impact_S1_h100['Impact Total']
```

```
    relative_share_S1 = relative_share_S1*100
    share_S1_range1 = np.append(share_S1_range1, relative_share_S1)

share_S1_range2 = []
for i in range(len(impact_S1_h150)):
    relative_share_S1 = impact_S1_h150['Impact Stability'] / impact_S1_h150['Impact Total']
    relative_share_S1 = relative_share_S1*100
    share_S1_range2 = np.append(share_S1_range2, relative_share_S1)

share_S1_range3 = []
for i in range(len(impact_S1_h200)):
    relative_share_S1 = impact_S1_h200['Impact Stability'] / impact_S1_h200['Impact Total']
    relative_share_S1 = relative_share_S1*100
    share_S1_range3 = np.append(share_S1_range3, relative_share_S1)

share_S1_range4 = []
for i in range(len(impact_S1_h250)):
    relative_share_S1 = impact_S1_h250['Impact Stability'] / impact_S1_h250['Impact Total']
    relative_share_S1 = relative_share_S1*100
    share_S1_range4 = np.append(share_S1_range4, relative_share_S1)

share_S1_range5 = []
for i in range(len(impact_S1_h300)):
    relative_share_S1 = impact_S1_h300['Impact Stability'] / impact_S1_h300['Impact Total']
    relative_share_S1 = relative_share_S1*100
    share_S1_range5 = np.append(share_S1_range5, relative_share_S1)
```

In [ ]:
```
print(len(impact_S1_h100))
print(len(impact_S1_h150))
print(len(impact_S1_h200))
print(len(impact_S1_h250))
print(len(impact_S1_h300))
```

In [ ]:
```
data_share_S1 = [share_S1_range1, share_S1_range2, share_S1_range3, share_S1_range4, share_S1_range5]
medians = [np.median(x) for x in data_share_S1]
q1 = [np.percentile(x, 25) for x in data_share_S1]

q3 = [np.percentile(x, 75) for x in data_share_S1]
iqr = [q3[i] - q1[i] for i in range(len(data_share_S1))]
upper_whisker = [q3[i] + 1.5 * iqr[i] for i in range(len(data_share_S1))]
minimum = [min(data_share_S1[i]) for i in range(len(data_share_S1))]
lower_whisker = [q1[i] - 1.5 * iqr[i] for i in range(len(data_share_S1))]
lower_whisker = [max(minimum[i], lower_whisker[i]) for i in range(len(data_share_S1))]
print(medians)
```

In [ ]:
```
plt.figure(figsize=(14, 6))
bp = plt.boxplot(data_share_S1, patch_artist=True, showfliers=False, labels=['48 - 100 m', '100 - 150 m', '150 - 200 m', '200 - 250 m', '250 - 300 m']

# Customizing box colors
colors = [color_green , color_green, color_green, color_green, color_green]
for box, color in zip(bp['boxes'], colors):
    box.set(color='black', linewidth=0.5)
    box.set(facecolor=color)

for median in bp['medians']:
    median.set(color='black')

plt.title('Share of EmCO2 of Stability Framework of S1 to Total EmCO2 (= Superstructure + Floors)', fontsize = 14)
plt.xlabel('Height of the Structure', fontsize = 14)
plt.ylabel('Percentage [%]', fontsize = 14)
plt.xticks(fontsize=13)
plt.ylim(0,100)
plt.grid(True)
plt.show()
```

i) S2: Outrigger

In [ ]:
```
mass_S2
```

In [ ]:
```
ratio_IC_Out_to_stability = 0.46
ratio_IC_noOut_to_stability = 0.22
ratio_OC_Out_to_stability = 0.64
ratio_OC_noOut_to_stability = 0.27
```

In [ ]:
```
mass_S2['Impact Stability'] = (((ratio_IC_Out_to_stability) * mass_S2['Mass Final IC Out']
                              + (ratio_IC_noOut_to_stability) * mass_S2['Mass Final IC no Out']
                              + (ratio_OC_Out_to_stability) * mass_S2['Mass Final OC Out']
                              + (ratio_OC_noOut_to_stability) * mass_S2['Mass Final OC no Out']
                              +  mass_S2.iloc[:, 44]) * s_EmCO2
                              +  mass_S2.iloc[:, 45] * pc_rc_EmCO2)

mass_S2['Impact Vertical'] = (((1-ratio_IC_Out_to_stability) * mass_S2['Mass Final IC Out']
                              + (1-ratio_IC_noOut_to_stability) * mass_S2['Mass Final IC no Out']
                              + (1-ratio_OC_Out_to_stability) * mass_S2['Mass Final OC Out']
                              + (1-ratio_OC_noOut_to_stability) * mass_S2['Mass Final OC no Out']
                              +  mass_S2['Mass Final FB'] + mass_S2['Mass Final OB']) * s_EmCO2
                              +  (mass_S2['Mass Total Floors'] * hc_EmCO2))

mass_S2['Impact Total'] = mass_S2['Impact Vertical'] + mass_S2['Impact Stability']
```

In [ ]:
```
impact_S2_h100 = mass_S2[(mass_S2['in:Height [m]'] <= 100)]
impact_S2_h150 = mass_S2[(100 < mass_S2['in:Height [m]']) & (mass_S2['in:Height [m]'] <= 150)]
impact_S2_h200 = mass_S2[(150 < mass_S2['in:Height [m]']) & (mass_S2['in:Height [m]'] <= 200)]
impact_S2_h250 = mass_S2[(200 < mass_S2['in:Height [m]']) & (mass_S2['in:Height [m]'] <= 250)]
impact_S2_h300 = mass_S2[(250 < mass_S2['in:Height [m]']) & (mass_S2['in:Height [m]'] <= 300)]
```

In [ ]:
```
share_S2_range1 = []
for i in range(len(impact_S2_h100)):
    relative_share_S2 = impact_S2_h100['Impact Stability'] / impact_S2_h100['Impact Total']
    relative_share_S2 = relative_share_S2*100
    share_S2_range1 = np.append(share_S2_range1, relative_share_S2)
```

```
share_S2_range2 = []
for i in range(len(impact_S2_h150)):
    relative_share_S2 = impact_S2_h150['Impact Stability'] / impact_S2_h150['Impact Total']
    relative_share_S2 = relative_share_S2*100
    share_S2_range2 = np.append(share_S2_range2, relative_share_S2)

share_S2_range3 = []
for i in range(len(impact_S2_h200)):
    relative_share_S2 = impact_S2_h200['Impact Stability'] / impact_S2_h200['Impact Total']
    relative_share_S2 = relative_share_S2*100
    share_S2_range3 = np.append(share_S2_range3, relative_share_S2)

share_S2_range4 = []
for i in range(len(impact_S2_h250)):
    relative_share_S2 = impact_S2_h250['Impact Stability'] / impact_S2_h250['Impact Total']
    relative_share_S2 = relative_share_S2*100
    share_S2_range4 = np.append(share_S2_range4, relative_share_S2)

share_S2_range5 = []
for i in range(len(impact_S2_h300)):
    relative_share_S2 = impact_S2_h300['Impact Stability'] / impact_S2_h300['Impact Total']
    relative_share_S2 = relative_share_S2*100
    share_S2_range5 = np.append(share_S2_range5, relative_share_S2)
```

In [ ]:
```
print(len(impact_S2_h100))
print(len(impact_S2_h150))
print(len(impact_S2_h200))
print(len(impact_S2_h250))
print(len(impact_S2_h300))
```

In [ ]:
```
data_share_S2 = [share_S2_range1, share_S2_range2, share_S2_range3, share_S2_range4, share_S2_range5]
medians = [np.median(x) for x in data_share_S2]
q1 = [np.percentile(x, 25) for x in data_share_S2]

q3 = [np.percentile(x, 75) for x in data_share_S2]
iqr = [q3[i] - q1[i] for i in range(len(data_share_S2))]
upper_whisker = [q3[i] + 1.5 * iqr[i] for i in range(len(data_share_S2))]
minimum = [min(data_share_S2[i]) for i in range(len(data_share_S2))]
lower_whisker = [q1[i] - 1.5 * iqr[i] for i in range(len(data_share_S2))]
lower_whisker = [max(minimum[i], lower_whisker[i]) for i in range(len(data_share_S2))]
print(medians)
```

In [ ]:
```
plt.figure(figsize=(14, 6))
bp = plt.boxplot(data_share_S2, patch_artist=True, showfliers=False, labels=['48 - 100 m', '100 - 150 m', '150 - 200 m', '200 - 250 m', '250 - 300 m']

# Customizing box colors
colors = [color_darkblue , color_darkblue, color_darkblue, color_darkblue, color_darkblue]
for box, color in zip(bp['boxes'], colors):
    box.set(color='black', linewidth=0.5)
    box.set(facecolor=color)

for median in bp['medians']:
    median.set(color='black')

plt.title('Share of EmCO2 of Stability Framework of S2 to Total EmCO2 (= Superstructure + Floors)', fontsize = 14)
plt.xlabel('Height of the Structure', fontsize = 14)
plt.ylabel('Percentage [%]', fontsize = 14)
plt.xticks(fontsize=13)
plt.ylim(0,100)
plt.grid(True)
plt.show()
```

In [ ]:

In [ ]:

PREDICTION OF BEST STABILITY FRAMEWORK OF HIGH RISE STEEL BUILDINGS

COMPARISON BETWEEN BRACED FRAMED TUBE AND OUTRIGGER SYSTEM BASED ON COSTS OR EMBODIED CARBON

Emma Potijk (4715802)
MSc Thesis Structural Engineering
University: TU Delft
Company: Buro Happold
Period: 08/2023 - 05/2024

Import standard functions

In [ ]:
```
import pandas as pd
import numpy as np
from sklearn.model_selection import KFold
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_squared_error
from sklearn.neural_network import MLPRegressor
from geneticalgorithm import geneticalgorithm as ga
from sklearn.model_selection import train_test_split

import matplotlib.pyplot as plt
import seaborn as sns
sns.set(style="whitegrid")

import ipywidgets as widgets
from IPython.display import display
from ipywidgets import interact
import copy
import warnings
from mpl_toolkits.mplot3d import Axes3D

from sklearn.base import BaseEstimator
from tensorflow import keras
from keras.callbacks import EarlyStopping, ModelCheckpoint
from scipy.stats import norm
import math
```

In [ ]:
```
#pip install ipywidgets --upgrade
```

In [ ]:
```
warnings.filterwarnings(action='ignore', category=UserWarning)
```

In [ ]:
```
color_orange = '#FF914D'
color_lightblue = '#38B6FF'
color_green = '#589D62'
color_purple = '#CB6CE6'
color_darkblue = '#417DB9'
color_yellow = '#FFF3C2'
color_grey = '#A6A6A6'
color_lightgreen = '#8BE198'
```

--- CONTENT ---

1) Data preprocessing

a) Import data
b) Converting input features

2) Modelling procedure

a) Braced framed tube
    i) Splitting and scaling data
    ii) Building ANN model
    iii) Define fitness model and set up GA
    iv) Run and train ANN model
    v) Comparing model with test & train data

b) Outrigger
    i) Splitting and scaling data
    ii) Building ANN model
    iii) Define fitness model and set up GA
    iv) Run and train ANN model
    v) Comparing model with test & train data

3) Make predictions

a) Predictions regarding total costs and embodied carbon

4) Analysis of model results: Braced Framed Tube

a) Braced Framed Tube
    i) Comparison interpolation techniques
    ii) Graph of distribution of relative error
    iii) Determination of MAPE

b) Outrigger
    i) Comparison interpolation techniques
    ii) Graph of distribution of relative error
    iii) Determination of MAPE

5) Visualization with upper and lower band of RMSE

6) Comparison with input data

a) Comparison of braced framed tube
b) Comparison of outrigger

7) Comparison with literature

a) Division of number of stories
b) Division of floor area

8) Impact of optimizing stability framework

--- CODE ---

1) DATA PREPROCESSING

S1 = Braced Framed Tube System (grid size = 3.6 m)

S2 = Outrigger System (grid size = 1.8 m)

IMPORT DATA

i) S1: Braced Framed Tube

In [ ]:
```
file_path = r'C:\Users\epotijk\OneDrive - Buro Happold\MSc Thesis\Data Sets GH\Colibri_Braced_Framed_Tube\data.csv'
dataset_S1 = pd.read_csv(file_path, delimiter = ',')

dataset_S1 = dataset_S1.drop(columns = ['img'])
dataset_S1 = dataset_S1.loc[(dataset_S1 >= 0).all(axis=1)]
```

In [ ]:
```
ranges = [(2, 8), (8, 14), (14, 20), (20, 26)]

def check_descending(row):
    for start, end in ranges:
        for i in range(end-1, start-1, -1):  # Iterate in reverse order
            if i > start and row[i] > row[i-1]:  # Check if values are not descending
                row[i-1] = row[i]  # Replace smaller value with larger one
    return row

dataset_S1_v1 = dataset_S1.copy()
dataset_S1 = dataset_S1_v1.apply(check_descending, axis=1)
dataset_S1
```

ii) S1: Outrigger

In [ ]:
```
file_path = r'C:\Users\epotijk\OneDrive - Buro Happold\MSc Thesis\Data Sets GH\Colibri_Outrigger_DivisionColumns_v3\data.csv'
dataset_S2 = pd.read_csv(file_path, delimiter = ',')

dataset_S2 = dataset_S2.drop(columns = ['img'])
dataset_S2 = dataset_S2.loc[(dataset_S2 >= 0).all(axis=1)]
```

In [ ]:
```
dataset_S2 = dataset_S2.copy()
```

In [ ]:
```
mask = dataset_S2['out:Displacement Framework [cm]'] > dataset_S2['out:Max allowed displacement [cm]'] + 3
dataset_S2 = dataset_S2[~mask]
```

In [ ]:
```
dataset_S2.replace(-999.000000, 0.000000)
indices_to_drop6 = dataset_S2[dataset_S2['in:Number of grids'] == 6].index
indices_to_drop8 = dataset_S2[dataset_S2['in:Number of grids'] == 8].index

dataset_S2.drop(indices_to_drop6, inplace=True)
dataset_S2.drop(indices_to_drop8, inplace=True)
```

CONVERTING INPUT FEATURES

Geometry Variables

In [ ]:
```
#Convertion into length instead of number of grids or number of floors
dataset_S2 = dataset_S2.copy()
dataset_S1 = dataset_S1.copy()

grid_size_S1 = 3.6
grid_size_S2 = 1.8
floor_to_floor = 3.0

dataset_S1['in:Number of grids'] = dataset_S1['in:Number of grids'] * grid_size_S1
dataset_S1['in:Number of floors'] = dataset_S1['in:Number of floors'] * floor_to_floor

dataset_S2['in:Number of grids'] = dataset_S2.iloc[:,0] * grid_size_S2
dataset_S2['in:Number of floors'] = dataset_S2.iloc[:,1] * floor_to_floor

dataset_S1 = dataset_S1.rename(columns={'in:Number of grids' : 'in:Length [m]'})
dataset_S1 = dataset_S1.rename(columns={'in:Number of floors' : 'in:Height [m]'})

dataset_S2 = dataset_S2.rename(columns={'in:Number of grids' : 'in:Length [m]'})
dataset_S2 = dataset_S2.rename(columns={'in:Number of floors' : 'in:Height [m]'})
```

In [ ]:
```
mass_S1 = dataset_S1.copy()
mass_S2 = dataset_S2.copy()
```

In [ ]:
```
condition = (mass_S2['in:Length [m]'] == 54) & (mass_S2['in:Height [m]'] == 273)
mass_S2 = mass_S2[~condition]

condition = (mass_S2['in:Length [m]'] == 28.8) & (mass_S2['in:Height [m]'] == 219)
mass_S2 = mass_S2[~condition]
```

Determination Height Zones

In [ ]:
```
X_S1 = mass_S1[['in:Length [m]', 'in:Height [m]']]
X_S2 = mass_S2[['in:Length [m]', 'in:Height [m]']]
```

In [ ]:
```
def determine_wind_zones(X):
    if isinstance(X, pd.DataFrame):
        b = X.iloc[:, 0].values
        h = X.iloc[:, 1].values
    else:
        b = X[:, 0]
        h = X[:, 1]

    wind_zones = np.zeros_like(b, dtype=int)

    for i in range(len(b)):
        if h[i] <= b[i]:
            wind_zones[i] = 1
        elif b[i] <= h[i] < 2 * b[i]:
            wind_zones[i] = 2
        elif h[i] > 2 * b[i]:
            wind_zones[i] = 3

    return wind_zones

def determine_h_wind_zones(wind_zones, X):
    if isinstance(X, pd.DataFrame):
        b = X.iloc[:, 0].values
        h = X.iloc[:, 1].values
    else:
        b = X[:, 0]
        h = X[:, 1]

    z_e1 = np.zeros_like(wind_zones, dtype=float)
    z_e2 = np.zeros_like(wind_zones, dtype=float)
    z_e3 = np.zeros_like(wind_zones, dtype=float)

    floor_to_floor = 3

    for i, zone in enumerate(wind_zones):
        if zone == 1:
            z_e1[i] = h[i]
            z_e2[i] = 0
            z_e3[i] = 0
        elif zone == 2:
            z_e1[i] = b[i]
            z_e2[i] = h[i]
            z_e3[i] = 0
        elif zone == 3:
            z_e1[i] = (b[i] // floor_to_floor) * floor_to_floor
            z_e2[i] = h[i] - ((b[i] // floor_to_floor) * floor_to_floor)
            z_e3[i] = h[i]

    z_e1 = np.round(z_e1, 1)
    z_e2 = np.round(z_e2, 1)
    z_e3 = np.round(z_e3, 1)


    height_wind_zones = np.column_stack((z_e1, z_e2, z_e3))
    return height_wind_zones

def determine_height_zones(wind_zones, X, height_wind_zones):
    num_zones = np.zeros_like(wind_zones, dtype=float)

    def distribute_floors(total_floors, num_zones):
        if num_zones == 0:
            return [], 0

        base_floors_per_zone = total_floors // num_zones
        remaining_floors = total_floors % num_zones

        floors_per_zone = [base_floors_per_zone] * num_zones

        for i in range(int(remaining_floors)):
            floors_per_zone[i] += 1

        return floors_per_zone, len(floors_per_zone)

    for i in range(len(wind_zones)):
        if wind_zones[i] == 1:
            num_zones[i] = 1

        elif wind_zones[i] == 2:
            num_zones[i] = 2

        elif wind_zones[i] == 3:
            floors_2 = height_wind_zones[i, 1] - height_wind_zones[i, 0]
            ratio = (height_wind_zones[i, 1] - height_wind_zones[i, 0]) / height_wind_zones[i, 0]

            if ratio == 1.5:
                ratio = 2
            if ratio == 3.5:
                ratio = 4
            if ratio == 2.5:
                ratio = 3
            else:
                ratio = round(ratio)
                if ratio > 4:
                    ratio = 4
                else:
                    floors_distribution_windzone2, outcome_length = distribute_floors(floors_2, int(ratio))

            if ratio <= 1:
                num_zones[i] = 3

            elif ratio > 1:
                additional_zones = ratio
```

```
            additional_zones = min(additional_zones, 4)
            num_zones[i] = additional_zones + 2

    return num_zones
```

In [ ]:
```
wind_zones_S2 = determine_wind_zones(X_S2)
height_wind_zones_S2 = determine_h_wind_zones(wind_zones_S2, X_S2)
num_zones_S2 = mass_S2.iloc[:, 47]

wind_zones_S1 = determine_wind_zones(X_S1)
height_wind_zones_S1 = determine_h_wind_zones(wind_zones_S1, X_S1)
num_zones_S1 = mass_S1.iloc[:, 33]
```

In [ ]:
```
def determine_height_zones_length(height_wind_zones, num_zones):
    height_zones_length = []
    for i in range(len(num_zones)):
        if num_zones.iloc[i] == 2:
            l1 = height_wind_zones[i][0]
            l2 = height_wind_zones[i][1] - height_wind_zones[i][0]
            height_zones_length.append([l1, l2, 0, 0, 0, 0])
        if num_zones.iloc[i] == 3:
            l1 = height_wind_zones[i][0]
            l3 = height_wind_zones[i][2] - height_wind_zones[i][1]
            l2 = height_wind_zones[i][2] - l3 - l1
            height_zones_length.append([l1, l2, l3, 0, 0, 0])
        if num_zones.iloc[i] == 4:
            l1 = height_wind_zones[i][0]
            l4 = height_wind_zones[i][2] - height_wind_zones[i][1]
            l23 = height_wind_zones[i][2] - l4 - l1
            l2 = l23 / 2
            l3 = l23 / 2
            height_zones_length.append([l1, l2, l3, l4, 0, 0])
        if num_zones.iloc[i] == 5:
            l1 = height_wind_zones[i][0]
            l5 = height_wind_zones[i][2] - height_wind_zones[i][1]
            l234 = height_wind_zones[i][2] - l5 - l1
            l2 = l234 / 3
            l3 = l234 / 3
            l4 = l234 / 3
            height_zones_length.append([l1, l2, l3, l4, l5, 0])
        if num_zones.iloc[i] == 6:
            l1 = height_wind_zones[i][0]
            l6 = height_wind_zones[i][2] - height_wind_zones[i][1]
            l2345 = height_wind_zones[i][2] - l6 - l1
            l2 = l2345 / 4
            l3 = l2345 / 4
            l4 = l2345 / 4
            l5 = l2345 / 4
            height_zones_length.append([l1, l2, l3, l4, l5, l6])
    return height_zones_length
```

In [ ]:
```
height_zones_length_S2 = determine_height_zones_length(height_wind_zones_S2, num_zones_S2)
height_zones_length_S1 = determine_height_zones_length(height_wind_zones_S1, num_zones_S1)
```

In [ ]:
```
index_to_delete = 106

if 0 <= index_to_delete < len(height_zones_length_S2):
    del height_zones_length_S2[index_to_delete]
```

In [ ]:
```
index_label = mass_S2.iloc[106].name
mass_S2.drop(index_label, inplace=True)
```

Determination Average Mass per GFA over total Height of Structure

Braced Framed Tube

In [ ]:
```
#OUTER BEAMS
#Outer beams: average value over total structure height
mass_final_OB_S1 = []

for i in range(len(mass_S1)):
    sum_of_heights = sum(height_zones_length_S1[i][0:6])
    mass_final_OB_value = (
        mass_S1.iloc[i, 8] * height_zones_length_S1[i][0]
        + mass_S1.iloc[i, 9] * height_zones_length_S1[i][1]
        + mass_S1.iloc[i, 10] * height_zones_length_S1[i][2]
        + mass_S1.iloc[i, 11] * height_zones_length_S1[i][3]
        + mass_S1.iloc[i, 12] * height_zones_length_S1[i][4]
        + mass_S1.iloc[i, 13] * height_zones_length_S1[i][5]
    ) / sum_of_heights
    mass_final_OB_S1.append(mass_final_OB_value)


#OUTER COLUMNS
#Outer columns: average value over total structure height
mass_final_OC_S1 = []  # Initialize an empty list

for i in range(len(mass_S1)):
    sum_of_heights = sum(height_zones_length_S1[i][0:6])
    mass_final_OC_value = (
        mass_S1.iloc[i, 14] * height_zones_length_S1[i][0]
        + mass_S1.iloc[i, 15] * height_zones_length_S1[i][1]
        + mass_S1.iloc[i, 16] * height_zones_length_S1[i][2]
        + mass_S1.iloc[i, 17] * height_zones_length_S1[i][3]
        + mass_S1.iloc[i, 18] * height_zones_length_S1[i][4]
        + mass_S1.iloc[i, 19] * height_zones_length_S1[i][5]
    ) / sum_of_heights
    mass_final_OC_S1.append(mass_final_OC_value)


#FLOOR BEAMS
#Floor Beams: average value over total structure height
mass_final_FB_S1 = []  # Initialize an empty list

for i in range(len(mass_S1)):
    sum_of_heights = sum(height_zones_length_S1[i][0:6])
    mass_final_FB_value = (
        mass_S1.iloc[i, 20] * height_zones_length_S1[i][0]
        + mass_S1.iloc[i, 21] * height_zones_length_S1[i][1]
        + mass_S1.iloc[i, 22] * height_zones_length_S1[i][2]
        + mass_S1.iloc[i, 23] * height_zones_length_S1[i][3]
        + mass_S1.iloc[i, 24] * height_zones_length_S1[i][4]
        + mass_S1.iloc[i, 25] * height_zones_length_S1[i][5]
    ) / sum_of_heights
    mass_final_FB_S1.append(mass_final_FB_value)


#INNER COLUMNS
#Inner Columns: average value over total structure height
mass_final_IC_S1 = []  # Initialize an empty list

for i in range(len(mass_S1)):
    sum_of_heights = sum(height_zones_length_S1[i][0:6])
    mass_final_IC_value = (
        mass_S1.iloc[i, 26] * height_zones_length_S1[i][0]
        + mass_S1.iloc[i, 27] * height_zones_length_S1[i][1]
        + mass_S1.iloc[i, 28] * height_zones_length_S1[i][2]
        + mass_S1.iloc[i, 29] * height_zones_length_S1[i][3]
        + mass_S1.iloc[i, 30] * height_zones_length_S1[i][4]
        + mass_S1.iloc[i, 31] * height_zones_length_S1[i][5]
    ) / sum_of_heights
    mass_final_IC_S1.append(mass_final_IC_value)
```

In [ ]:
```
mass_S1['Mass Final OB'] = mass_final_OB_S1
mass_S1['Mass Final OC'] = mass_final_OC_S1
mass_S1['Mass Final FB'] = mass_final_FB_S1
mass_S1['Mass Final IC'] = mass_final_IC_S1
```

In [ ]:
```
#Material Split
mass_S1['Mass Total Steel'] = (mass_S1['Mass Final OB'] + mass_S1['Mass Final OC']
                               + mass_S1['Mass Final FB'] + mass_S1['Mass Final IC']
                               + mass_S1.iloc[:, 32])

mass_S1['Mass Total Floors'] = mass_S1.iloc[:, 2:8].sum(axis=1) / mass_S1.iloc[:, 33]
mass_S1['Mass Total'] = mass_S1['Mass Total Steel']
```

In [ ]:
```
#Material Split
mass_S1['Mass Total Steel'] = (mass_S1['Mass Final OB'] + mass_S1['Mass Final OC']
                               + mass_S1['Mass Final FB'] + mass_S1['Mass Final IC']
                               + mass_S1.iloc[:, 32])

mass_S1['Mass Total Floors'] = mass_S1.iloc[:, 2:8].sum(axis=1) / mass_S1.iloc[:, 33]
mass_S1['Mass Total'] = mass_S1['Mass Total Steel']
```

Outrigger System

In [ ]:
```
#OUTER BEAMS
#Outer beams: average value over total structure height
mass_final_OB_S2 = []

for i in range(len(mass_S2)):
    sum_of_heights = sum(height_zones_length_S2[i][0:6])
    mass_final_OB_value = (
        mass_S2.iloc[i, 8] * height_zones_length_S2[i][0]
        + mass_S2.iloc[i, 9] * height_zones_length_S2[i][1]
        + mass_S2.iloc[i, 10] * height_zones_length_S2[i][2]
        + mass_S2.iloc[i, 11] * height_zones_length_S2[i][3]
        + mass_S2.iloc[i, 12] * height_zones_length_S2[i][4]
        + mass_S2.iloc[i, 13] * height_zones_length_S2[i][5]
    ) / sum_of_heights
    mass_final_OB_S2.append(mass_final_OB_value)


#OUTER COLUMNS CONNECTED TO OUTRIGGER
#Outer columns connected to Outrigger: average value over total structure height
mass_final_OC_Out_S2 = []  # Initialize an empty list

for i in range(len(mass_S2)):
    sum_of_heights = sum(height_zones_length_S2[i][0:6])
    mass_final_OC_Out_value = (
        mass_S2.iloc[i, 14] * height_zones_length_S2[i][0]
        + mass_S2.iloc[i, 15] * height_zones_length_S2[i][1]
        + mass_S2.iloc[i, 16] * height_zones_length_S2[i][2]
        + mass_S2.iloc[i, 17] * height_zones_length_S2[i][3]
        + mass_S2.iloc[i, 18] * height_zones_length_S2[i][4]
        + mass_S2.iloc[i, 19] * height_zones_length_S2[i][5]
    ) / sum_of_heights
    mass_final_OC_Out_S2.append(mass_final_OC_Out_value)


#OUTER COLUMNS NOT CONNECTED TO OUTRIGGER
#Outer columns NOT connected to Outrigger: average value over total structure height
mass_final_OC_noOut_S2 = []  # Initialize an empty list

for i in range(len(mass_S2)):
    sum_of_heights = sum(height_zones_length_S2[i][0:6])
    mass_final_OC_noOut_value = (
        mass_S2.iloc[i, 20] * height_zones_length_S2[i][0]
        + mass_S2.iloc[i, 21] * height_zones_length_S2[i][1]
        + mass_S2.iloc[i, 22] * height_zones_length_S2[i][2]
        + mass_S2.iloc[i, 23] * height_zones_length_S2[i][3]
        + mass_S2.iloc[i, 24] * height_zones_length_S2[i][4]
        + mass_S2.iloc[i, 25] * height_zones_length_S2[i][5]
    ) / sum_of_heights
    mass_final_OC_noOut_S2.append(mass_final_OC_noOut_value)


#FLOOR BEAMS
#Floor Beams: average value over total structure height
mass_final_FB_S2 = []  # Initialize an empty list

for i in range(len(mass_S2)):
    sum_of_heights = sum(height_zones_length_S2[i][0:6])
    mass_final_FB_value = (
        mass_S2.iloc[i, 26] * height_zones_length_S2[i][0]
        + mass_S2.iloc[i, 27] * height_zones_length_S2[i][1]
        + mass_S2.iloc[i, 28] * height_zones_length_S2[i][2]
        + mass_S2.iloc[i, 29] * height_zones_length_S2[i][3]
        + mass_S2.iloc[i, 30] * height_zones_length_S2[i][4]
        + mass_S2.iloc[i, 31] * height_zones_length_S2[i][5]
    ) / sum_of_heights
    mass_final_FB_S2.append(mass_final_FB_value)


#INNER COLUMNS CONNECTED TO OUTRIGGER
#Inner Columns: average value over total structure height
mass_final_IC_Out_S2 = []  # Initialize an empty list

for i in range(len(mass_S2)):
    sum_of_heights = sum(height_zones_length_S2[i][0:6])
    mass_final_IC_Out_value = (
        mass_S2.iloc[i, 32] * height_zones_length_S2[i][0]
        + mass_S2.iloc[i, 33] * height_zones_length_S2[i][1]
        + mass_S2.iloc[i, 34] * height_zones_length_S2[i][2]
        + mass_S2.iloc[i, 35] * height_zones_length_S2[i][3]
        + mass_S2.iloc[i, 36] * height_zones_length_S2[i][4]
        + mass_S2.iloc[i, 37] * height_zones_length_S2[i][5]
    ) / sum_of_heights
    mass_final_IC_Out_S2.append(mass_final_IC_Out_value)


#INNER COLUMNS NOT CONNECTED TO OUTRIGGER
#Inner Columns: average value over total structure height
mass_final_IC_noOut_S2 = []  # Initialize an empty list

for i in range(len(mass_S2)):
    sum_of_heights = sum(height_zones_length_S2[i][0:6])
    mass_final_IC_noOut_value = (
        mass_S2.iloc[i, 38] * height_zones_length_S2[i][0]
        + mass_S2.iloc[i, 39] * height_zones_length_S2[i][1]
        + mass_S2.iloc[i, 40] * height_zones_length_S2[i][2]
        + mass_S2.iloc[i, 41] * height_zones_length_S2[i][3]
        + mass_S2.iloc[i, 42] * height_zones_length_S2[i][4]
        + mass_S2.iloc[i, 43] * height_zones_length_S2[i][5]
    ) / sum_of_heights
```

```
    mass_final_IC_noOut_S2.append(mass_final_IC_noOut_value)
```

In [ ]:
```
mass_S2['Mass Final OB'] = mass_final_OB_S2
mass_S2['Mass Final OC Out'] = mass_final_OC_Out_S2
mass_S2['Mass Final OC no Out'] = mass_final_OC_noOut_S2
mass_S2['Mass Final FB'] = mass_final_FB_S2
mass_S2['Mass Final IC Out'] = mass_final_IC_Out_S2
mass_S2['Mass Final IC no Out'] = mass_final_IC_noOut_S2
```

Material Split

In [ ]:
```
mass_S2['Mass Total Steel'] = (mass_S2['Mass Final OB'] + mass_S2['Mass Final OC Out'] + mass_S2['Mass Final OC no Out']
                               + mass_S2['Mass Final FB'] + mass_S2['Mass Final IC Out'] + mass_S2['Mass Final IC no Out']
                               + mass_S2['out:Average Total Mass Outrigger [kg/m2GFA]'])

mass_S2['Mass Total Floors'] = mass_S2.iloc[:, 2:8].sum(axis=1) / mass_S2.iloc[:, 47]
mass_S2['Mass Total Concrete Core'] = mass_S2.iloc[:, 45]
mass_S2['Mass Total'] = mass_S2['Mass Total Steel'] + mass_S2['Mass Total Concrete Core']
```

Average masses for both S1 and S2

In [ ]:
```
mean_mass_S1 = np.mean(mass_S1['Mass Total Steel'])
mean_mass_floors_S1 = np.mean(mass_S1['Mass Total Floors'])
min_mass_S1 = np.min(mass_S1['Mass Total Steel'])
p25_mass_S1 = np.percentile((mass_S1['Mass Total Steel']), 25)
p75_mass_S1 = np.percentile((mass_S1['Mass Total Steel']), 75)

print('Mean mass of steel of S1 is', round(mean_mass_S1,1), 'kg/GFA')
print('The 25th percentile of the mass of steel of S1 is', round(p25_mass_S1,1), 'kg/GFA')
print('The 75th percentile of the mass of steel of S1 is', round(p75_mass_S1,1), 'kg/GFA')

print("")

total_mass_S2 = (mass_S2['Mass Total Steel']) + (mass_S2['Mass Total Concrete Core'])
mean_total_mass_S2 = np.mean(total_mass_S2)
p25_total_mass_S2 = np.percentile(total_mass_S2, 25)
p75_total_mass_S2 = np.percentile(total_mass_S2, 75)

mean_mass_steel_S2 = np.mean(mass_S2['Mass Total Steel'])
mean_mass_concrete_S2 = np.mean(mass_S2['Mass Total Concrete Core'])

print('Mean mass of steel of S2 is', round(mean_mass_steel_S2,1), 'kg/GFA')
print('Mean mass of concrete of S2 is', round(mean_mass_concrete_S2,1), 'kg/GFA')

print('Mean total mass of S2 is', round(mean_total_mass_S2,1), 'kg/GFA')
print('The 25th percentile of the total mass of S2 is', round(p25_total_mass_S2,1), 'kg/GFA')
print('The 75th percentile of the total mass of S2 is', round(p75_total_mass_S2,1), 'kg/GFA')
```

In [ ]:
```
slenderness_S1 = mass_S1['in:Height [m]'] / mass_S1['in:Length [m]']
slenderness_S1_sorted = sorted(slenderness_S1, reverse=True)
print(slenderness_S1_sorted[0])

slenderness_S2 = mass_S2['in:Height [m]'] / mass_S2['in:Length [m]']
slenderness_S2_sorted = sorted(slenderness_S2, reverse=True)
print(slenderness_S2_sorted[0])
```

Performance Parameters Variables

i) Costs

In [ ]:
```
#Prefab Concrete Reinforced
pc_rc_price_area = 850                          # concrete price [€/m3]
density_pc_rc = 2500                            # density prefab reinforced concrete [kg/m3]
pc_rc_price = pc_rc_price_area / density_pc_rc  # concrete price [€/kg]

#Hollow Core
hc_price_area = 110                             # hollow core slab price (h=320mm) [€/m2]
density_hc = 429                                # hollow core slab mass per area (h=320mm) [kg/m2]
hc_price = hc_price_area / density_hc           # hollow core slab price (h=320mm) [€/kg]

#Steel
s_price = 3.5                                   # steel price S355 [€/kg]


print('Price: prefab reinforced concrete =', round(pc_rc_price,3), '€/kg')
print('Price: hollow core concrete =', round(hc_price,3), '€/kg')
print('Price: steel =', round(s_price,3), '€/kg')
```

i) Embodied Carbon

In [ ]:
```
CO2_sprice = 0.05         # shaduw price CO2 eq [€/kgCO2eq]

pc_rc_EmCO2 = 0.238       # EmCO2 precast reinforced [kgCO2eq/kg]
hc_EmCO2 = 0.155          # EmCO2 hollow core [kgCO2eq/kg]
s_EmCO2 = 1.12            # EmCO2 steel S355 [kgCO2eq/kg]

print('Embodied Carbon: prefab reinforced concrete =', round(pc_rc_EmCO2,3), 'kgCO2eq/kg')
print('Embodied Carbon: hollow core concrete =', round(hc_EmCO2,3), 'kgCO2eq/kg')
print('Embodied Carbon: steel =', round(s_EmCO2,3), 'kgCO2eq/kg')
```

2a) MODELLING PROCEDURE : BRACED FRAMED TUBE

SPLITTING AND SCALING DATA

In [ ]:
```
# Assuming df is your DataFrame and X, y are features and target variable
# Clean data and split into features (X) and target variable (y)

X = mass_S1[['in:Length [m]', 'in:Height [m]']]
y = mass_S1['Mass Total Steel']
X_initial = X
```

In [ ]:
```
X
```

Splitting data into training and test sets & scaling the sets

In [ ]:
```
# Split data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
X_train_df = pd.DataFrame(X_train, columns=['in:Length [m]', 'in:Height [m]'])
X_test_df = pd.DataFrame(X_test, columns=['in:Length [m]', 'in:Height [m]'])

scaler1 = MinMaxScaler()
X_scaled = scaler1.fit_transform(X)
X_train_scaled = scaler1.transform(X_train)
X_test_scaled = scaler1.transform(X_test_df)
```

In [ ]:
```
print("Shape of X_train:", X_train_scaled.shape)
```

k-fold cross-validation

In [ ]:
```
# Split data with k-fold cross-validation
kf = KFold(n_splits=5, shuffle=True, random_state=42)

for train_index, val_index in kf.split(X_train_df):
    X_train_fold, X_val_fold = X_train_df.iloc[train_index], X_train_df.iloc[val_index]
    y_train_fold, y_val_fold = y_train.iloc[train_index], y_train.iloc[val_index]

X_val_fold_scaled = scaler1.transform(X_val_fold)
X_train_fold_scaled = scaler1.transform(X_train_fold)
```

BUILDING ANN MODEL

In [ ]:
```
class ANNModel_S1(BaseEstimator):
    def __init__(self, neurons, kernel_initializer, bias_initializer, alpha=0.01, learning_rate_init=0.01):
        self.neurons = neurons
        self.alpha = alpha
        self.learning_rate_init = learning_rate_init
        self.kernel_initializer = kernel_initializer
        self.bias_initializer = bias_initializer

        self.model = MLPRegressor(
            hidden_layer_sizes=(neurons, neurons),
            activation='relu',
            solver='adam',
            max_iter=10000,
            n_iter_no_change=10,
            alpha=self.alpha,
            learning_rate_init=self.learning_rate_init,
            tol=1e-4,
            verbose=False,
            warm_start=False
        )

        self.losses = {'training_loss': [], 'validation_loss': []}

    def partial_fit(self, X, y):
        self.model.partial_fit(X, y)

    def fit(self, X, y, X_val=None, y_val=None):
        if X_val is not None and y_val is not None:
            self.model.validation_set = (X_val, y_val)
        else:
            self.model.validation_set = None

        for epoch in range(self.model.max_iter):
            self.model.partial_fit(X, y)
            training_loss = self.model.loss_
            self.losses['training_loss'].append(training_loss)

            if self.model.validation_set is not None:
                val_predictions = self.model.predict(X_val_fold_scaled)
                val_loss = np.mean((val_predictions - y_val) ** 2)  # Calculate MSE
                self.losses['validation_loss'].append(val_loss)

            if self.model.n_iter_no_change is not None and len(self.losses['training_loss']) > self.model.n_iter_no_change:
                recent_losses = self.losses['training_loss'][-self.model.n_iter_no_change:]
                if max(recent_losses) - min(recent_losses) < self.model.tol:
                    break

    def predict(self, X):
        predictions = self.model.predict(X)
        predictions[predictions < 0] = 0

        X_unscaled = scaler1.inverse_transform(X)
        slenderness = np.zeros_like(X_unscaled[:, 0])


        for i in range(len(slenderness)):
            slenderness[i] = X_unscaled[i, 1] / X_unscaled[i, 0]
            if slenderness[i] > 11.67:
                predictions[i] = np.nan

        return predictions
```

DEFINE FITNESS MODEL AND SET UP GA

In [ ]:
```
# Define the fitness function
def fitness_func(solution):
    neurons = int(solution[0])
    kernel_initializer_option = solution[1]
    bias_initializer_option = solution[2]
    alpha_option = solution[3]
    learning_rate_option = solution[4]

    alpha = 0.01 if alpha_option == 1 else 0.001
    learning_rate = 0.01 if learning_rate_option == 1 else 0.001
    kernel_initializer = 'random_uniform' if kernel_initializer_option == 0 else 'constant' if kernel_initializer_option == 1 else 'zeros'
    bias_initializer = 'random_uniform' if bias_initializer_option == 0 else 'constant' if bias_initializer_option == 1 else 'zeros'

    # Create and train the model
    model = ANNModel_S1(neurons, alpha=alpha, learning_rate_init=learning_rate,
                        kernel_initializer=kernel_initializer, bias_initializer=bias_initializer)

    model.fit(X_train_scaled, y_train, X_val_fold_scaled, y_val_fold)

    # Predict on the validation set
    y_pred_fold = model.predict(X_val_fold_scaled)

    # Calculate mean squared error
    mse = mean_squared_error(y_val_fold, y_pred_fold)

    # Print the neurons and mse value
    print("Neurons:", neurons, "Alpha:", alpha, "Learning Rate:", learning_rate,
          "Kernel Initializer:", kernel_initializer, "Bias Initializer", bias_initializer)
    print("MSE:", mse)

    return mse

# Define the parameter bounds for the genetic algorithm
varbound = np.array([[1, 50], [0, 2], [0, 2], [0, 1], [0, 1]])

# Define the algorithm parameters
algorithm_param = {
    'max_num_iteration': 30,
    'population_size': 100,
    'mutation_probability': 0.1,
    'parents_portion': 0.3,
    'elit_ratio': 0.1,
    'crossover_probability': 0.8,
    'crossover_type': 'uniform',
    'max_iteration_without_improv': 3
}

# Create the genetic algorithm
modelGA = ga(function=fitness_func, dimension=5, variable_type='int', variable_boundaries=varbound,
             algorithm_parameters=algorithm_param, function_timeout = 100)

# Run the genetic algorithm
modelGA.run()
```

Run ANN Model on the test data with the best parameters

In [ ]:
```
# Get the best solution found by the genetic algorithm
optimum_solution = modelGA.output_dict['variable']
best_fitness = modelGA.output_dict['function']

# Extract the values for number of neurons, alpha, and learning rate from the solution
optimum_neurons = int(optimum_solution[0])
optimum_alpha = 0.01 if optimum_solution[3] == 1 else 0.001
optimum_learning_rate = 0.01 if optimum_solution[4] == 1 else 0.001
optimum_kernel_initializer = 'random_uniform' if optimum_solution[1] == 0 else 'constant' if optimum_solution[1] == 1 else 'zeros'
optimum_bias_initializer = 'random_uniform' if optimum_solution[2] == 0 else 'constant' if optimum_solution[2] == 1 else 'zeros'

print("Best solution (number of neurons):", optimum_neurons)
print("Best alpha option:", optimum_alpha)
print("Best learning rate option:", optimum_learning_rate)
print("Best kernel initializer", optimum_kernel_initializer)
print("Best bias initializer", optimum_bias_initializer)
print("Corresponding fitness (MSE):", best_fitness)

# Create a new instance of the ANN model with the best parameters
best_model_S1 = ANNModel_S1(neurons=optimum_neurons, kernel_initializer=optimum_kernel_initializer,
                        bias_initializer=optimum_bias_initializer,alpha=optimum_alpha,
                        learning_rate_init=optimum_learning_rate)
```

Calculate the Mean Squared Error for the validation fold with the best parameters

In [ ]:
```
#Neurons: 44 Alpha: 0.001 Learning Rate: 0.01 Kernel Initializer: zeros Bias Initializer zeros
#MSE: 542.8312998703304

best_model_S1 = ANNModel_S1(neurons=44, kernel_initializer='constant',
                          bias_initializer='random_uniform',alpha=0.01,
                          learning_rate_init=0.01)

# Train the model on the current fold
best_model_S1.fit(X_train_scaled, y_train, X_val_fold_scaled, y_val_fold)

# Predict on the test set for this fold
y_pred_fold = best_model_S1.predict(X_val_fold_scaled)

# Calculate mean squared error for this fold
mse_fold = mean_squared_error(y_val_fold, y_pred_fold)

print("Mean Squared Error for this fold:", mse_fold)
```

In [ ]:
```
y_val_fold
df = pd.DataFrame(y_pred_fold)
```

Check the shapes of the input data sets

In [ ]:
```
X_train_fold.reset_index(drop=True, inplace=True)
y_train_fold.reset_index(drop=True, inplace=True)
```

In [ ]:
```
print("Shape of X_train_fold:", X_train_fold.shape)
print("Shape of y_train_fold:", y_train_fold.shape)
print("Shape of y_train:", y_train.shape)
print("Shape of X_train:", X_train_scaled.shape)
#print("Index values of X_train_fold:", X_train_fold.index)
#print("Index values of y_train_fold:", y_train_fold.index)
print("Shape of y_val:", y_val_fold.shape)
print("Shape of X_val_scaled:", X_val_fold_scaled.shape)
```

Calculate MSE

In [ ]:
```
# Calculate mean squared error for this fold
y_pred_val = best_model_S1.predict(X_val_fold_scaled)
mse_valfold = mean_squared_error(y_val_fold, y_pred_val)
print(mse_valfold)
```

In [ ]:
```
# Calculate mean squared error for the test data
y_pred_test = best_model_S1.predict(X_test_scaled)
mse_test = mean_squared_error(y_test, y_pred_test)
print(mse_test)
```

COMPARING MODEL WITH TEST & TRAIN DATA

Actual test data and predicted test data based on the scaled input data (X test)

In [ ]:
```
y_test_array = np.array(y_test)
y_pred_test = best_model_S1.predict(X_test_scaled)
```

Difference between the predicted test data and the actual test data

In [ ]:
```
diff_test = y_test_array - y_pred_test
```

Difference in percentage between the predicted test data and the actual data

In [ ]:
```
diff_percentage = (diff_test / y_test) * 100
diff_percentage = np.array(diff_percentage)
```

Illustrating the distribution of the absolute percentages between actual and predicted values (test data)

In [ ]:
```
# Create bins for the histogram
bins = np.arange(-101, 101, 2)  # Adjust the range and width of bins as needed

# Create the histogram
plt.hist(diff_percentage, bins=bins, color=color_lightgreen, density=True)

# Calculate the mean and standard deviation of your data
mu = np.mean(diff_percentage)
std = np.std(diff_percentage)

# Generate the x values for your normal distribution
xmin, xmax = plt.xlim()
x = np.linspace(xmin, xmax, 100)

# Generate the y values (probability density function) for your normal distribution
p = norm.pdf(x, mu, std)

# Plot the normal distribution
plt.plot(x, p, 'k', linewidth=2, color=color_green)

# Add labels and title
plt.xlabel('Percentage Difference between Actual and Predicted Values')
plt.ylabel('Probability Density')
plt.title('Distribution of Percentage Differences of Braced Framed Tube (S1)')

# Show the plot
plt.show()

print("The mean is equal to", round(mu,3), '%')
```

2b) MODELLING PROCEDURE: S2: Outrigger

SPLITTING AND SCALING DATA

In [ ]:
```
# Assuming df is your DataFrame and X, y are features and target variable
# Clean data and split into features (X) and target variable (y)

X2 = mass_S2[['in:Length [m]', 'in:Height [m]']]
y2 = mass_S2[['Mass Total Steel', 'Mass Total Concrete Core']]
```

In [ ]:
```
len(y2)
```

In [ ]:
```
# Split data into training and test sets
X_train2, X_test2, y_train2, y_test2 = train_test_split(X2, y2, test_size=0.2, random_state=42)
X_train_df2 = pd.DataFrame(X_train2, columns=['in:Length [m]', 'in:Height [m]'])
X_test_df2 = pd.DataFrame(X_test2, columns=['in:Length [m]', 'in:Height [m]'])

scaler2 = MinMaxScaler()
X_scaled2 = scaler2.fit_transform(X2)
X_train_scaled2 = scaler2.transform(X_train2)
X_test_scaled2 = scaler2.transform(X_test_df2)
```

In [ ]:
```
# Split data with k-fold cross-validation
kf = KFold(n_splits=5, shuffle=True, random_state=42)

for train_index, val_index in kf.split(X_train_df2):
    X_train_fold2, X_val_fold2 = X_train_df2.iloc[train_index], X_train_df2.iloc[val_index]
    y_train_fold2, y_val_fold2 = y_train2.iloc[train_index], y_train2.iloc[val_index]

X_val_fold_scaled2 = scaler2.transform(X_val_fold2)
X_train_fold_scaled2 = scaler2.transform(X_train_fold2)
```

Check the shape of the input data

In [ ]:
```
print("Shape of X_train_fold:", X_train_fold2.shape)
print("Shape of y_train_fold:", y_train_fold2.shape)
print("Shape of y_train:", y_train2.shape)
print("Shape of X_train:", X_train_scaled2.shape)
#print("Index values of X_train_fold:", X_train_fold.index)
#print("Index values of y_train_fold:", y_train_fold.index)
print("Shape of y_val:", y_val_fold2.shape)
print("Shape of X_val_scaled:", X_val_fold_scaled2.shape)
```

BUILDING ANN MODEL

k-fold cross-validation

In [ ]:
```
class ANNModel_S2(BaseEstimator):
    def __init__(self, neurons, kernel_initializer, bias_initializer, alpha=0.01, learning_rate_init=0.01):
        self.neurons = neurons
        self.alpha = alpha
        self.learning_rate_init = learning_rate_init
        self.kernel_initializer = kernel_initializer
        self.bias_initializer = bias_initializer

        self.model = MLPRegressor(
            hidden_layer_sizes=(neurons, neurons),
            activation='relu',
            solver='adam',
            max_iter=10000,
            n_iter_no_change=10,
            alpha=self.alpha,
            learning_rate_init=self.learning_rate_init,
            tol=1e-4,
            verbose=False,
            warm_start=False
        )

        self.losses = {'training_loss': [], 'validation_loss': []}

    def partial_fit(self, X, y):
        self.model.partial_fit(X, y)

    def fit(self, X, y, X_val=None, y_val=None):
        if X_val is not None and y_val is not None:
            self.model.validation_set = (X_val, y_val)
        else:
            self.model.validation_set = None

        for epoch in range(self.model.max_iter):
            self.model.partial_fit(X, y)
            training_loss = self.model.loss_
            self.losses['training_loss'].append(training_loss)

            if self.model.validation_set is not None:
                val_predictions = self.model.predict(X_val)
                val_loss = np.mean((val_predictions - y_val) ** 2, axis=0)
                self.losses['validation_loss'].append(np.mean(val_loss))

            if self.model.n_iter_no_change is not None and len(self.losses['training_loss']) > self.model.n_iter_no_change:
                recent_losses = self.losses['training_loss'][-self.model.n_iter_no_change:]
                if max(recent_losses) - min(recent_losses) < self.model.tol:
                    break

    def predict(self, X):
        predictions = self.model.predict(X)
        predictions[predictions < 0] = 0

        X_unscaled = scaler2.inverse_transform(X)
        slenderness = np.zeros_like(X_unscaled[:, 0])


        for i in range(len(slenderness)):
            slenderness[i] = X_unscaled[i, 1] / X_unscaled[i, 0]
            if slenderness[i] > 12.6:
                predictions[i] = np.nan

        return predictions
```

DEFINE THE FITNESS FUNCTION AND SET UP GA

In [ ]:
```
def fitness_func(solution):
    neurons = int(solution[0])
    alpha_option = solution[3]
    learning_rate_option = solution[4]
    kernel_initializer_option = solution[1]
    bias_initializer_option = solution[2]

    alpha = 0.01 if alpha_option == 1 else 0.001
    learning_rate = 0.01 if learning_rate_option == 1 else 0.001
    kernel_initializer = 'random_uniform' if kernel_initializer_option == 0 else 'constant' if kernel_initializer_option == 1 else 'zeros'
    bias_initializer = 'random_uniform' if bias_initializer_option == 0 else 'constant' if bias_initializer_option == 1 else 'zeros'

    # Create and train the model
    model = ANNModel_S2(neurons, kernel_initializer=kernel_initializer, bias_initializer=bias_initializer,
                        alpha=alpha, learning_rate_init=learning_rate)

    model.fit(X_train_scaled2, y_train2, X_val_fold_scaled2, y_val_fold2)

    # Predict on the validation set
    y_pred_fold2 = model.predict(X_val_fold_scaled2)

    # Calculate mean squared error
    mse = mean_squared_error(y_val_fold2, y_pred_fold2)


    # Print the neurons and mse value
    print("Neurons:", neurons, "Alpha:", alpha, "Learning Rate:", learning_rate,
          "Kernel Initializer:", kernel_initializer, "Bias Initializer", bias_initializer)
    print("MSE:", mse)

    return mse

# Define the parameter bounds for the genetic algorithm
varbound = np.array([[1, 50], [0, 1], [0, 1], [0, 2], [0, 2]])

# Define the algorithm parameters
algorithm_param = {
    'max_num_iteration': 30,
    'population_size': 100,
    'mutation_probability': 0.1,
    'parents_portion': 0.3,
    'elit_ratio': 0.1,
    'crossover_probability': 0.8,
    'crossover_type': 'uniform',
    'max_iteration_without_improv': 3
}

# Create the genetic algorithm
modelGA = ga(function=fitness_func, dimension=5, variable_type='int', variable_boundaries=varbound,
             algorithm_parameters=algorithm_param, function_timeout = 20000)

# Run the genetic algorithm
modelGA.run()
```

Run ANN model on the test data with the best parameters

In [ ]:
```
#Neurons: 36 Alpha: 0.01 Learning Rate: 0.01 Kernel Initializer: constant Bias Initializer random_uniform
#MSE: 717.8287529135639

# Get the best solution found by the genetic algorithm
optimum_solution = modelGA.output_dict['variable']
best_fitness = modelGA.output_dict['function']

# Extract the values for number of neurons, alpha, and learning rate from the solution
optimum_neurons = int(optimum_solution[0])
optimum_alpha = 0.01 if optimum_solution[3] == 1 else 0.001
optimum_learning_rate = 0.01 if optimum_solution[4] == 1 else 0.001
optimum_kernel_initializer = 'random_uniform' if optimum_solution[1] == 0 else 'constant' if optimum_solution[1] == 1 else 'zeros'
optimum_bias_initializer = 'random_uniform' if optimum_solution[2] == 0 else 'constant' if optimum_solution[2] == 1 else 'zeros'

print("Best solution (number of neurons):", optimum_neurons)
print("Best alpha option:", optimum_alpha)
print("Best learning rate option:", optimum_learning_rate)
print("Best kernel initializer", optimum_kernel_initializer)
print("Best bias initializer", optimum_bias_initializer)
print("Corresponding fitness (MSE):", best_fitness)

# Create a new instance of the ANN model with the best parameters
best_model = ANNModel_S2(neurons=optimum_neurons, kernel_initializer=optimum_kernel_initializer,
                         bias_initializer=optimum_bias_initializer, alpha=optimum_alpha,
                         learning_rate_init=optimum_learning_rate)
```

Calculate the Mean Squared Error for the validation fold with the best parameters

In [ ]:
```
best_model_S2 = ANNModel_S2(neurons=36, kernel_initializer='constant',
                            bias_initializer='random_uniform', alpha=0.01,
                            learning_rate_init=0.01)

# Train the model on the current fold
best_model_S2.fit(X_train_scaled2, y_train2, X_val_fold_scaled2, y_val_fold2)

# Predict on the test set for this fold
y_pred_fold2 = best_model_S2.predict(X_val_fold_scaled2)

# Calculate mean squared error for this fold
mse_fold2 = mean_squared_error(y_val_fold2, y_pred_fold2)

print("Mean Squared Error for this fold:", mse_fold2)
```

Calculate the MSE

In [ ]:
```
# Calculate mean squared error for this fold
y_pred_val2 = best_model_S2.predict(X_val_fold_scaled2)
mse_valfold = mean_squared_error(y_val_fold2, y_pred_val2)
print(mse_valfold)
```

In [ ]:
```
# Calculate mean squared error for the test data
y_pred_test2 = best_model_S2.predict(X_test_scaled2)
mse_test = mean_squared_error(y_test2, y_pred_test2)
print(mse_test)
```

COMPARING MODEL WITH TEST & TRAIN DATA

Actual test data and predicted test data based on the scaled input data (X test)

In [ ]:
```
y_test_array2 = np.array(y_test2)
y_pred_test2 = best_model_S2.predict(X_test_scaled2)
```

Difference between the predicted test data and the actual test data

In [ ]:
```
diff_test2 = y_test_array2 - y_pred_test2
```

Difference in percentage between the predicted test data and the actual data

In [ ]:
```
diff_percentage2 = (diff_test2 / y_test2) * 100
diff_percentage2 = np.array(diff_percentage2)
diff_percentage2 = pd.DataFrame(diff_percentage2)
```

Illustrating the distribution of the absolute percentages between actual and predicted values (test data)

In [ ]:
```
# Create bins for the histogram
bins = np.arange(-101, 101, 2)  # Adjust the range and width of bins as needed

diff_percentage2 = diff_percentage2.values.ravel()

# Create the histogram
plt.hist(diff_percentage2, bins=bins, color=color_lightblue, density=True)

# Calculate the mean and standard deviation of your data
mu = np.mean(diff_percentage2)
std = np.std(diff_percentage2)

# Generate the x values for your normal distribution
xmin, xmax = plt.xlim()
x = np.linspace(xmin, xmax, 100)

# Generate the y values (probability density function) for your normal distribution
p = norm.pdf(x, mu, std)

# Plot the normal distribution
plt.plot(x, p, 'k', linewidth=2, color=color_darkblue)

# Add labels and title
plt.xlabel('Percentage Difference between Actual and Predicted Values')
plt.ylabel('Probability Density')
plt.title('Distribution of Percentage Differences of Outrigger System (S2)')

# Show the plot
plt.show()

print("The mean is equal to", round(mu,3), '%')
```

3) MAKE PREDICTIONS

A) PREDICTIONS REGARDING TOTAL COSTS AND EMBODIED CARBON

Interactively making predictions

In [ ]:
```
scaler = MinMaxScaler()
```

In [ ]:
```
def make_predictions_S1(length, height, scaler):
    # Prepare input data
    input_data = pd.DataFrame({
    'in:Length [m]': [length],
    'in:Height [m]': [height]
    })

    input_data1 = scaler1.transform(input_data)
    input_data2 = scaler2.transform(input_data)

    # Convert the predictons from mass to costs and embodied carbon
    # Costs
    predictions_S1 = best_model_S1.predict(input_data1)
    predictions_S1 *= s_price
    p_total_costs_S1 = predictions_S1

    predictions_S2 = best_model_S2.predict(input_data2)
    predictions_S2[:, 0] *= s_price
    predictions_S2[:, 1] *= pc_rc_price
    p_total_costs_S2 = predictions_S2.sum(axis=1)

    predictions_S1 = best_model_S1.predict(input_data1)
    predictions_S1 *= s_EmCO2*CO2_sprice
    p_total_EmCO2_S1 = predictions_S1

    predictions_S2 = best_model_S2.predict(input_data2)
    predictions_S2[:, 0] *= s_EmCO2*CO2_sprice
    predictions_S2[:, 1] *= pc_rc_EmCO2*CO2_sprice
    p_total_EmCO2_S2 = predictions_S2.sum(axis=1)

    rounded_total_costs_S1 = round(float(p_total_costs_S1), 2)
    rounded_total_EmCO2_S1 = round(float(p_total_EmCO2_S1), 2)

    rounded_total_costs_S2 = round(float(p_total_costs_S2), 2)
    rounded_total_EmCO2_S2 = round(float(p_total_EmCO2_S2), 2)

    print('The predicted total costs of the Braced Framed Tube is', rounded_total_costs_S1, '€/GFA')
    print('The predicted embodied carbon of the Braced Framed Tube is', rounded_total_EmCO2_S1, '€/GFA')
    print('')
    print('The predicted total costs of the Outrigger System is', rounded_total_costs_S2, '€/GFA')
    print('The predicted embodied carbon of the Outrigger System is', rounded_total_EmCO2_S2, '€/GFA')
    print('')

    if rounded_total_costs_S1 < rounded_total_costs_S2:
        print('The structural form with the lowest construction costs is Braced Framed Tube')
    else:
        print('The structural form with the lowest construction costs is Outrigger System')

    if rounded_total_EmCO2_S1 < rounded_total_EmCO2_S2:
        print('The structural form with the lowest embodied carbon costs is Braced Framed Tube')
    else:
        print('The structural form with the lowest embodied carbon costs is Outrigger System')

    return p_total_costs_S1, p_total_EmCO2_S1, p_total_costs_S2, p_total_EmCO2_S2

# Create interactive sliders for floorplan length and height
floorplan_length_slider = widgets.FloatSlider(value=20, min=5,
                                              max=75, step=1,
                                              description='Width:')

floorplan_length_slider.layout.width = '40%'
floorplan_length_slider.style = {'description_width': '50%'}

height_slider = widgets.FloatSlider(value=100, min=50,
                                    max=500, step=1,
                                    description='Height:')

height_slider.layout.width = '40%'
height_slider.style = {'description_width': '50%'}

def make_predictions_wrapper(length, height):
    return make_predictions_S1(length, height, scaler=MinMaxScaler())

output = widgets.interactive_output(make_predictions_wrapper,
                                    {'length': floorplan_length_slider,
                                     'height': height_slider})
# Display the sliders and output
display(floorplan_length_slider, height_slider, output)
```

In [ ]:
```
# Define constant variables
constant_options = ['Width', 'Height']

# Create interactive sliders and dropdown for constant variables
constant_variable_dropdown = widgets.Dropdown(options=constant_options, value='Width', description='Constant Parameter:')
constant_value_input = widgets.FloatText(value=20, description='Constant Parameter:')
variable_slider = widgets.FloatSlider(value=20, min=5, max=500, step=1, description='Variable Parameter:')

# Update descriptions based on the constant variable
def update_slider_descriptions(constant_variable):
    if constant_variable == 'Height':
        constant_value_input.description = 'Constant Parameter = Height'
        variable_slider.description = 'Variable Parameter = Width'
        variable_slider.min = 0
        variable_slider.max = 100
    else:
        constant_value_input.description = 'Constant Parameter = Width'
        variable_slider.description = 'Variable Parameter = Height'
        variable_slider.min = 50
        variable_slider.max = 300

# Set width for the sliders and dropdown
constant_variable_dropdown.layout.width = '40%'
constant_value_input.layout.width = '40%'
variable_slider.layout.width = '40%'
constant_variable_dropdown.style = {'description_width': '50%'}
constant_value_input.style = {'description_width': '50%'}
variable_slider.style = {'description_width': '50%'}

# Observe changes in the dropdown and update slider descriptions accordingly
widgets.interactive(update_slider_descriptions, constant_variable=constant_variable_dropdown)

# Function to make predictions and plot the graph
def make_predictions_and_plot_S1(constant_variable, constant_value, variable_range):
    # Generate a range of variable values within the specified range
    variable_values = np.linspace(variable_slider.min, variable_slider.max, 100).reshape(-1, 1)

    # Prepare input data
    if constant_variable == 'Width':
        input_data = pd.DataFrame({
            'in:Length [m]': constant_value,
            'in:Height [m]': variable_values.flatten()
        })

    else:
        input_data = pd.DataFrame({
            'in:Length [m]': variable_values.flatten(),
            'in:Height [m]': constant_value
        })

    input_data1 = scaler1.transform(input_data)
    input_data2 = scaler2.transform(input_data)

    # Convert the predictons from mass to costs and embodied carbon
    # Costs
    predictions_S1 = best_model_S1.predict(input_data1)
    predictions_S1 *= s_price
    p_total_costs_S1 = predictions_S1

    predictions_S1 = best_model_S1.predict(input_data1)
    predictions_S1 *= s_EmCO2*CO2_sprice
    p_total_EmCO2_S1 = predictions_S1

    predictions_S2 = best_model_S2.predict(input_data2)
    predictions_S2[:, 0] *= s_price
    predictions_S2[:, 1] *= pc_rc_price
    p_total_costs_S2 = predictions_S2.sum(axis=1)

    predictions_S2 = best_model_S2.predict(input_data2)
    predictions_S2[:, 0] *= s_EmCO2*CO2_sprice
    predictions_S2[:, 1] *= pc_rc_EmCO2*CO2_sprice
    p_total_EmCO2_S2 = predictions_S2.sum(axis=1)


    if constant_variable == 'Width':
        constant_value_df = pd.DataFrame({'in:Length [m]': [constant_value]})
        constant_value_df.columns = ['in:Length [m]']
        variable_range_df = pd.DataFrame({'in:Height [m]': [variable_range]})
        variable_range_df.columns = ['in:Height [m]']

        input_data_dot = pd.DataFrame({
            'in:Length [m]': constant_value_df['in:Length [m]'],
            'in:Height [m]': variable_range_df['in:Height [m]']
       })

    else:
        constant_value_df = pd.DataFrame({'in:Height [m]': [constant_value]})
        constant_value_df.columns = ['in:Height [m]']
        variable_range_df = pd.DataFrame({'in:Length [m]': [variable_range]})
        variable_range_df.columns = ['in:Length [m]']

        input_data_dot = pd.DataFrame({
            'in:Length [m]': variable_range_df['in:Length [m]'],
            'in:Height [m]': constant_value_df['in:Height [m]']
        })


    # Extract the relevant columns for plotting
    if constant_variable == 'Width':
        X_filtered = input_data['in:Height [m]']
    else:
        X_filtered = input_data['in:Length [m]']
```

```
    input_data_dot1 = scaler1.transform(input_data_dot)
    input_data_dot2 = scaler2.transform(input_data_dot)

    predictions_dot_S1 = best_model_S1.predict(input_data_dot1)
    predictions_dot_S1 *= s_price
    p_total_costs_S1_dot = predictions_dot_S1

    predictions_dot_S1 = best_model_S1.predict(input_data_dot1)
    predictions_dot_S1 *= s_EmCO2*CO2_sprice
    p_total_EmCO2_S1_dot = predictions_dot_S1

    predictions_dot_S2 = best_model_S2.predict(input_data_dot2)
    predictions_dot_S2[:, 0] *= s_price
    predictions_dot_S2[:, 1] *= pc_rc_price
    p_total_costs_S2_dot = predictions_dot_S2.sum(axis=1)

    predictions_dot_S2 = best_model_S2.predict(input_data_dot2)
    predictions_dot_S2[:, 0] *= s_EmCO2*CO2_sprice
    predictions_dot_S2[:, 1] *= pc_rc_EmCO2*CO2_sprice
    p_total_EmCO2_S2_dot = predictions_dot_S2.sum(axis=1)

    # Create subplots
    plt.figure(figsize=(15, 5))


    # Plot for Total Costs per GFA
    plt.subplot(1, 2, 1)

    #plt.xlim(0, max(X_filtered) + 20)
    #plt.ylim(0, max(p_total_costs_S1) + 50)

    plt.plot(X_filtered, p_total_costs_S1, color=color_green, label=f'S1: Braced Framed Tube')
    plt.plot(X_filtered, p_total_costs_S2, color=color_darkblue, label=f'S2: Outrigger System')

    plt.scatter([variable_range], np.round(p_total_costs_S1_dot, 2), color=color_lightgreen, marker='o',
                label=f'S1: Predicted Costs for chosen value = {np.round(p_total_costs_S1_dot[0], 2)}')

    plt.scatter([variable_range], np.round(p_total_costs_S2_dot, 2), color=color_lightblue, marker='o',
                label=f'S2: Predicted Costs for chosen value = {np.round(p_total_costs_S2_dot[0], 2)}')

    plt.title(f'Total Costs [€/GFA] for {constant_variable}={constant_value}')

    if constant_variable == 'Width':
        plt.xlabel('Height [m]')

    else:
        plt.xlabel('Width [m]')

    plt.ylabel('Total Costs [€/GFA]')
    plt.legend(loc='upper left')

    # Plot for Total Embodied Carbon per GFA
    plt.subplot(1, 2, 2)

    #plt.xlim(0, max(X_filtered) + 20)
    #plt.ylim(0, max(p_total_EmCO2_S1) + 10)

    plt.plot(X_filtered, p_total_EmCO2_S1, color=color_green, label=f'S1: Braced Framed Tube')
    plt.plot(X_filtered, p_total_EmCO2_S2, color=color_darkblue, label=f'S2: Outrigger System')

    plt.scatter([variable_range], np.round(p_total_EmCO2_S1_dot, 2), color=color_lightgreen, marker='o',
                label=f'S1: Predicted EmCO2 for chosen value = {np.round(p_total_EmCO2_S1_dot[0], 2)}')

    plt.scatter([variable_range], np.round(p_total_EmCO2_S2_dot, 2), color=color_lightblue, marker='o',
                label=f'S2: Predicted EmCO2 for chosen value = {np.round(p_total_EmCO2_S2_dot[0], 2)}')

    plt.title(f'Total Embodied Carbon [€/GFA] for {constant_variable}={constant_value}')

    if constant_variable == 'Width':
        plt.xlabel('Height [m]')

    else:
        plt.xlabel('Width [m]')

    plt.ylabel('Total Embodied Carbon [€/GFA]')
    plt.legend(loc='upper left')

    # Adjust layout
    plt.tight_layout()
    plt.show()

# Create an interactive output widget
output = widgets.interactive_output(make_predictions_and_plot_S1, {
    'constant_variable': constant_variable_dropdown,
    'constant_value': constant_value_input,
    'variable_range': variable_slider
})


# Create an interactive layout with sliders and output
interactive_layout = widgets.VBox([
    constant_variable_dropdown,
    constant_value_input,
    variable_slider,
    output
])

# Display the interactive layout
display(interactive_layout)
```

4a) ANALYSIS OF MODEL RESULT : BRACED FRAMED TUBE

In [ ]:
```
from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import PolynomialFeatures
from sklearn.metrics import mean_squared_error
```

COMPARISON INTERPOLATION TECHNIQUES

A) Analysis of ANN

Determination of ANN in terms of MSE

In [ ]:
```
# Predictions on the test set
predictions_test_S1 = best_model_S1.predict(X_test_scaled)

# Calculate costs and EmCO2 separately for test set
y_pred_test_costs = predictions_test_S1 * s_price
y_pred_test_EmCo2 = predictions_test_S1 * s_EmCO2 * CO2_sprice

# Predictions on the training set
predictions_train_S1 = best_model_S1.predict(X_train_scaled)

# Calculate costs and EmCO2 separately for training set
y_pred_train_costs = predictions_train_S1 * s_price
y_pred_train_EmCo2 = predictions_train_S1 * s_EmCO2 * CO2_sprice


# TRAINING DATA
# Converting training data for costs
y_train_costs = y_train.copy()
y_train_costs *= s_price

# Converting training data for EmCO2
y_train_EmCO2 = y_train.copy()
y_train_EmCO2 *= s_EmCO2 * CO2_sprice

# Calculation of MSE and RMSE
mse_ann_train_EmCo2 = mean_squared_error(y_train_EmCO2, y_pred_train_EmCo2)
mse_ann_train_costs = mean_squared_error(y_train_costs, y_pred_train_costs)

rmse_ann_train_EmCo2 = np.sqrt(mse_ann_train_EmCo2)
rmse_ann_train_costs = np.sqrt(mse_ann_train_costs)

print("S1: Braced Framed Tube - EmCO2: ANN Model (Training Set), the MSE:", round(mse_ann_train_EmCo2, 3), "and RMSE:", round(rmse_ann_tra
print("S1: Braced Framed Tube - Costs: ANN Model (Training Set), the MSE:", round(mse_ann_train_costs, 3), "and RMSE:", round(rmse_ann_tra
print("")


# TEST DATA
# Converting training data for costs
y_test_costs = y_test.copy()
y_test_costs *= s_price

# Converting training data for EmCO2
y_test_EmCO2 = y_test.copy()
y_test_EmCO2 *= s_EmCO2 * CO2_sprice

# Calculation of MSE and RMSE
mse_ann_test_EmCo2 = mean_squared_error(y_test_EmCO2, y_pred_test_EmCo2)
mse_ann_test_costs = mean_squared_error(y_test_costs, y_pred_test_costs)

rmse_ann_test_EmCo2_S1 = np.sqrt(mse_ann_test_EmCo2)
rmse_ann_test_costs_S1 = np.sqrt(mse_ann_test_costs)

print("S1: Braced Framed Tube - EmCO2: ANN Model (Test Set), the MSE:", round(mse_ann_test_EmCo2, 3), "and RMSE:", round(rmse_ann_test_EmC
print("S1: Braced Framed Tube - Costs: ANN Model (Test Set), the MSE:", round(mse_ann_test_costs, 3), "and RMSE:", round(rmse_ann_test_cos
```

Plot of actual and predicted values based on ANN

In [ ]:
```
# Create subplots
plt.figure(figsize=(12, 6))

# Plot for Costs
plt.subplot(1, 2, 1)

y_train_array = np.array(y_train_costs)
y_test_array = np.array(y_test_costs)
y_pred_test_costs_array = np.array(y_pred_test_costs)
y_pred_train_costs_array = np.array(y_pred_train_costs)

plt.scatter(y_test_array, y_pred_test_costs_array, color=color_purple, label='Test data')
plt.scatter(y_train_array, y_pred_train_costs_array, color=color_darkblue, label='Train data')

max_value_costs = max(y_test_array.max(), y_train_array.max(), y_pred_test_costs_array.max(), y_pred_train_costs_array.max())

plt.plot([0, max_value_costs], [0, max_value_costs], color=color_orange, linestyle='--', linewidth=2, label='True value')
plt.title('S1: Braced Framed Tube - ANN: Total Costs')
plt.xlabel('Actual Total Costs [€/GFA]')
plt.ylabel('Predicted Total Costs [€/GFA]')
plt.legend()

# Plot for Embodied Carbon
plt.subplot(1, 2, 2)

y_train_array = np.array(y_train_EmCO2)
y_test_array = np.array(y_test_EmCO2)
y_pred_test_EmCo2_array = np.array(y_pred_test_EmCo2)
y_pred_train_EmCo2_array = np.array(y_pred_train_EmCo2)

plt.scatter(y_test_array, y_pred_test_EmCo2_array, color=color_purple, label='Test data')
plt.scatter(y_train_array, y_pred_train_EmCo2_array, color=color_darkblue, label='Train data')

max_value_EmCo2 = max(y_test_array.max(), y_train_array.max(), y_pred_test_EmCo2_array.max(), y_pred_train_EmCo2_array.max())

plt.plot([0, max_value_EmCo2], [0, max_value_EmCo2], color=color_orange, linestyle='--', linewidth=2, label='True value')
plt.title('S1: Braced Framed Tube - ANN: Total Embodied Carbon')
plt.xlabel('Actual Total Embodied Carbon [€/GFA]')
plt.ylabel('Predicted Total Embodied Carbon [€/GFA]')
plt.legend()

# Adjust layout
plt.tight_layout()
plt.show()
```

Plot of convergence of ANN

In [ ]:
```
fig, axes = plt.subplots(1, 2, figsize=(12, 5))  # Create a figure with 1 row and 2 columns

# Plot the original plot on the first subplot
axes[0].plot(best_model_S1.losses['training_loss'], label='Training Loss', color = color_darkblue)
axes[0].plot(best_model_S1.losses['validation_loss'], label='Validation Loss', color = color_orange)
axes[0].set_title('Convergence Plot for Braced Framed Tube (S1)')
axes[0].set_xlabel('Epochs')
axes[0].set_ylabel('Loss [MSE]')
axes[0].legend()

ymax = np.max(best_model_S1.losses['validation_loss'])
axes[0].set_ylim(bottom=-1000, top=ymax+5000)



# Plot the zoomed-in plot on the second subplot
axes[1].plot(best_model_S1.losses['training_loss'], label='Training Loss', color = color_darkblue)
axes[1].plot(best_model_S1.losses['validation_loss'], label='Validation Loss', color = color_orange)
axes[1].set_title('Zoomed-in Convergence Plot for Braced Framed Tube (S1)')
axes[1].set_xlabel('Epochs')
axes[1].set_ylabel('Loss [MSE]')
axes[1].legend()

# Set limits for better visualization on the zoomed-in plot
axes[1].set_xlim(left=0-20, right=600)
axes[1].set_ylim(bottom=-1000, top=ymax+5000)

plt.tight_layout()
plt.show()
```

B) Analysis of Linear Regression

Determination of ANN in terms of MSE

In [ ]:
```
linear_model = LinearRegression()

linear_model.fit(X_train_scaled, y_train)

# Predictions on the test set
predictions_test_S1 = linear_model.predict(X_test_scaled)
predictions_test_S1[predictions_test_S1 < 0] = 0

# Calculate costs and EmCO2 separately for test set
y_pred_test_costs = predictions_test_S1 * s_price
y_pred_test_EmCo2 = predictions_test_S1 * s_EmCO2 * CO2_sprice

# Predictions on the training set
predictions_train_S1 = linear_model.predict(X_train_scaled)
predictions_train_S1[predictions_train_S1 < 0] = 0

# Calculate costs and EmCO2 separately for training set
y_pred_train_costs = predictions_train_S1 * s_price
y_pred_train_EmCo2 = predictions_train_S1 * s_EmCO2 * CO2_sprice


# TRAINING DATA
# Converting training data for costs
y_train_costs = y_train.copy()
y_train_costs *= s_price

# Converting training data for EmCO2
y_train_EmCO2 = y_train.copy()
y_train_EmCO2 *= s_EmCO2 * CO2_sprice

# Calculation of MSE and RMSE
mse_linear_train_EmCo2 = mean_squared_error(y_train_EmCO2, y_pred_train_EmCo2)
mse_linear_train_costs = mean_squared_error(y_train_costs, y_pred_train_costs)

rmse_linear_train_EmCo2 = np.sqrt(mse_linear_train_EmCo2)
rmse_linear_train_costs = np.sqrt(mse_linear_train_costs)

print("S1: Braced Framed Tube - EmCO2: Linear Regression (Training Set), the MSE:", round(mse_linear_train_EmCo2, 3), "and RMSE:", round(r
print("S1: Braced Framed Tube - Costs: Linear Regression (Training Set), the MSE:", round(mse_linear_train_costs, 3), "and RMSE:", round(r
print("")


# TEST DATA
# Converting training data for costs
y_test_costs = y_test.copy()
y_test_costs *= s_price

# Converting training data for EmCO2
y_test_EmCO2 = y_test.copy()
y_test_EmCO2 *= s_EmCO2 * CO2_sprice

# Calculation of MSE and RMSE
mse_linear_test_EmCo2 = mean_squared_error(y_test_EmCO2, y_pred_test_EmCo2)
mse_linear_test_costs = mean_squared_error(y_test_costs, y_pred_test_costs)

rmse_linear_test_EmCo2 = np.sqrt(mse_linear_test_EmCo2)
rmse_linear_test_costs = np.sqrt(mse_linear_test_costs)

print("S1: Braced Framed Tube - EmCO2: Linear Regression (Test Set), the MSE:", round(mse_linear_test_EmCo2, 3), "and RMSE:", round(rmse_l
print("S1: Braced Framed Tube - Costs: Linear Regression (Test Set), the MSE:", round(mse_linear_test_costs, 3), "and RMSE:", round(rmse_l
```

Plot of actual and predicted values based on 1st Order

In [ ]:
```
# Create subplots
plt.figure(figsize=(12, 6))

# Plot for Costs
plt.subplot(1, 2, 1)

y_train_array = np.array(y_train_costs)
y_test_array = np.array(y_test_costs)
y_pred_test_costs_array = np.array(y_pred_test_costs)
y_pred_train_costs_array = np.array(y_pred_train_costs)

plt.scatter(y_test_array, y_pred_test_costs_array, color=color_purple, label='Test data')
plt.scatter(y_train_array, y_pred_train_costs_array, color=color_darkblue, label='Train data')

max_value_costs = max(y_test_array.max(), y_train_array.max(), y_pred_test_costs_array.max(), y_pred_train_costs_array.max())

plt.plot([0, max_value_costs], [0, max_value_costs], color=color_orange, linestyle='--', linewidth=2, label='True value')
plt.title('S1: Braced Framed Tube - Linear Regression: Total Costs')
plt.xlabel('Actual Total Costs [€/GFA]')
plt.ylabel('Predicted Total Costs [€/GFA]')
plt.legend()

# Plot for Embodied Carbon
plt.subplot(1, 2, 2)

y_train_array = np.array(y_train_EmCO2)
y_test_array = np.array(y_test_EmCO2)
y_pred_test_EmCo2_array = np.array(y_pred_test_EmCo2)
y_pred_train_EmCo2_array = np.array(y_pred_train_EmCo2)

plt.scatter(y_test_array, y_pred_test_EmCo2_array, color=color_purple, label='Test data')
plt.scatter(y_train_array, y_pred_train_EmCo2_array, color=color_darkblue, label='Train data')

max_value_EmCo2 = max(y_test_array.max(), y_train_array.max(), y_pred_test_EmCo2_array.max(), y_pred_train_EmCo2_array.max())

plt.plot([0, max_value_EmCo2], [0, max_value_EmCo2], color=color_orange, linestyle='--', linewidth=2, label='True value')
plt.title('S1: Braced Framed Tube - Linear Regression: Total Embodied Carbon')
plt.xlabel('Actual Total Embodied Carbon [€/GFA]')
plt.ylabel('Predicted Total Embodied Carbon [€/GFA]')
plt.legend()

# Adjust layout
plt.tight_layout()
plt.show()
```

C) Analysis of 2nd Polynomial Regression

Determination of 2nd Order in terms of MSE

In [ ]:
```
poly2nd = PolynomialFeatures(degree = 2, interaction_only = False, include_bias = True)

# Transform features to 2nd degree polynomial features
X_poly2nd_train = poly2nd.fit_transform(X_train_scaled)
X_poly2nd_test = poly2nd.transform(X_test_scaled)

# Train a Linear Regression model on the polynomial features
reg2nd = LinearRegression().fit(X_poly2nd_train, y_train)

# Predictions on the test set
predictions_test_S1 = reg2nd.predict(X_poly2nd_test)

# Calculate costs and EmCO2 separately for test set
y_pred_test_costs = predictions_test_S1 * s_price
y_pred_test_EmCo2 = predictions_test_S1 * s_EmCO2 * CO2_sprice

# Predictions on the training set
predictions_train_S1 = reg2nd.predict(X_poly2nd_train)

# Calculate costs and EmCO2 separately for training set
y_pred_train_costs = predictions_train_S1 * s_price
y_pred_train_EmCo2 = predictions_train_S1 * s_EmCO2 * CO2_sprice

# TRAINING DATA
# Converting training data for costs
y_train_costs = y_train.copy()
y_train_costs *= s_price

# Converting training data for EmCO2
y_train_EmCO2 = y_train.copy()
y_train_EmCO2 *= s_EmCO2 * CO2_sprice

# Calculation of MSE and RMSE
mse_2nd_train_EmCo2 = mean_squared_error(y_train_EmCO2, y_pred_train_EmCo2)
mse_2nd_train_costs = mean_squared_error(y_train_costs, y_pred_train_costs)

rmse_2nd_train_EmCo2 = np.sqrt(mse_2nd_train_EmCo2)
rmse_2nd_train_costs = np.sqrt(mse_2nd_train_costs)

print("S1: Braced Framed Tube - EmCO2: 2nd Regression (Training Set), the MSE:", round(mse_2nd_train_EmCo2, 3), "and RMSE:", round(rmse_2n
print("S1: Braced Framed Tube - Costs: 2nd Regression (Training Set), the MSE:", round(mse_2nd_train_costs, 3), "and RMSE:", round(rmse_2n
print("")


# TEST DATA
# Converting training data for costs
y_test_costs = y_test.copy()
y_test_costs *= s_price

# Converting training data for EmCO2
y_test_EmCO2 = y_test.copy()
y_test_EmCO2 *= s_EmCO2 * CO2_sprice

# Calculation of MSE and RMSE
mse_2nd_test_EmCo2 = mean_squared_error(y_test_EmCO2, y_pred_test_EmCo2)
mse_2nd_test_costs = mean_squared_error(y_test_costs, y_pred_test_costs)

rmse_2nd_test_EmCo2 = np.sqrt(mse_2nd_test_EmCo2)
rmse_2nd_test_costs = np.sqrt(mse_2nd_test_costs)

print("S1: Braced Framed Tube - EmCO2: 2nd Regression (Test Set), the MSE:", round(mse_2nd_test_EmCo2, 3), "and RMSE:", round(rmse_2nd_tes
print("S1: Braced Framed Tube - Costs: 2nd Regression (Test Set), the MSE:", round(mse_2nd_test_costs, 3), "and RMSE:", round(rmse_2nd_tes
```

Plot of actual and predicted values based on 2nd Order

In [ ]:
```
# Create subplots
plt.figure(figsize=(12, 6))

# Plot for Costs
plt.subplot(1, 2, 1)

y_train_array = np.array(y_train_costs)
y_test_array = np.array(y_test_costs)
y_pred_test_costs_array = np.array(y_pred_test_costs)
y_pred_train_costs_array = np.array(y_pred_train_costs)

plt.scatter(y_test_array, y_pred_test_costs_array, color=color_purple, label='Test data')
plt.scatter(y_train_array, y_pred_train_costs_array, color=color_darkblue, label='Train data')

max_value_costs = max(y_test_array.max(), y_train_array.max(), y_pred_test_costs_array.max(), y_pred_train_costs_array.max())

plt.plot([0, max_value_costs], [0, max_value_costs], color=color_orange, linestyle='--', linewidth=2, label='True value')
plt.title('S1: Braced Framed Tube - Polynomial degree 2: Total Costs')
plt.xlabel('Actual Total Costs [€/GFA]')
plt.ylabel('Predicted Total Costs [€/GFA]')
plt.legend()

# Plot for Embodied Carbon
plt.subplot(1, 2, 2)

y_train_array = np.array(y_train_EmCO2)
y_test_array = np.array(y_test_EmCO2)
y_pred_test_EmCo2_array = np.array(y_pred_test_EmCo2)
y_pred_train_EmCo2_array = np.array(y_pred_train_EmCo2)

plt.scatter(y_test_array, y_pred_test_EmCo2_array, color=color_purple, label='Test data')
plt.scatter(y_train_array, y_pred_train_EmCo2_array, color=color_darkblue, label='Train data')

max_value_EmCo2 = max(y_test_array.max(), y_train_array.max(), y_pred_test_EmCo2_array.max(), y_pred_train_EmCo2_array.max())

plt.plot([0, max_value_EmCo2], [0, max_value_EmCo2], color=color_orange, linestyle='--', linewidth=2, label='True value')
plt.title('S1: Braced Framed Tube - Polynomial degree 2: Total Embodied Carbon')
plt.xlabel('Actual Total Embodied Carbon [€/GFA]')
plt.ylabel('Predicted Total Embodied Carbon [€/GFA]')
plt.legend()

# Adjust layout
plt.tight_layout()
plt.show()
```

D) 3rd Polynomial Regression

Determination of 3rd in terms of MSE

In [ ]:
```
poly3rd = PolynomialFeatures(degree=3, interaction_only=False, include_bias=True)

# Transform features to 3rd degree polynomial features
X_poly3rd_train = poly3rd.fit_transform(X_train_scaled)
X_poly3rd_test = poly3rd.transform(X_test_scaled)

# Train a Linear Regression model on the polynomial features
reg3rd = LinearRegression().fit(X_poly3rd_train, y_train)

# Predictions on the test set
predictions_test_S1 = reg3rd.predict(X_poly3rd_test)

# Calculate costs and EmCO2 separately for test set
y_pred_test_costs = predictions_test_S1 * s_price
y_pred_test_EmCo2 = predictions_test_S1 * s_EmCO2 * CO2_sprice

# Predictions on the training set
predictions_train_S1 = reg3rd.predict(X_poly3rd_train)

# Calculate costs and EmCO2 separately for training set
y_pred_train_costs = predictions_train_S1 * s_price
y_pred_train_EmCo2 = predictions_train_S1 * s_EmCO2 * CO2_sprice

# TRAINING DATA
# Converting training data for costs
y_train_costs = y_train.copy()
y_train_costs *= s_price

# Converting training data for EmCO2
y_train_EmCO2 = y_train.copy()
y_train_EmCO2 *= s_EmCO2 * CO2_sprice

# Calculation of MSE and RMSE
mse_3rd_train_EmCo2 = mean_squared_error(y_train_EmCO2, y_pred_train_EmCo2)
mse_3rd_train_costs = mean_squared_error(y_train_costs, y_pred_train_costs)

rmse_3rd_train_EmCo2 = np.sqrt(mse_3rd_train_EmCo2)
rmse_3rd_train_costs = np.sqrt(mse_3rd_train_costs)

print("S1: Braced Framed Tube - EmCO2: 3rd Regression (Training Set), the MSE:", round(mse_3rd_train_EmCo2, 3), "and RMSE:", round(rmse_3r
print("S1: Braced Framed Tube - Costs: 3rd Regression (Training Set), the MSE:", round(mse_3rd_train_costs, 3), "and RMSE:", round(rmse_3r
print("")


# TEST DATA
# Converting training data for costs
y_test_costs = y_test.copy()
y_test_costs *= s_price

# Converting training data for EmCO2
y_test_EmCO2 = y_test.copy()
y_test_EmCO2 *= s_EmCO2 * CO2_sprice

# Calculation of MSE and RMSE
mse_3rd_test_EmCo2 = mean_squared_error(y_test_EmCO2, y_pred_test_EmCo2)
mse_3rd_test_costs = mean_squared_error(y_test_costs, y_pred_test_costs)

rmse_3rd_test_EmCo2 = np.sqrt(mse_3rd_test_EmCo2)
rmse_3rd_test_costs = np.sqrt(mse_3rd_test_costs)

print("S1: Braced Framed Tube - EmCO2: 3rd Regression (Test Set), the MSE:", round(mse_3rd_test_EmCo2, 3), "and RMSE:", round(rmse_3rd_tes
print("S1: Braced Framed Tube - Costs: 3rd Regression (Test Set), the MSE:", round(mse_3rd_test_costs, 3), "and RMSE:", round(rmse_3rd_tes
```

Plot of actual and predicted values based on 3rd Order

In [ ]:
```
# Create subplots
plt.figure(figsize=(12, 6))

# Plot for Costs
plt.subplot(1, 2, 1)

y_train_array = np.array(y_train_costs)
y_test_array = np.array(y_test_costs)
y_pred_test_costs_array = np.array(y_pred_test_costs)
y_pred_train_costs_array = np.array(y_pred_train_costs)

plt.scatter(y_test_array, y_pred_test_costs_array, color=color_purple, label='Test data')
plt.scatter(y_train_array, y_pred_train_costs_array, color=color_darkblue, label='Train data')

max_value_costs = max(y_test_array.max(), y_train_array.max(), y_pred_test_costs_array.max(), y_pred_train_costs_array.max())

plt.plot([0, max_value_costs], [0, max_value_costs], color=color_orange, linestyle='--', linewidth=2, label='True value')
plt.title('S1: Braced Framed Tube - Polynomial degree 3: Total Costs')
plt.xlabel('Actual Total Costs [€/GFA]')
plt.ylabel('Predicted Total Costs [€/GFA]')
plt.legend()

# Plot for Embodied Carbon
plt.subplot(1, 2, 2)

y_train_array = np.array(y_train_EmCO2)
y_test_array = np.array(y_test_EmCO2)
y_pred_test_EmCo2_array = np.array(y_pred_test_EmCo2)
y_pred_train_EmCo2_array = np.array(y_pred_train_EmCo2)

plt.scatter(y_test_array, y_pred_test_EmCo2_array, color=color_purple, label='Test data')
plt.scatter(y_train_array, y_pred_train_EmCo2_array, color=color_darkblue, label='Train data')

max_value_EmCo2 = max(y_test_array.max(), y_train_array.max(), y_pred_test_EmCo2_array.max(), y_pred_train_EmCo2_array.max())

plt.plot([0, max_value_EmCo2], [0, max_value_EmCo2], color=color_orange, linestyle='--', linewidth=2, label='True value')
plt.title('S1: Braced Framed Tube - Polynomial degree 3: Total Embodied Carbon')
plt.xlabel('Actual Total Embodied Carbon [€/GFA]')
plt.ylabel('Predicted Total Embodied Carbon [€/GFA]')
plt.legend()

# Adjust layout
plt.tight_layout()
plt.show()
```

In [ ]:
```
predictions_test_S1_ANN = best_model_S1.predict(X_test_scaled)

#Linear
linear_model = LinearRegression()
linear_model.fit(X_train_scaled, y_train)
predictions_test_S1_linear = linear_model.predict(X_test_scaled)

#2nd Order
poly2nd = PolynomialFeatures(degree = 2, interaction_only = False, include_bias = True)
X_poly2nd_train = poly2nd.fit_transform(X_train_scaled)
X_poly2nd_test = poly2nd.transform(X_test_scaled)
reg2nd = LinearRegression().fit(X_poly2nd_train, y_train)
predictions_test_S1_2nd = reg2nd.predict(X_poly2nd_test)

#3rd Order
poly3rd = PolynomialFeatures(degree = 3, interaction_only = False, include_bias = True)
X_poly3rd_train = poly3rd.fit_transform(X_train_scaled)
X_poly3rd_test = poly3rd.transform(X_test_scaled)
reg3rd = LinearRegression().fit(X_poly3rd_train, y_train)
predictions_test_S1_3rd = reg3rd.predict(X_poly3rd_test)

y_test_array = np.array(y_test)
y_test_array_pd = pd.DataFrame(y_test_array)

#ANN
diff_test_ANN = predictions_test_S1_ANN - y_test_array
diff_percentage_ANN = (diff_test_ANN / y_test_array) * 100
diff_percentage_ANN = np.array(diff_percentage_ANN)

mu_ANN = np.mean(diff_percentage_ANN)
std_ANN = np.std(diff_percentage_ANN)

#Linear
diff_test_linear = predictions_test_S1_linear - y_test_array
diff_percentage_linear = (diff_test_linear / y_test_array) * 100
diff_percentage_linear = np.array(diff_percentage_linear)

mu_linear = np.mean(diff_percentage_linear)
std_linear = np.std(diff_percentage_linear)

#2nd Order
diff_test_2nd = predictions_test_S1_2nd - y_test_array
diff_percentage_2nd = (diff_test_2nd / y_test_array) * 100
diff_percentage_2nd = np.array(diff_percentage_2nd)

mu_2nd = np.mean(diff_percentage_2nd)
std_2nd = np.std(diff_percentage_2nd)

#3rd Order
diff_test_3rd = predictions_test_S1_3rd - y_test_array
diff_percentage_3rd = (diff_test_3rd / y_test_array) * 100
diff_percentage_3rd = np.array(diff_percentage_3rd)

mu_3rd = np.mean(diff_percentage_3rd)
std_3rd = np.std(diff_percentage_3rd)


# Generate the x values for your normal distribution
x = np.linspace(-201, 201, 100)

# Generate the y values (probability density function) for your normal distribution
p_ANN = norm.pdf(x, mu_ANN, std_ANN)
p_linear = norm.pdf(x, mu_linear, std_linear)
p_2nd = norm.pdf(x, mu_2nd, std_2nd)
p_3rd = norm.pdf(x, mu_3rd, std_3rd)

plt.figure(figsize=(10,6))
# Plot the normal distribution
plt.plot(x, p_ANN, 'k', linewidth=2, color=color_orange, label='ANN')
plt.plot(x, p_linear, 'k', linewidth=2, color=color_grey, label='Linear')
plt.plot(x, p_2nd, 'k', linewidth=2, color=color_darkblue,label='2nd Order')
plt.plot(x, p_3rd, 'k', linewidth=2, color=color_green, label='3rd Order')

# Add labels and title
plt.xlabel('Relative Error [%]')
plt.ylabel('Probability Density [-]')
plt.title('Distribution of Relative Error of Braced Framed Tube (S1)')
plt.legend()

plt.axvline(0, color='black', linestyle='dotted')
plt.axvline(15, color=color_grey, linestyle='dotted')
plt.axvline(-15, color=color_grey, linestyle='dotted')

plt.xticks(np.arange(min(x)+1, max(x)+1, 25))

# Show the plot
plt.show()

# Compute RMSE for ANN
rmse_ANN = np.sqrt(np.mean((y_test_array - predictions_test_S1_ANN)**2))

# Compute RMSE for Linear
rmse_linear = np.sqrt(np.mean((y_test_array - predictions_test_S1_linear)**2))

# Compute RMSE for 2nd Order
rmse_2nd = np.sqrt(np.mean((y_test_array - predictions_test_S1_2nd)**2))

# Compute RMSE for 3rd Order
rmse_3rd = np.sqrt(np.mean((y_test_array - predictions_test_S1_3rd)**2))

print("The RMSE of ANN is equal to", round(rmse_ANN, 3))
print("The RMSE of Linear is equal to", round(rmse_linear, 3))
```

```
print("The RMSE of 2nd Order is equal to", round(rmse_2nd, 3))
print("The RMSE of 3rd Order is equal to", round(rmse_3rd, 3))
```

Total Probability

In [ ]:
```
from scipy.stats import norm
from scipy.integrate import quad

def integrate_pdf_within_range(pdf_function, lower_bound, upper_bound, *args):
    integral, _ = quad(pdf_function, lower_bound, upper_bound, args=args)
    return integral

# Define the range (-15% to +15%)
lower_bound = -15
upper_bound = 15

# Calculate the total probability within the specified range for each model
total_probability_ANN = integrate_pdf_within_range(norm.pdf, lower_bound, upper_bound, mu_ANN, std_ANN)
total_probability_linear = integrate_pdf_within_range(norm.pdf, lower_bound, upper_bound, mu_linear, std_linear)
total_probability_2nd = integrate_pdf_within_range(norm.pdf, lower_bound, upper_bound, mu_2nd, std_2nd)
total_probability_3rd = integrate_pdf_within_range(norm.pdf, lower_bound, upper_bound, mu_3rd, std_3rd)

print("Total Probability within -15% to +15% for ANN:", total_probability_ANN)
print("Total Probability within -15% to +15% for Linear:", total_probability_linear)
print("Total Probability within -15% to +15% for 2nd Order:", total_probability_2nd)
print("Total Probability within -15% to +15% for 3rd Order:", total_probability_3rd)
```

In [ ]:
```
print("The mean (mu) of ANN is equal to", round(mu_ANN, 2))
print("The mean (mu) of Linear is equal to", round(mu_linear, 2))
print("The mean (mu) of 2nd Order is equal to", round(mu_2nd, 2))
print("The mean (mu) of 3rd Order is equal to", round(mu_3rd, 2))
print("")
print("The standard deviation (sigma) of ANN is equal to", round(std_ANN, 1))
print("The standard deviation (sigma) of Linear is equal to", round(std_linear, 1))
print("The standard deviation (sigma) of 2nd Order is equal to", round(std_2nd, 1))
print("The standard deviation (sigma) of 3rd Order is equal to", round(std_3rd, 1))
```

Determination of MAPE

In [ ]:
```
# Compute MAPE for ANN
mape_ANN = np.mean(np.abs(diff_percentage_ANN))
print("The MAPE of ANN is equal to", round(mape_ANN, 3))

# Compute MAPE for Linear
mape_linear = np.mean(np.abs(diff_percentage_linear))
print("The MAPE of Linear is equal to", round(mape_linear, 3))

# Compute MAPE for 2nd Order
mape_2nd = np.mean(np.abs(diff_percentage_2nd))
print("The MAPE of 2nd Order is equal to", round(mape_2nd, 3))

# Compute MAPE for 3rd Order
mape_3rd = np.mean(np.abs(diff_percentage_3rd))
print("The MAPE of 3rd Order is equal to", round(mape_3rd, 3))
```

4b) ANALYSIS OF MODEL RESULT : OUTRIGGER SYSTEM

COMPARISON INTERPOLATION TECHNIQUES

A) Analysis of ANN

Determination of ANN in terms of MSE

In [ ]:
```
# Predictions on the test set
predictions_test_S2 = best_model_S2.predict(X_test_scaled2)

# Calculate costs and EmCO2 separately for test set
y_pred_test_costs =  predictions_test_S2[:, 0] * s_price + predictions_test_S2[:, 1] * pc_rc_price

y_pred_test_EmCo2 = predictions_test_S2[:, 0] * s_EmCO2 * CO2_sprice + predictions_test_S2[:, 1] * pc_rc_EmCO2 * CO2_sprice

# Predictions on the training set
predictions_train_S2 = best_model_S2.predict(X_train_scaled2)

# Calculate costs and EmCO2 separately for training set
y_pred_train_costs = predictions_train_S2[:, 0] * s_price + predictions_train_S2[:, 1] * pc_rc_price
y_pred_train_EmCo2 = predictions_train_S2[:, 0] * s_EmCO2 * CO2_sprice + predictions_train_S2[:, 1] * pc_rc_EmCO2 * CO2_sprice

# TRAINING DATA
y_train_costs = y_train2.copy()
y_train_costs.iloc[:, 0] *= s_price
y_train_costs.iloc[:, 1] *= pc_rc_price
y_train_costs = y_train_costs.iloc[:, 0] + y_train_costs.iloc[:, 1]

# Converting training data for EmCO2
y_train_EmCO2 = y_train2.copy()
y_train_EmCO2.iloc[:, 0] *= s_EmCO2 * CO2_sprice
y_train_EmCO2.iloc[:, 1] *= pc_rc_EmCO2 * CO2_sprice
y_train_EmCO2 = y_train_EmCO2.iloc[:, 0] + y_train_EmCO2.iloc[:, 1]

# Calculation of MSE and RMSE
mse_ann_train_EmCo2 = mean_squared_error(y_train_EmCO2, y_pred_train_EmCo2)
mse_ann_train_costs = mean_squared_error(y_train_costs, y_pred_train_costs)

rmse_ann_train_EmCo2 = np.sqrt(mse_ann_train_EmCo2)
rmse_ann_train_costs = np.sqrt(mse_ann_train_costs)

print("S2: Outrigger - EmCO2: ANN Model (Training Set), the MSE:", round(mse_ann_train_EmCo2, 3), "and RMSE:", round(rmse_ann_train_EmCo2,
print("S2: Outrigger - Costs: ANN Model (Training Set), the MSE:", round(mse_ann_train_costs, 3), "and RMSE:", round(rmse_ann_train_costs,
print("")


# TEST DATA
# Converting training data for costs
y_test_costs = y_test2.copy()
y_test_costs.iloc[:, 0] *= s_price
y_test_costs.iloc[:, 1] *= pc_rc_price
y_test_costs = y_test_costs.iloc[:, 0] + y_test_costs.iloc[:, 1]

# Converting training data for EmCO2
y_test_EmCO2 = y_test2.copy()
y_test_EmCO2.iloc[:, 0] *= s_EmCO2 * CO2_sprice
y_test_EmCO2.iloc[:, 1] *= pc_rc_EmCO2 * CO2_sprice
y_test_EmCO2 = y_test_EmCO2.iloc[:, 0] + y_test_EmCO2.iloc[:, 1]

# Calculation of MSE and RMSE
mse_ann_test_EmCo2 = mean_squared_error(y_test_EmCO2, y_pred_test_EmCo2)
mse_ann_test_costs = mean_squared_error(y_test_costs, y_pred_test_costs)

rmse_ann_test_EmC02_S2 = np.sqrt(mse_ann_test_EmCo2)
rmse_ann_test_costs_S2 = np.sqrt(mse_ann_test_costs)

print("S2: Outrigger - EmCO2: ANN Model (Test Set), the MSE:", round(mse_ann_test_EmCo2, 3), "and RMSE:", round(rmse_ann_test_EmC02_S2, 3)
print("S2: Outrigger - Costs: ANN Model (Test Set), the MSE:", round(mse_ann_test_costs, 3), "and RMSE:", round(rmse_ann_test_costs_S2, 3)
```

In [ ]:
```
# Predictions on the test set
predictions_test_S2 = best_model_S2.predict(X_test_scaled2)

# Predictions on the training set
predictions_train_S2 = best_model_S2.predict(X_train_scaled2)

# Calculate RMSE for test set for costs and EmCO2 separately
rmse_ann_test_steel_S2 = np.sqrt(mean_squared_error(y_test2.iloc[:, 0], predictions_test_S2[:, 0]))
rmse_ann_test_concrete_S2 = np.sqrt(mean_squared_error(y_test2.iloc[:, 1], predictions_test_S2[:, 1]))

print("S2: Outrigger - RMSE Steel:", round(rmse_ann_test_costs_S2, 3))
print("S2: Outrigger - RMSE Concrete:", round(rmse_ann_test_EmC02_S2, 3))
```

Plot of actual and predicted values based on ANN

In [ ]:
```
# Create subplots
plt.figure(figsize=(12, 6))

# Plot for Costs
plt.subplot(1, 2, 1)

y_train_array = np.array(y_train_costs)
y_test_array = np.array(y_test_costs)
y_pred_test_costs_array = np.array(y_pred_test_costs)
y_pred_train_costs_array = np.array(y_pred_train_costs)

plt.scatter(y_test_array, y_pred_test_costs_array, color=color_purple, label='Test data')
plt.scatter(y_train_array, y_pred_train_costs_array, color=color_darkblue, label='Train data')

max_value_costs = max(y_test_array.max(), y_train_array.max(), y_pred_test_costs_array.max(), y_pred_train_costs_array.max())

plt.plot([0, max_value_costs], [0, max_value_costs], color=color_orange, linestyle='--', linewidth=2, label='True value')
plt.title('S2: Outrigger - ANN: Total Structural Costs')
plt.xlabel('Actual Total Structural Costs [€/m2GFA]')
plt.ylabel('Predicted Total Structural Costs [€/m2GFA]')
plt.legend()

# Plot for Embodied Carbon
plt.subplot(1, 2, 2)

y_train_array = np.array(y_train_EmCO2)
y_test_array = np.array(y_test_EmCO2)
y_pred_test_EmCo2_array = np.array(y_pred_test_EmCo2)
y_pred_train_EmCo2_array = np.array(y_pred_train_EmCo2)

plt.scatter(y_test_array, y_pred_test_EmCo2_array, color=color_purple, label='Test data')
plt.scatter(y_train_array, y_pred_train_EmCo2_array, color=color_darkblue, label='Train data')

max_value_EmCo2 = max(y_test_array.max(), y_train_array.max(), y_pred_test_EmCo2_array.max(), y_pred_train_EmCo2_array.max())

plt.plot([0, max_value_EmCo2], [0, max_value_EmCo2], color=color_orange, linestyle='--', linewidth=2, label='True value')
plt.title('S2: Outrigger - ANN: Total Embodied Carbon')
plt.xlabel('Actual Total Embodied Carbon [€/m2GFA]')
plt.ylabel('Predicted Total Embodied Carbon [€/m2GFA]')
plt.legend()

# Adjust layout
plt.tight_layout()
plt.show()
```

Plot convergence graph of ANN

In [ ]:
```
fig, axes = plt.subplots(1, 2, figsize=(12, 5))  # Create a figure with 1 row and 2 columns

# Plot the original plot on the first subplot
axes[0].plot(best_model_S2.losses['training_loss'], label='Training Loss', color = color_darkblue)
axes[0].plot(best_model_S2.losses['validation_loss'], label='Validation Loss',  color = color_orange)

#axes[0].set_ylim(bottom=-1000000, top=ymax+5000)
#axes[1].set_xlim(left=)

axes[0].set_title('Convergence Plot for Outrigger (S2)')
axes[0].set_xlabel('Epochs')
axes[0].set_ylabel('Loss [MSE]')
axes[0].legend()

ymax = np.max(best_model_S2.losses['validation_loss'])

axes[0].set_xlim(left=-500, right=10000)
axes[0].set_ylim(bottom=-1000, top=ymax)

# Plot the zoomed-in plot on the second subplot
axes[1].plot(best_model_S2.losses['training_loss'], label='Training Loss', color = color_darkblue)
axes[1].plot(best_model_S2.losses['validation_loss'], label='Validation Loss', color = color_orange)
axes[1].set_title('Zoomed-in Convergence Plot for Outrigger (S2)')
axes[1].set_xlabel('Epochs')
axes[1].set_ylabel('Loss [MSE]')
axes[1].legend()

# Set limits for better visualization on the zoomed-in plot
axes[1].set_xlim(left=-20, right=800)
axes[1].set_ylim(bottom=-1000, top=ymax+5000)

plt.tight_layout()
plt.show()
```

B) Analysis of Linear Regression

Determination of 1st Order in terms of MSE

In [ ]:
```
linear_model = LinearRegression()
linear_model.fit(X_train_scaled2, y_train2)

# Predictions on the test set
predictions_test_S2 = linear_model.predict(X_test_scaled2)
predictions_test_S2[predictions_test_S2 < 0] = 0

# Calculate costs and EmCO2 separately for test set
y_pred_test_costs =  predictions_test_S2[:, 0] * s_price + predictions_test_S2[:, 1] * pc_rc_price
y_pred_test_EmCo2 = predictions_test_S2[:, 0] * s_EmCO2 * CO2_sprice + predictions_test_S2[:, 1] * pc_rc_EmCO2 * CO2_sprice

# Predictions on the training set
predictions_train_S2 = linear_model.predict(X_train_scaled2)
predictions_train_S2[predictions_train_S2 < 0] = 0

# Calculate costs and EmCO2 separately for training set
y_pred_train_costs =  predictions_train_S2[:, 0] * s_price + predictions_train_S2[:, 1] * pc_rc_price
y_pred_train_EmCo2 = predictions_train_S2[:, 0] * s_EmCO2 * CO2_sprice + predictions_train_S2[:, 1] * pc_rc_EmCO2 * CO2_sprice

# TRAINING DATA
y_train_costs = y_train2.copy()
y_train_costs.iloc[:, 0] *= s_price
y_train_costs.iloc[:, 1] *= pc_rc_price
y_train_costs = y_train_costs.iloc[:, 0] + y_train_costs.iloc[:, 1]

# Converting training data for EmCO2
y_train_EmCO2 = y_train2.copy()
y_train_EmCO2.iloc[:, 0] *= s_EmCO2 * CO2_sprice
y_train_EmCO2.iloc[:, 1] *= pc_rc_EmCO2 * CO2_sprice
y_train_EmCO2 = y_train_EmCO2.iloc[:, 0] + y_train_EmCO2.iloc[:, 1]

# Calculation of MSE and RMSE
mse_linear_train_EmCo2 = mean_squared_error(y_train_EmCO2, y_pred_train_EmCo2)
mse_linear_train_costs = mean_squared_error(y_train_costs, y_pred_train_costs)

rmse_linear_train_EmCo2 = np.sqrt(mse_linear_train_EmCo2)
rmse_linear_train_costs = np.sqrt(mse_linear_train_costs)

print("S2: Outrigger - EmCO2: Linear Regression (Training Set), the MSE:", round(mse_linear_train_EmCo2, 3), "and RMSE:", round(rmse_linea
print("S2: Outrigger - Costs: Linear Regression (Training Set), the MSE:", round(mse_linear_train_costs, 3), "and RMSE:", round(rmse_linea
print("")

# TEST DATA
# Converting training data for costs
y_test_costs = y_test2.copy()
y_test_costs.iloc[:, 0] *= s_price
y_test_costs.iloc[:, 1] *= pc_rc_price
y_test_costs = y_test_costs.iloc[:, 0] + y_test_costs.iloc[:, 1]

# Converting training data for EmCO2
y_test_EmCO2 = y_test2.copy()
y_test_EmCO2.iloc[:, 0] *= s_EmCO2 * CO2_sprice
y_test_EmCO2.iloc[:, 1] *= pc_rc_EmCO2 * CO2_sprice
y_test_EmCO2 = y_test_EmCO2.iloc[:, 0] + y_test_EmCO2.iloc[:, 1]

# Calculation of MSE and RMSE
mse_linear_test_EmCo2 = mean_squared_error(y_test_EmCO2, y_pred_test_EmCo2)
mse_linear_test_costs = mean_squared_error(y_test_costs, y_pred_test_costs)

rmse_linear_test_EmCo2 = np.sqrt(mse_linear_test_EmCo2)
rmse_linear_test_costs = np.sqrt(mse_linear_test_costs)

print("S2: Outrigger - EmCO2: Linear Regression (Test Set), the MSE:", round(mse_linear_test_EmCo2, 3), "and RMSE:", round(rmse_linear_tes
print("S2: Outrigger - Costs: Linear Regression (Test Set), the MSE:", round(mse_linear_test_costs, 3), "and RMSE:", round(rmse_linear_tes
```

Plot of actual and predicted values based on 1st Order

In [ ]:
```
# Create subplots
plt.figure(figsize=(12, 6))

# Plot for Costs
plt.subplot(1, 2, 1)

y_train_array = np.array(y_train_costs)
y_test_array = np.array(y_test_costs)
y_pred_test_costs_array = np.array(y_pred_test_costs)
y_pred_train_costs_array = np.array(y_pred_train_costs)

plt.scatter(y_test_array, y_pred_test_costs_array, color=color_purple, label='Test data')
plt.scatter(y_train_array, y_pred_train_costs_array, color=color_darkblue, label='Train data')

max_value_costs = max(y_test_array.max(), y_train_array.max(), y_pred_test_costs_array.max(), y_pred_train_costs_array.max())

plt.plot([0, max_value_costs], [0, max_value_costs], color=color_orange, linestyle='--', linewidth=2, label='True value')
plt.title('S2: Outrigger - Linear Regression: Total Structural Costs')
plt.xlabel('Actual Total Structural Costs [€/m2GFA]')
plt.ylabel('Predicted Total Structural Costs [€/m2GFA]')
plt.legend()

# Plot for Embodied Carbon
plt.subplot(1, 2, 2)

y_train_array = np.array(y_train_EmCO2)
y_test_array = np.array(y_test_EmCO2)
y_pred_test_EmCo2_array = np.array(y_pred_test_EmCo2)
y_pred_train_EmCo2_array = np.array(y_pred_train_EmCo2)

plt.scatter(y_test_array, y_pred_test_EmCo2_array, color=color_purple, label='Test data')
plt.scatter(y_train_array, y_pred_train_EmCo2_array, color=color_darkblue, label='Train data')

max_value_EmCo2 = max(y_test_array.max(), y_train_array.max(), y_pred_test_EmCo2_array.max(), y_pred_train_EmCo2_array.max())

plt.plot([0, max_value_EmCo2], [0, max_value_EmCo2], color=color_orange, linestyle='--', linewidth=2, label='True value')
plt.title('S2: Outrigger - Linear Regression: Total Embodied Carbon')
plt.xlabel('Actual Total Embodied Carbon [€/m2GFA]')
plt.ylabel('Predicted Total Embodied Carbon [€/m2GFA]')
plt.legend()

# Adjust layout
plt.tight_layout()
plt.show()
```

C) Analysis of 2nd Polynomial Regression

Determination of 2nd Order in terms of MSE

In [ ]:
```
poly2nd = PolynomialFeatures(degree = 2, interaction_only = False, include_bias = True)

# Transform features to 2nd degree polynomial features
X_poly2nd_train = poly2nd.fit_transform(X_train_scaled2)
X_poly2nd_test = poly2nd.transform(X_test_scaled2)

# Train a Linear Regression model on the polynomial features
reg2nd = LinearRegression().fit(X_poly2nd_train, y_train2)

# Predictions on the test set
predictions_test_S2 = reg2nd.predict(X_poly2nd_test)
predictions_train_S2[predictions_train_S2 < 0] = 0

# Calculate costs and EmCO2 separately for test set
y_pred_test_costs =  predictions_test_S2[:, 0] * s_price + predictions_test_S2[:, 1] * pc_rc_price
y_pred_test_costs[y_pred_test_costs < 0] = 0

y_pred_test_EmCo2 = predictions_test_S2[:, 0] * s_EmCO2 * CO2_sprice + predictions_test_S2[:, 1] * pc_rc_EmCO2 * CO2_sprice
y_pred_test_EmCo2[y_pred_test_EmCo2 < 0] = 0

# Predictions on the training set
predictions_train_S2 = reg2nd.predict(X_poly2nd_train)
predictions_train_S2[predictions_train_S2 < 0] = 0

# Calculate costs and EmCO2 separately for training set
y_pred_train_costs =  predictions_train_S2[:, 0] * s_price + predictions_train_S2[:, 1] * pc_rc_price

y_pred_train_EmCo2 = predictions_train_S2[:, 0] * s_EmCO2 * CO2_sprice + predictions_train_S2[:, 1] * pc_rc_EmCO2 * CO2_sprice

# TRAINING DATA
y_train_costs = y_train2.copy()
y_train_costs.iloc[:, 0] *= s_price
y_train_costs.iloc[:, 1] *= pc_rc_price
y_train_costs = y_train_costs.iloc[:, 0] + y_train_costs.iloc[:, 1]

# Converting training data for EmCO2
y_train_EmCO2 = y_train2.copy()
y_train_EmCO2.iloc[:, 0] *= s_EmCO2 * CO2_sprice
y_train_EmCO2.iloc[:, 1] *= pc_rc_EmCO2 * CO2_sprice
y_train_EmCO2 = y_train_EmCO2.iloc[:, 0] + y_train_EmCO2.iloc[:, 1]


# Calculation of MSE and RMSE
mse_2nd_train_EmCo2 = mean_squared_error(y_train_EmCO2, y_pred_train_EmCo2)
mse_2nd_train_costs = mean_squared_error(y_train_costs, y_pred_train_costs)

rmse_2nd_train_EmCo2 = np.sqrt(mse_2nd_train_EmCo2)
rmse_2nd_train_costs = np.sqrt(mse_2nd_train_costs)

print("S2: Outrigger - EmCO2: 2nd Regression (Training Set), the MSE:", round(mse_2nd_train_EmCo2, 3), "and RMSE:", round(rmse_2nd_train_E
print("S2: Outrigger - Costs: 2nd Regression (Training Set), the MSE:", round(mse_2nd_train_costs, 3), "and RMSE:", round(rmse_2nd_train_c
print("")

# TEST DATA
# Converting training data for costs
y_test_costs = y_test2.copy()
y_test_costs.iloc[:, 0] *= s_price
y_test_costs.iloc[:, 1] *= pc_rc_price
y_test_costs = y_test_costs.iloc[:, 0] + y_test_costs.iloc[:, 1]

# Converting training data for EmCO2
y_test_EmCO2 = y_test2.copy()
y_test_EmCO2.iloc[:, 0] *= s_EmCO2 * CO2_sprice
y_test_EmCO2.iloc[:, 1] *= pc_rc_EmCO2 * CO2_sprice
y_test_EmCO2 = y_test_EmCO2.iloc[:, 0] + y_test_EmCO2.iloc[:, 1]

# Calculation of MSE and RMSE
mse_2nd_test_EmCo2 = mean_squared_error(y_test_EmCO2, y_pred_test_EmCo2)
mse_2nd_test_costs = mean_squared_error(y_test_costs, y_pred_test_costs)

rmse_2nd_test_EmCo2 = np.sqrt(mse_2nd_test_EmCo2)
rmse_2nd_test_costs = np.sqrt(mse_2nd_test_costs)

print("S2: Outrigger - EmCO2: 2nd Regression (Test Set), the MSE:", round(mse_2nd_test_EmCo2, 3), "and RMSE:", round(rmse_2nd_test_EmCo2,
print("S2: Outrigger - Costs: 2nd Regression (Test Set), the MSE:", round(mse_2nd_test_costs, 3), "and RMSE:", round(rmse_2nd_test_costs,
```

Plot of actual and predicted values based on 2nd Order

In [ ]:
```
# Create subplots
plt.figure(figsize=(12, 6))

# Plot for Costs
plt.subplot(1, 2, 1)

y_train_array = np.array(y_train_costs)
y_test_array = np.array(y_test_costs)
y_pred_test_costs_array = np.array(y_pred_test_costs)
y_pred_train_costs_array = np.array(y_pred_train_costs)

plt.scatter(y_test_array, y_pred_test_costs_array, color=color_purple, label='Test data')
plt.scatter(y_train_array, y_pred_train_costs_array, color=color_darkblue, label='Train data')

max_value_costs = max(y_test_array.max(), y_train_array.max(), y_pred_test_costs_array.max(), y_pred_train_costs_array.max())

plt.plot([0, max_value_costs], [0, max_value_costs], color=color_orange, linestyle='--', linewidth=2, label='True value')
plt.title('S2: Outrigger - Polynomial degree 2: Total Structural Costs')
plt.xlabel('Actual Total Structural Costs [€/m2GFA]')
plt.ylabel('Predicted Total Structural Costs [€/m2GFA]')
plt.legend()

# Plot for Embodied Carbon
plt.subplot(1, 2, 2)

y_train_array = np.array(y_train_EmCO2)
y_test_array = np.array(y_test_EmCO2)
y_pred_test_EmCo2_array = np.array(y_pred_test_EmCo2)
y_pred_train_EmCo2_array = np.array(y_pred_train_EmCo2)

plt.scatter(y_test_array, y_pred_test_EmCo2_array, color=color_purple, label='Test data')
plt.scatter(y_train_array, y_pred_train_EmCo2_array, color=color_darkblue, label='Train data')

max_value_EmCo2 = max(y_test_array.max(), y_train_array.max(), y_pred_test_EmCo2_array.max(), y_pred_train_EmCo2_array.max())

plt.plot([0, max_value_EmCo2], [0, max_value_EmCo2], color=color_orange, linestyle='--', linewidth=2, label='True value')
plt.title('S2: Outrigger - Polynomial degree 2: Total Embodied Carbon')
plt.xlabel('Actual Total Embodied Carbon [€/m2GFA]')
plt.ylabel('Predicted Total Embodied Carbon [€/m2GFA]')
plt.legend()

# Adjust layout
plt.tight_layout()
plt.show()
```

D) Analysis of 3rd Polynomial Regression

Determination of 3rd Order in terms of MSE

In [ ]:
```
poly3rd = PolynomialFeatures(degree=3, interaction_only=False, include_bias=True)

# Transform features to 3rd degree polynomial features
X_poly3rd_train = poly3rd.fit_transform(X_train_scaled2)
X_poly3rd_test = poly3rd.transform(X_test_scaled2)

# Train a Linear Regression model on the polynomial features
reg3rd = LinearRegression().fit(X_poly3rd_train, y_train2)

# Predictions on the test set
predictions_test_S2 = reg3rd.predict(X_poly3rd_test)
predictions_test_S2[predictions_test_S2 < 0] = 0

# Calculate costs and EmCO2 separately for test set
y_pred_test_costs =  predictions_test_S2[:, 0] * s_price + predictions_test_S2[:, 1] * pc_rc_price

y_pred_test_EmCo2 = predictions_test_S2[:, 0] * s_EmCO2 * CO2_sprice + predictions_test_S2[:, 1] * pc_rc_EmCO2 * CO2_sprice

# Predictions on the training set
predictions_train_S2 = reg3rd.predict(X_poly3rd_train)
predictions_train_S2[predictions_train_S2 < 0] = 0

# Calculate costs and EmCO2 separately for training set
y_pred_train_costs =  predictions_train_S2[:, 0] * s_price + predictions_train_S2[:, 1] * pc_rc_price

y_pred_train_EmCo2 = predictions_train_S2[:, 0] * s_EmCO2 * CO2_sprice + predictions_train_S2[:, 1] * pc_rc_EmCO2 * CO2_sprice

# TRAINING DATA
y_train_costs = y_train2.copy()
y_train_costs.iloc[:, 0] *= s_price
y_train_costs.iloc[:, 1] *= pc_rc_price
y_train_costs = y_train_costs.iloc[:, 0] + y_train_costs.iloc[:, 1]

# Converting training data for EmCO2
y_train_EmCO2 = y_train2.copy()
y_train_EmCO2.iloc[:, 0] *= s_EmCO2 * CO2_sprice
y_train_EmCO2.iloc[:, 1] *= pc_rc_EmCO2 * CO2_sprice
y_train_EmCO2 = y_train_EmCO2.iloc[:, 0] + y_train_EmCO2.iloc[:, 1]

# Calculation of MSE and RMSE
mse_3rd_train_EmCo2 = mean_squared_error(y_train_EmCO2, y_pred_train_EmCo2)
mse_3rd_train_costs = mean_squared_error(y_train_costs, y_pred_train_costs)

rmse_3rd_train_EmCo2 = np.sqrt(mse_3rd_train_EmCo2)
rmse_3rd_train_costs = np.sqrt(mse_3rd_train_costs)

print("S2: Outrigger - EmCO2: 3rd Regression (Training Set), the MSE:", round(mse_3rd_train_EmCo2, 3), "and RMSE:", round(rmse_3rd_train_E
print("S2: Outrigger - Costs: 3rd Regression (Training Set), the MSE:", round(mse_3rd_train_costs, 3), "and RMSE:", round(rmse_3rd_train_c
print("")


# TEST DATA
# Converting training data for costs
y_test_costs = y_test2.copy()
y_test_costs.iloc[:, 0] *= s_price
y_test_costs.iloc[:, 1] *= pc_rc_price
y_test_costs = y_test_costs.iloc[:, 0] + y_test_costs.iloc[:, 1]

# Converting training data for EmCO2
y_test_EmCO2 = y_test2.copy()
y_test_EmCO2.iloc[:, 0] *= s_EmCO2 * CO2_sprice
y_test_EmCO2.iloc[:, 1] *= pc_rc_EmCO2 * CO2_sprice
y_test_EmCO2 = y_test_EmCO2.iloc[:, 0] + y_test_EmCO2.iloc[:, 1]

# Calculation of MSE and RMSE
mse_3rd_test_EmCo2 = mean_squared_error(y_test_EmCO2, y_pred_test_EmCo2)
mse_3rd_test_costs = mean_squared_error(y_test_costs, y_pred_test_costs)

rmse_3rd_test_EmCo2 = np.sqrt(mse_3rd_test_EmCo2)
rmse_3rd_test_costs = np.sqrt(mse_3rd_test_costs)

print("S2: Outrigger - EmCO2: 3rd Regression (Test Set), the MSE:", round(mse_3rd_test_EmCo2, 3), "and RMSE:", round(rmse_3rd_test_EmCo2,
print("S2: Outrigger - Costs: 3rd Regression (Test Set), the MSE:", round(mse_3rd_test_costs, 3), "and RMSE:", round(rmse_3rd_test_costs,
```

Plot of actual and predicted values based on 3rd Order

In [ ]:
```
# Create subplots
plt.figure(figsize=(12, 6))

# Plot for Costs
plt.subplot(1, 2, 1)

y_train_array = np.array(y_train_costs)
y_test_array = np.array(y_test_costs)
y_pred_test_costs_array = np.array(y_pred_test_costs)
y_pred_train_costs_array = np.array(y_pred_train_costs)

plt.scatter(y_test_array, y_pred_test_costs_array, color=color_purple, label='Test data')
plt.scatter(y_train_array, y_pred_train_costs_array, color=color_darkblue, label='Train data')

max_value_costs = max(y_test_array.max(), y_train_array.max(), y_pred_test_costs_array.max(), y_pred_train_costs_array.max())

plt.plot([0, max_value_costs], [0, max_value_costs], color=color_orange, linestyle='--', linewidth=2, label='True value')
plt.title('S2: Outrigger - Polynomial degree 3: Total Structural Costs')
plt.xlabel('Actual Total Structural Costs [€/m2GFA]')
plt.ylabel('Predicted Total Structural Costs [€/m2GFA]')
plt.legend()

# Plot for Embodied Carbon
plt.subplot(1, 2, 2)

y_train_array = np.array(y_train_EmCO2)
y_test_array = np.array(y_test_EmCO2)
y_pred_test_EmCo2_array = np.array(y_pred_test_EmCo2)
y_pred_train_EmCo2_array = np.array(y_pred_train_EmCo2)

plt.scatter(y_test_array, y_pred_test_EmCo2_array, color=color_purple, label='Test data')
plt.scatter(y_train_array, y_pred_train_EmCo2_array, color=color_darkblue, label='Train data')

max_value_EmCo2 = max(y_test_array.max(), y_train_array.max(), y_pred_test_EmCo2_array.max(), y_pred_train_EmCo2_array.max())

plt.plot([0, max_value_EmCo2], [0, max_value_EmCo2], color=color_orange, linestyle='--', linewidth=2, label='True value')
plt.title('S2: Outrigger - Polynomial degree 3: Total Embodied Carbon')
plt.xlabel('Actual Total Embodied Carbon [€/m2GFA]')
plt.ylabel('Predicted Total Embodied Carbon [€/m2GFA]')
plt.legend()

# Adjust layout
plt.tight_layout()
plt.show()
```

Graph of Distrubition of Relative Error

In [ ]:
```
diff_test_ANN
```

In [ ]:
```
predictions_test_S2_ANN = best_model_S2.predict(X_test_scaled2)

#Linear
linear_model = LinearRegression()
linear_model.fit(X_train_scaled2, y_train2)
predictions_test_S2_linear = linear_model.predict(X_test_scaled2)
predictions_test_S2_linear[predictions_test_S2_linear < 0] = 0
relative_errors_test_costs_linear = ((y_test_array2 - predictions_test_S2_linear) / y_test_array2) * 100
mu_test, std_test = norm.fit(relative_errors_test_costs_linear)


#2nd Order
poly2nd = PolynomialFeatures(degree = 2, interaction_only = False, include_bias = True)
X_poly2nd_train2 = poly2nd.fit_transform(X_train_scaled2)
X_poly2nd_test2 = poly2nd.transform(X_test_scaled2)
reg2nd = LinearRegression().fit(X_poly2nd_train2, y_train2)
predictions_test_S2_2nd = reg2nd.predict(X_poly2nd_test2)
predictions_test_S2_2nd[predictions_test_S2_2nd < 0] = 0

#3rd Order
poly3rd = PolynomialFeatures(degree = 3, interaction_only = False, include_bias = True)
X_poly3rd_train2 = poly3rd.fit_transform(X_train_scaled2)
X_poly3rd_test2 = poly3rd.transform(X_test_scaled2)
reg3rd = LinearRegression().fit(X_poly3rd_train2, y_train2)
predictions_test_S2_3rd = reg3rd.predict(X_poly3rd_test2)
predictions_test_S2_3rd[predictions_test_S2_3rd < 0] = 0

y_test_array2 = np.array(y_test2)
y_test_array_pd2 = pd.DataFrame(y_test_array2)

#ANN
y_test_total = (y_test_array2[:, 0] + y_test_array2[:, 1])
diff_test_ANN = (predictions_test_S2_ANN[:, 0] + predictions_test_S2_ANN[:, 1]) - y_test_total
diff_percentage_ANN = (diff_test_ANN / y_test_total) * 100
diff_percentage_ANN = np.array(diff_percentage_ANN)

mu_ANN = np.mean(diff_percentage_ANN)
std_ANN = np.std(diff_percentage_ANN)

#Linear
diff_test_linear = (predictions_test_S2_linear[:, 0] + predictions_test_S2_linear[:, 0]) - y_test_total
diff_percentage_linear = (diff_test_linear / y_test_total) * 100
diff_percentage_linear = np.array(diff_percentage_linear)

mu_linear = np.mean(diff_percentage_linear)
std_linear = np.std(diff_percentage_linear)

#2nd Order
diff_test_2nd = (predictions_test_S2_2nd[:, 0] + predictions_test_S2_2nd[:, 1]) - y_test_total
diff_percentage_2nd = (diff_test_2nd / y_test_total) * 100
diff_percentage_2nd = np.array(diff_percentage_2nd)

mu_2nd = np.mean(diff_percentage_2nd)
std_2nd = np.std(diff_percentage_2nd)

#3rd Order
diff_test_3rd = (predictions_test_S2_3rd[:, 0] + predictions_test_S2_3rd[:, 1]) - y_test_total
diff_percentage_3rd = (diff_test_3rd / y_test_total) * 100
diff_percentage_3rd = np.array(diff_percentage_3rd)

mu_3rd = np.mean(diff_percentage_3rd)
std_3rd = np.std(diff_percentage_3rd)


# Generate the x values for your normal distribution
x = np.linspace(-201, 201, 100)

# Generate the y values (probability density function) for your normal distribution
p_ANN = norm.pdf(x, mu_ANN, std_ANN)
p_linear = norm.pdf(x, mu_linear, std_linear)
p_2nd = norm.pdf(x, mu_2nd, std_2nd)
p_3rd = norm.pdf(x, mu_3rd, std_3rd)

plt.figure(figsize=(10,6))

# Plot the normal distribution
plt.plot(x, p_ANN, 'k', linewidth=2, color=color_orange, label='ANN')
plt.plot(x, p_linear, 'k', linewidth=2, color=color_grey, label='Linear')
plt.plot(x, p_2nd, 'k', linewidth=2, color=color_darkblue,label='2nd Order')
plt.plot(x, p_3rd, 'k', linewidth=2, color=color_green, label='3rd Order')

plt.axvline(0, color='black', linestyle='dotted')
plt.axvline(15, color=color_grey, linestyle='dotted')
plt.axvline(-15, color=color_grey, linestyle='dotted')

# Add labels and title
plt.xlabel('Relative Error [%]')
plt.ylabel('Probability Density [-]')
plt.title('Distribution of Relative Error of Outrigger System (S2)')
plt.legend()

plt.xticks(np.arange(min(x)+1, max(x)+1, 25))

# Show the plot
plt.show()

# Compute RMSE for ANN
rmse_ANN = np.sqrt(np.mean((y_test_array2 - predictions_test_S2_ANN)**2))

# Compute RMSE for Linear
rmse_linear = np.sqrt(np.mean((y_test_array2 - predictions_test_S2_linear)**2))
```

```
# Compute RMSE for 2nd Order
rmse_2nd = np.sqrt(np.mean((y_test_array2 - predictions_test_S2_2nd)**2))

# Compute RMSE for 3rd Order
rmse_3rd = np.sqrt(np.mean((y_test_array2 - predictions_test_S2_3rd)**2))

print("The RMSE of ANN is equal to", round(rmse_ANN, 3))
print("The RMSE of Linear is equal to", round(rmse_linear, 3))
print("The RMSE of 2nd Order is equal to", round(rmse_2nd, 3))
print("The RMSE of 3rd Order is equal to", round(rmse_3rd, 3))

print(mu_linear)
print(mu_test)
```

In [ ]:
```
print("The mean (mu) of ANN is equal to", round(mu_ANN, 2))
print("The mean (mu) of Linear is equal to", round(mu_linear, 2))
print("The mean (mu) of 2nd Order is equal to", round(mu_2nd, 2))
print("The mean (mu) of 3rd Order is equal to", round(mu_3rd, 2))
print("")
print("The standard deviation (sigma) of ANN is equal to", round(std_ANN, 1))
print("The standard deviation (sigma) of Linear is equal to", round(std_linear, 1))
print("The standard deviation (sigma) of 2nd Order is equal to", round(std_2nd, 1))
print("The standard deviation (sigma) of 3rd Order is equal to", round(std_3rd, 1))
```

Total Probability

In [ ]:
```
def integrate_pdf_within_range(pdf_function, lower_bound, upper_bound, *args):
    integral, _ = quad(pdf_function, lower_bound, upper_bound, args=args)
    return integral

# Define the range (-15% to +15%)
lower_bound = -15
upper_bound = 15

# Calculate the total probability within the specified range for each model
total_probability_ANN = integrate_pdf_within_range(norm.pdf, lower_bound, upper_bound, mu_ANN, std_ANN)
total_probability_linear = integrate_pdf_within_range(norm.pdf, lower_bound, upper_bound, mu_linear, std_linear)
total_probability_2nd = integrate_pdf_within_range(norm.pdf, lower_bound, upper_bound, mu_2nd, std_2nd)
total_probability_3rd = integrate_pdf_within_range(norm.pdf, lower_bound, upper_bound, mu_3rd, std_3rd)

print("Total Probability within -15% to +15% for ANN:", total_probability_ANN)
print("Total Probability within -15% to +15% for Linear:", total_probability_linear)
print("Total Probability within -15% to +15% for 2nd Order:", total_probability_2nd)
print("Total Probability within -15% to +15% for 3rd Order:", total_probability_3rd)
```

Determination of MAPE

In [ ]:
```
# Compute MAPE for ANN
mape_ANN = np.mean(np.abs(diff_percentage_ANN))
print("The MAPE of ANN is equal to", round(mape_ANN, 3))

# Compute MAPE for Linear
mape_linear = np.mean(np.abs(diff_percentage_linear))
print("The MAPE of Linear is equal to", round(mape_linear, 3))

# Compute MAPE for 2nd Order
mape_2nd = np.mean(np.abs(diff_percentage_2nd))
print("The MAPE of 2nd Order is equal to", round(mape_2nd, 3))

# Compute MAPE for 3rd Order
mape_3rd = np.mean(np.abs(diff_percentage_3rd))
print("The MAPE of 3rd Order is equal to", round(mape_3rd, 3))
```

RMSE for different height ranges

S1: Braced Framed Tube

Different height ranges

In [ ]:
```
# Function to calculate RMSE for different height ranges
height_ranges = [(50, 100), (100, 150), (150, 200), (200, 250), (250,300)]  # Define the height ranges
rmse_values_S1_diffheight = []

for height_range in height_ranges:
    indices_train = X_train[(X_train['in:Height [m]'] >= height_range[0]) & (X_train['in:Height [m]'] < height_range[1])].index
    train_values = y_train.loc[indices_train]
    train_X = X_train.loc[indices_train]

    indices_test = X_test[(X_test['in:Height [m]'] >= height_range[0]) & (X_test['in:Height [m]'] < height_range[1])].index
    test_values = y_test.loc[indices_test]
    test_X = X_test.loc[indices_test]

    test_X_scaled = scaler1.transform(test_X)
    predictions_values = best_model_S1.predict(test_X_scaled)

    rmse = np.sqrt(mean_squared_error(test_values, predictions_values))
    rmse_values_S1_diffheight.append(rmse)


rmse_values_S1_diffheight = [rmse * s_price for rmse in rmse_values_S1_diffheight]

# Print the RMSE values for different height ranges
for i, height_range in enumerate(height_ranges):
    print(f"RMSE for the range of height: {height_range[0]}-{height_range[1]} for S1: {round(rmse_values_S1_diffheight[i],1)}" f" [€/GFA]"
```

Different width ranges

In [ ]:
```
width_ranges = [(10, 20), (20, 30), (30, 40), (40, 50), (50,60)]  # Define the width ranges
rmse_values_S1_diffwidth = []

for width_range in width_ranges:
    indices_train = X_train[(X_train['in:Length [m]'] >= width_range[0]) & (X_train['in:Length [m]'] < width_range[1])].index
    train_values = y_train.loc[indices_train]
    train_X = X_train.loc[indices_train]

    indices_test = X_test[(X_test['in:Length [m]'] >= width_range[0]) & (X_test['in:Length [m]'] < width_range[1])].index
    test_values = y_test.loc[indices_test]
    test_X = X_test.loc[indices_test]

    test_X_scaled = scaler1.transform(test_X)
    predictions_values = best_model_S1.predict(test_X_scaled)

    rmse = np.sqrt(mean_squared_error(test_values, predictions_values))
    rmse_values_S1_diffwidth.append(rmse)

rmse_values_S1_diffwidth = [rmse * s_price for rmse in rmse_values_S1_diffwidth]

# Print the RMSE values for different width ranges
for i, width_range in enumerate(width_ranges):
    print(f"RMSE for the range of width: {width_range[0]}-{width_range[1]} for S1: {round(rmse_values_S1_diffwidth[i],1)}" f" [€/GFA]")
```

S2: Outrigger

Different height ranges

In [ ]:
```
rmse_values_S2_diffheight_steel = []
rmse_values_S2_diffheight_concrete = []
mape_values_S2_diffheight = []

for height_range in height_ranges:
    indices_train = X_train2[(X_train['in:Height [m]'] >= height_range[0]) & (X_train2['in:Height [m]'] < height_range[1])].index
    train_values = y_train2.loc[indices_train]
    train_X = X_train2.loc[indices_train]

    indices_test = X_test2[(X_test2['in:Height [m]'] >= height_range[0]) & (X_test2['in:Height [m]'] < height_range[1])].index
    test_values = y_test2.loc[indices_test]
    test_X = X_test2.loc[indices_test]

    test_X_scaled = scaler2.transform(test_X)
    predictions_values = best_model_S2.predict(test_X_scaled)

    predictions_values_S2_1 = predictions_values[:, 0]
    predictions_values_S2_2 = predictions_values[:, 1]
    total_predictions_S2 = predictions_values_S2_1 + predictions_values_S2_2

    # Separate y_true for each output of S2
    test_values_S2_1 = test_values.iloc[:, 0]
    test_values_S2_2 = test_values.iloc[:, 1]
    total_test_S2 = test_values_S2_1 + test_values_S2_2

    diff_test_S2 = np.abs(total_test_S2 - total_predictions_S2)
    diff_percentage_S2 = (diff_test_S2 / total_test_S2) * 100
    mape_S2 = np.mean(np.abs(diff_percentage_S2))

    mape_values_S2_diffheight.append(mape_S2)

    # Calculate RMSE for each output of S2
    rmse_S2_1 = np.sqrt(mean_squared_error(test_values_S2_1, predictions_values_S2_1))
    rmse_S2_2 = np.sqrt(mean_squared_error(test_values_S2_2, predictions_values_S2_2))

    rmse_values_S2_diffheight_steel.append(rmse_S2_1)
    rmse_values_S2_diffheight_concrete.append(rmse_S2_2)

    rmse = np.sqrt(mean_squared_error(test_values, predictions_values))
    rmse_values_S1_diffheight.append(rmse)

rmse_values_S1_diffheight_steel = [rmse * s_price for rmse in rmse_values_S2_diffheight_steel]
rmse_values_S1_diffheight_concrete = [rmse * pc_rc_price for rmse in rmse_values_S2_diffheight_concrete]

mape_values_S1_diffheight = [mape for mape in mape_values_S2_diffheight]

rmse_values_S2_diffheight = [a + b for a, b in zip(rmse_values_S2_diffheight_steel, rmse_values_S2_diffheight_concrete)]

# Print the RMSE values for different height ranges for the first output of S2
for i, height_range in enumerate(height_ranges):
    print(f"RMSE for the range of height: {height_range[0]}-{height_range[1]} (S2: Steel & Output Concrete): "
          f"{round(rmse_values_S2_diffheight_steel[i],1)} and {round(rmse_values_S2_diffheight_concrete[i],1)}"  f" [€/GFA]")

print("")
for i, height_range in enumerate(height_ranges):
    print(f"RMSE for the range of heigt: {height_range[0]}-{height_range[1]} for S2: "
          f"{round(rmse_values_S2_diffheight[i],1)}"  f" [€/GFA]")

print("")
for i, height_range in enumerate(height_ranges):
    print(f"MAPE for the range of heigt: {height_range[0]}-{height_range[1]} for S2: "
          f"{round(mape_values_S2_diffheight[i],1)}"  f" [%]")
```

Different width ranges

In [ ]:
```
rmse_values_S2_diffwidth_steel = []
rmse_values_S2_diffwidth_concrete = []

for width_range in width_ranges:
    indices_train = X_train2[(X_train2['in:Length [m]'] >= width_range[0]) & (X_train2['in:Length [m]'] < width_range[1])].index
    train_values = y_train2.loc[indices_train]
    train_X = X_train2.loc[indices_train]

    indices_test = X_test2[(X_test2['in:Length [m]'] >= width_range[0]) & (X_test2['in:Length [m]'] < width_range[1])].index
    test_values = y_test2.loc[indices_test]
    test_X = X_test2.loc[indices_test]

    test_X_scaled = scaler2.transform(test_X)
    predictions_values = best_model_S2.predict(test_X_scaled)

    predictions_values_S2_1 = predictions_values[:, 0]
    predictions_values_S2_2 = predictions_values[:, 1]

    # Separate y_true for each output of S2
    test_values_S2_1 = test_values.iloc[:, 0]
    test_values_S2_2 = test_values.iloc[:, 1]

    # Calculate RMSE for each output of S2
    rmse_S2_1 = np.sqrt(mean_squared_error(test_values_S2_1, predictions_values_S2_1))
    rmse_S2_2 = np.sqrt(mean_squared_error(test_values_S2_2, predictions_values_S2_2))

    rmse_values_S2_diffwidth_steel.append(rmse_S2_1)
    rmse_values_S2_diffwidth_concrete.append(rmse_S2_2)

rmse_values_S2_diffwidth_steel = [rmse * s_price for rmse in rmse_values_S2_diffwidth_steel]
rmse_values_S2_diffwidth_concrete = [rmse * pc_rc_price for rmse in rmse_values_S2_diffwidth_concrete]

rmse_values_S2_diffwidth = [a + b for a, b in zip(rmse_values_S2_diffwidth_steel, rmse_values_S2_diffwidth_concrete)]

# Print the RMSE values for different height ranges for the first output of S2
for i, width_range in enumerate(width_ranges):
    print(f"RMSE for the range of width: {width_range[0]}-{width_range[1]} (S2: Steel & Output Concrete): "
          f"{round(rmse_values_S2_diffwidth_steel[i],1)} and {round(rmse_values_S2_diffwidth_concrete[i],1)}"  f" [€/GFA]")

print("")
for i, width_range in enumerate(width_ranges):
    print(f"RMSE for the range of width: {width_range[0]}-{width_range[1]} for S2:: "
          f"{round(rmse_values_S2_diffwidth[i],1)}"  f" [€/GFA]")
```

5) VISUALIZATION WITH UPPER AND LOWER BAND OF RMSE

In [ ]:
```
# Define constant variables
constant_options = ['Width', 'Height']

# Create interactive sliders and dropdown for constant variables
constant_variable_dropdown = widgets.Dropdown(options=constant_options, value='Width', description='Constant Parameter:')
constant_value_input = widgets.FloatText(value=20, description='Constant Parameter:')
variable_slider = widgets.FloatSlider(value=20, min=5, max=500, step=1, description='Variable Parameter:')


# Update descriptions based on the constant variable
def update_slider_descriptions(constant_variable):
    if constant_variable == 'Height':
        constant_value_input.description = 'Constant Parameter = Height'
        variable_slider.description = 'Variable Parameter = Width'
        variable_slider.min = 50
        variable_slider.max = 300
    else:
        constant_value_input.description = 'Constant Parameter = Width'
        variable_slider.description = 'Variable Parameter = Height'
        variable_slider.min = 50
        variable_slider.max = 300

# Set width for the sliders and dropdown
constant_variable_dropdown.layout.width = '40%'
constant_value_input.layout.width = '40%'
variable_slider.layout.width = '40%'
constant_variable_dropdown.style = {'description_width': '50%'}
constant_value_input.style = {'description_width': '50%'}
variable_slider.style = {'description_width': '50%'}

# Observe changes in the dropdown and update slider descriptions accordingly
widgets.interactive(update_slider_descriptions, constant_variable=constant_variable_dropdown)

# Function to make predictions and plot the graph
def make_predictions_and_plot_S1(constant_variable, constant_value, variable_range):
    # Generate a range of variable values within the specified range
    variable_values = np.linspace(variable_slider.min, variable_slider.max, 100).reshape(-1, 1)

    # Prepare input data
    if constant_variable == 'Width':
        input_data = pd.DataFrame({
            'in:Length [m]': constant_value,
            'in:Height [m]': variable_values.flatten()
        })

    else:
        input_data = pd.DataFrame({
            'in:Length [m]': variable_values.flatten(),
            'in:Height [m]': constant_value
        })

    input_data1 = scaler1.transform(input_data)
    input_data2 = scaler2.transform(input_data)

    # Convert the predictons from mass to costs and embodied carbon
    # Costs
    predictions_S1 = best_model_S1.predict(input_data1)
    predictions_S1 *= s_price
    p_total_costs_S1 = predictions_S1

    predictions_S1 = best_model_S1.predict(input_data1)
    predictions_S1 *= s_EmCO2*CO2_sprice
    p_total_EmCO2_S1 = predictions_S1

    predictions_S2 = best_model_S2.predict(input_data2)
    predictions_S2[:, 0] *= s_price
    predictions_S2[:, 1] *= pc_rc_price
    p_total_costs_S2 = predictions_S2.sum(axis=1)

    predictions_S2 = best_model_S2.predict(input_data2)
    predictions_S2[:, 0] *= s_EmCO2*CO2_sprice
    predictions_S2[:, 1] *= pc_rc_EmCO2*CO2_sprice
    p_total_EmCO2_S2 = predictions_S2.sum(axis=1)

    # Upper and lower bands based on RMSE for each system

    rmse_values_width_S1 = rmse_values_S1_diffwidth
    rmse_values_height_S1 = rmse_values_S1_diffheight

    rmse_values_width_S2 = rmse_values_S2_diffwidth
    rmse_values_height_S2 = rmse_values_S2_diffheight

    height_ranges = [(50, 100), (100, 150), (150, 200), (200, 250), (250,300)]
    width_ranges = [(10, 20), (20, 30), (30, 40), (40, 50), (50,60)]

    # Choose the appropriate RMSE values based on the selected constant variable
    if constant_variable == 'Width':
        rmse_values_S1 = rmse_values_height_S1
        rmse_values_S2 = rmse_values_height_S2

        all_predictions_S1 = []
        all_predictions_S2 = []
        all_range_of_input = []

        for height_range in height_ranges:
            indices = input_data[(input_data['in:Height [m]'] >= height_range[0]) & (input_data['in:Height [m]'] < height_range[1])].index
            range_of_input = input_data.loc[indices]
            range_of_input_scaled = scaler1.transform(range_of_input)
            predictions_S1 = best_model_S1.predict(range_of_input_scaled)
            predictions_S1 *= s_price
            all_predictions_S1.append(predictions_S1)
```

```
        range_of_input_scaled2 = scaler2.transform(range_of_input)
        predictions_S2 = best_model_S2.predict(range_of_input_scaled2)
        predictions_S2[:, 0] *= s_price
        predictions_S2[:, 1] *= pc_rc_price
        all_predictions_S2.append(predictions_S2.sum(axis=1))

        all_range_of_input.append(range_of_input)

    all_predictions_upper_S1 = np.concatenate([arr + 0.5*rmse for arr, rmse in zip(all_predictions_S1, rmse_values_height_S1)])
    all_predictions_lower_S1 = np.concatenate([arr - 0.5*rmse for arr, rmse in zip(all_predictions_S1, rmse_values_height_S1)])

    all_predictions_upper_S2 = np.concatenate([arr + 0.5*rmse for arr, rmse in zip(all_predictions_S2, rmse_values_height_S2)])
    all_predictions_lower_S2 = np.concatenate([arr - 0.5*rmse for arr, rmse in zip(all_predictions_S2, rmse_values_height_S2)])

    all_heights = np.concatenate([df['in:Height [m]'].values.flatten() for df in all_range_of_input])


else:
    rmse_values_S1 = rmse_values_width_S1
    rmse_values_S2 = rmse_values_width_S2

    all_predictions_S1 = []
    all_predictions_S2 = []
    all_range_of_input = []

    for width_range in width_ranges:
        indices = input_data[(input_data['in:Length [m]'] >= width_range[0]) & (input_data['in:Length [m]'] < width_range[1])].index
        range_of_input = input_data.loc[indices]
        range_of_input_scaled = scaler1.transform(range_of_input)
        predictions_S1 = best_model_S1.predict(range_of_input_scaled)
        predictions_S1 *= s_price
        all_predictions_S1.append(predictions_S1)

        range_of_input_scaled2 = scaler2.transform(range_of_input)
        predictions_S2 = best_model_S2.predict(range_of_input_scaled2)
        predictions_S2[:, 0] *= s_price
        predictions_S2[:, 1] *= pc_rc_price
        all_predictions_S2.append(predictions_S2.sum(axis=1))

        all_range_of_input.append(range_of_input)

    all_predictions_upper_S1 = np.concatenate([arr + 0.5*rmse for arr, rmse in zip(all_predictions_S1, rmse_values_width_S1)])
    all_predictions_lower_S1 = np.concatenate([arr - 0.5*rmse for arr, rmse in zip(all_predictions_S1, rmse_values_width_S1)])

    all_predictions_upper_S2 = np.concatenate([arr + 0.5*rmse for arr, rmse in zip(all_predictions_S2, rmse_values_width_S2)])
    all_predictions_lower_S2 = np.concatenate([arr - 0.5*rmse for arr, rmse in zip(all_predictions_S2, rmse_values_width_S2)])

    all_widths = np.concatenate([df['in:Length [m]'].values.flatten() for df in all_range_of_input])


if constant_variable == 'Width':
    constant_value_df = pd.DataFrame({'in:Length [m]': [constant_value]})
    constant_value_df.columns = ['in:Length [m]']
    variable_range_df = pd.DataFrame({'in:Height [m]': [variable_range]})
    variable_range_df.columns = ['in:Height [m]']

    input_data_dot = pd.DataFrame({
        'in:Length [m]': constant_value_df['in:Length [m]'],
        'in:Height [m]': variable_range_df['in:Height [m]']
   })

else:
    constant_value_df = pd.DataFrame({'in:Height [m]': [constant_value]})
    constant_value_df.columns = ['in:Height [m]']
    variable_range_df = pd.DataFrame({'in:Length [m]': [variable_range]})
    variable_range_df.columns = ['in:Length [m]']

    input_data_dot = pd.DataFrame({
        'in:Length [m]': variable_range_df['in:Length [m]'],
        'in:Height [m]': constant_value_df['in:Height [m]']
    })


# Extract the relevant columns for plotting
if constant_variable == 'Width':
    X_filtered = input_data['in:Height [m]']
else:
    X_filtered = input_data['in:Length [m]']

input_data_dot1 = scaler1.transform(input_data_dot)
input_data_dot2 = scaler2.transform(input_data_dot)

predictions_dot_S1 = best_model_S1.predict(input_data_dot1)
predictions_dot_S1 *= s_price
p_total_costs_S1_dot = predictions_dot_S1

predictions_dot_S2 = best_model_S2.predict(input_data_dot2)
predictions_dot_S2[:, 0] *= s_price
predictions_dot_S2[:, 1] *= pc_rc_price
p_total_costs_S2_dot = predictions_dot_S2.sum(axis=1)

# Create subplots
plt.figure(figsize=(12, 12))


# Plot for Total Costs per GFA
plt.subplot(2, 1, 1)

#plt.xlim(0, max(X_filtered) + 20)
#plt.ylim(0, max(p_total_costs_S1) + 50)

plt.plot(X_filtered, p_total_costs_S1, color=color_green, label=f'S1: Braced Framed Tube')
```

```
        plt.plot(X_filtered, p_total_costs_S2, color=color_darkblue, label=f'S2: Outrigger System')

        plt.scatter([variable_range], np.round(p_total_costs_S1_dot, 2), color=color_lightgreen, marker='o',
                    label=f'S1: Predicted Costs for chosen value = {np.round(p_total_costs_S1_dot[0], 2)}')

        plt.scatter([variable_range], np.round(p_total_costs_S2_dot, 2), color=color_lightblue, marker='o',
                    label=f'S2: Predicted Costs for chosen value = {np.round(p_total_costs_S2_dot[0], 2)}')

        # Plot upper and lower bands
        if constant_variable == 'Width':
            plt.plot(all_heights, all_predictions_upper_S1, '--', color=color_green, label='Upper Band S1',alpha = 0.2)
            plt.plot(all_heights, all_predictions_lower_S1, '--', color=color_green, label='Lower Band S1', alpha = 0.2)
            plt.plot(all_heights, all_predictions_upper_S2, '--', color=color_darkblue, label='Upper Band S2',alpha = 0.2)
            plt.plot(all_heights, all_predictions_lower_S2, '--', color=color_darkblue, label='Lower Band S2', alpha = 0.2)

        else:
            plt.plot(all_widths, all_predictions_upper_S1, '--', color=color_green, label='Upper Band S1',alpha = 0.2)
            plt.plot(all_widths, all_predictions_lower_S1, '--', color=color_green, label='Lower Band S1', alpha = 0.2)
            plt.plot(all_widths, all_predictions_upper_S2, '--', color=color_darkblue, label='Upper Band S2',alpha = 0.2)
            plt.plot(all_widths, all_predictions_lower_S2, '--', color=color_darkblue, label='Lower Band S2', alpha = 0.2)

        plt.title(f'Total $\mathbf{{Structural}}$ $\mathbf{{Costs}}$ [€/m2GFA] for {constant_variable}={constant_value}')

        if constant_variable == 'Width':
            plt.xlabel('Height [m]')

        else:
            plt.xlabel('Width [m]')

        plt.ylabel('Total Structural Costs [€/m2GFA]')
        plt.legend(bbox_to_anchor=(1.02, 1), loc='upper left', borderaxespad=0)


        # Adjust layout
        plt.tight_layout()
        plt.ylim(-500, 7000)
        if constant_variable == 'Width':
            plt.xlim(0,320)
        else:
            plt.xlim(0,65)

        plt.show()

# Create an interactive output widget
output = widgets.interactive_output(make_predictions_and_plot_S1, {
    'constant_variable': constant_variable_dropdown,
    'constant_value': constant_value_input,
    'variable_range': variable_slider
})


# Create an interactive layout with sliders and output
interactive_layout = widgets.VBox([
    constant_variable_dropdown,
    constant_value_input,
    variable_slider,
    output
])

# Display the interactive layout
display(interactive_layout)
```

In [ ]:
```
# Define constant variables
constant_options = ['Width', 'Height']

# Create interactive sliders and dropdown for constant variables
constant_variable_dropdown = widgets.Dropdown(options=constant_options, value='Width', description='Constant Parameter:')
constant_value_input = widgets.FloatText(value=20, description='Constant Parameter:')
variable_slider = widgets.FloatSlider(value=20, min=5, max=500, step=1, description='Variable Parameter:')


# Update descriptions based on the constant variable
def update_slider_descriptions(constant_variable):
    if constant_variable == 'Height':
        constant_value_input.description = 'Constant Parameter = Height'
        variable_slider.description = 'Variable Parameter = Width'
        variable_slider.min = 10
        variable_slider.max = 300
    else:
        constant_value_input.description = 'Constant Parameter = Width'
        variable_slider.description = 'Variable Parameter = Height'
        variable_slider.min = 50
        variable_slider.max = 300

# Set width for the sliders and dropdown
constant_variable_dropdown.layout.width = '40%'
constant_value_input.layout.width = '40%'
variable_slider.layout.width = '40%'
constant_variable_dropdown.style = {'description_width': '50%'}
constant_value_input.style = {'description_width': '50%'}
variable_slider.style = {'description_width': '50%'}

# Observe changes in the dropdown and update slider descriptions accordingly
widgets.interactive(update_slider_descriptions, constant_variable=constant_variable_dropdown)

# Function to make predictions and plot the graph
def make_predictions_and_plot_S1(constant_variable, constant_value, variable_range):
    # Generate a range of variable values within the specified range
    variable_values = np.linspace(variable_slider.min, variable_slider.max, 100).reshape(-1, 1)

    # Prepare input data
    if constant_variable == 'Width':
        input_data = pd.DataFrame({
            'in:Length [m]': constant_value,
            'in:Height [m]': variable_values.flatten()
        })

    else:
        input_data = pd.DataFrame({
            'in:Length [m]': variable_values.flatten(),
            'in:Height [m]': constant_value
        })

    input_data1 = scaler1.transform(input_data)
    input_data2 = scaler2.transform(input_data)

    # Convert the predictons from mass to costs and embodied carbon
    # Costs
    predictions_S1 = best_model_S1.predict(input_data1)
    predictions_S1 *= s_price
    p_total_costs_S1 = predictions_S1

    predictions_S1 = best_model_S1.predict(input_data1)
    predictions_S1 *= s_EmCO2*CO2_sprice
    p_total_EmCO2_S1 = predictions_S1

    predictions_S2 = best_model_S2.predict(input_data2)
    predictions_S2[:, 0] *= s_price
    predictions_S2[:, 1] *= pc_rc_price
    p_total_costs_S2 = predictions_S2.sum(axis=1)

    predictions_S2 = best_model_S2.predict(input_data2)
    predictions_S2[:, 0] *= s_EmCO2*CO2_sprice
    predictions_S2[:, 1] *= pc_rc_EmCO2*CO2_sprice
    p_total_EmCO2_S2 = predictions_S2.sum(axis=1)

    # Upper and lower bands based on RMSE for each system

    rmse_values_width_S1 = rmse_values_S1_diffwidth
    rmse_values_height_S1 = rmse_values_S1_diffheight

    rmse_values_width_S2 = rmse_values_S2_diffwidth
    rmse_values_height_S2 = rmse_values_S2_diffheight

    height_ranges = [(50, 100), (100, 150), (150, 200), (200, 250), (250,300)]
    width_ranges = [(10, 20), (20, 30), (30, 40), (40, 50), (50,60)]

    # Choose the appropriate RMSE values based on the selected constant variable
    if constant_variable == 'Width':
        rmse_values_S1 = rmse_values_height_S1
        rmse_values_S2 = rmse_values_height_S2

        all_predictions_S1 = []
        all_predictions_S2 = []
        all_range_of_input = []

        for height_range in height_ranges:
            indices = input_data[(input_data['in:Height [m]'] >= height_range[0]) & (input_data['in:Height [m]'] < height_range[1])].index
            range_of_input = input_data.loc[indices]
            range_of_input_scaled = scaler1.transform(range_of_input)
            predictions_S1 = best_model_S1.predict(range_of_input_scaled)
            predictions_S1 *= s_price
            all_predictions_S1.append(predictions_S1)
```

```
        range_of_input_scaled2 = scaler2.transform(range_of_input)
        predictions_S2 = best_model_S2.predict(range_of_input_scaled2)
        predictions_S2[:, 0] *= s_price
        predictions_S2[:, 1] *= pc_rc_price
        all_predictions_S2.append(predictions_S2.sum(axis=1))

        all_range_of_input.append(range_of_input)


else:
    rmse_values_S1 = rmse_values_width_S1
    rmse_values_S2 = rmse_values_width_S2

    all_predictions_S1 = []
    all_predictions_S2 = []
    all_range_of_input = []

    for width_range in width_ranges:
        indices = input_data[(input_data['in:Length [m]'] >= width_range[0]) & (input_data['in:Length [m]'] < width_range[1])].index
        range_of_input = input_data.loc[indices]
        range_of_input_scaled = scaler1.transform(range_of_input)
        predictions_S1 = best_model_S1.predict(range_of_input_scaled)
        predictions_S1 *= s_price
        all_predictions_S1.append(predictions_S1)

        range_of_input_scaled2 = scaler2.transform(range_of_input)
        predictions_S2 = best_model_S2.predict(range_of_input_scaled2)
        predictions_S2[:, 0] *= s_price
        predictions_S2[:, 1] *= pc_rc_price
        all_predictions_S2.append(predictions_S2.sum(axis=1))

        all_range_of_input.append(range_of_input)


    all_widths = np.concatenate([df['in:Length [m]'].values.flatten() for df in all_range_of_input])


if constant_variable == 'Width':
    constant_value_df = pd.DataFrame({'in:Length [m]': [constant_value]})
    constant_value_df.columns = ['in:Length [m]']
    variable_range_df = pd.DataFrame({'in:Height [m]': [variable_range]})
    variable_range_df.columns = ['in:Height [m]']

    input_data_dot = pd.DataFrame({
        'in:Length [m]': constant_value_df['in:Length [m]'],
        'in:Height [m]': variable_range_df['in:Height [m]']
   })

else:
    constant_value_df = pd.DataFrame({'in:Height [m]': [constant_value]})
    constant_value_df.columns = ['in:Height [m]']
    variable_range_df = pd.DataFrame({'in:Length [m]': [variable_range]})
    variable_range_df.columns = ['in:Length [m]']

    input_data_dot = pd.DataFrame({
        'in:Length [m]': variable_range_df['in:Length [m]'],
        'in:Height [m]': constant_value_df['in:Height [m]']
    })


# Extract the relevant columns for plotting
if constant_variable == 'Width':
    X_filtered = input_data['in:Height [m]']
else:
    X_filtered = input_data['in:Length [m]']

input_data_dot1 = scaler1.transform(input_data_dot)
input_data_dot2 = scaler2.transform(input_data_dot)

predictions_dot_S1 = best_model_S1.predict(input_data_dot1)
predictions_dot_S1 *= s_price
p_total_costs_S1_dot = predictions_dot_S1

predictions_dot_S2 = best_model_S2.predict(input_data_dot2)
predictions_dot_S2[:, 0] *= s_price
predictions_dot_S2[:, 1] *= pc_rc_price
p_total_costs_S2_dot = predictions_dot_S2.sum(axis=1)

# Create subplots
plt.figure(figsize=(12, 12))


# Plot for Total Costs per GFA
plt.subplot(2, 1, 1)

#plt.xlim(0, max(X_filtered) + 20)
#plt.ylim(0, max(p_total_costs_S1) + 50)

plt.plot(X_filtered, p_total_costs_S1, color=color_green, label=f'S1: Braced Framed Tube')
plt.plot(X_filtered, p_total_costs_S2, color=color_darkblue, label=f'S2: Outrigger System')

plt.scatter([variable_range], np.round(p_total_costs_S1_dot, 2), color=color_lightgreen, marker='o',
            label=f'S1: Predicted Costs for chosen value = {np.round(p_total_costs_S1_dot[0], 2)}')

plt.scatter([variable_range], np.round(p_total_costs_S2_dot, 2), color=color_lightblue, marker='o',
            label=f'S2: Predicted Costs for chosen value = {np.round(p_total_costs_S2_dot[0], 2)}')


plt.title(f'Total $\mathbf{{Structural}}$ $\mathbf{{Costs}}$ [€/m2GFA] for {constant_variable}={constant_value}')

if constant_variable == 'Width':
    plt.xlabel('Height [m]')
```

```
        else:
            plt.xlabel('Width [m]')

        plt.ylabel('Total Structural Costs [€/m2GFA]')
        plt.legend(bbox_to_anchor=(1.02, 1), loc='upper left', borderaxespad=0)


        # Adjust layout
        plt.tight_layout()
        plt.ylim(-500, 5000)
        if constant_variable == 'Width':
            plt.xlim(0,320)
        else:
            plt.xlim(0,65)

        plt.show()

    # Create an interactive output widget
    output = widgets.interactive_output(make_predictions_and_plot_S1, {
        'constant_variable': constant_variable_dropdown,
        'constant_value': constant_value_input,
        'variable_range': variable_slider
    })


    # Create an interactive layout with sliders and output
    interactive_layout = widgets.VBox([
        constant_variable_dropdown,
        constant_value_input,
        variable_slider,
        output
    ])

    # Display the interactive layout
    display(interactive_layout)
```

6) COMPARISON WITH INPUT DATA

Comparision of S1

In [ ]:
```
constant_variable = 'in:Length [m]'
constant_value = (28.8) # Set the constant value you want to use

# Filter data for the specified constant value
filtered_data_S1 = mass_S1[mass_S1[constant_variable] == constant_value]

# Extract the relevant columns
X_filtered_S1 = filtered_data_S1['in:Height [m]']
y_costs_S1 = filtered_data_S1['Mass Total Steel'] * s_price
```

In [ ]:
```
# Define constant variables
constant_options = ['Width', 'Height']

# Create interactive sliders and dropdown for constant variables
constant_variable_dropdown = widgets.Dropdown(options=constant_options, value='Width', description='Constant Parameter:')
constant_value_input = widgets.FloatText(value=20, description='Constant Parameter:')
variable_slider = widgets.FloatSlider(value=20, min=5, max=500, step=1, description='Variable Parameter:')


# Update descriptions based on the constant variable
def update_slider_descriptions(constant_variable):
    if constant_variable == 'Height':
        constant_value_input.description = 'Constant Parameter = Height'
        variable_slider.description = 'Variable Parameter = Width'
        variable_slider.min = 10
        variable_slider.max = 60
    else:
        constant_value_input.description = 'Constant Parameter = Width'
        variable_slider.description = 'Variable Parameter = Height'
        variable_slider.min = 50
        variable_slider.max = 300

# Set width for the sliders and dropdown
constant_variable_dropdown.layout.width = '40%'
constant_value_input.layout.width = '40%'
variable_slider.layout.width = '40%'
constant_variable_dropdown.style = {'description_width': '50%'}
constant_value_input.style = {'description_width': '50%'}
variable_slider.style = {'description_width': '50%'}

# Observe changes in the dropdown and update slider descriptions accordingly
widgets.interactive(update_slider_descriptions, constant_variable=constant_variable_dropdown)

# Function to make predictions and plot the graph
def make_predictions_and_plot_S1(constant_variable, constant_value, variable_range):
    # Generate a range of variable values within the specified range
    variable_values = np.linspace(variable_slider.min, variable_slider.max, 100).reshape(-1, 1)

    # Prepare input data
    if constant_variable == 'Width':
        input_data = pd.DataFrame({
            'in:Length [m]': constant_value,
            'in:Height [m]': variable_values.flatten()
        })

    else:
        input_data = pd.DataFrame({
            'in:Length [m]': variable_values.flatten(),
            'in:Height [m]': constant_value
        })

    input_data1 = scaler1.transform(input_data)
    input_data2 = scaler2.transform(input_data)

    # Convert the predictons from mass to costs and embodied carbon
    # Costs
    predictions_S1 = best_model_S1.predict(input_data1)
    predictions_S1 *= s_price
    p_total_costs_S1 = predictions_S1

    predictions_S1 = best_model_S1.predict(input_data1)
    predictions_S1 *= s_EmCO2*CO2_sprice
    p_total_EmCO2_S1 = predictions_S1

    # Upper and lower bands based on RMSE for each system

    rmse_values_width_S1 = rmse_values_S1_diffwidth
    rmse_values_height_S1 = rmse_values_S1_diffheight

    height_ranges = [(50, 100), (100, 150), (150, 200), (200, 250), (250,300)]
    width_ranges = [(10, 20), (20, 30), (30, 40), (40, 50), (50,60)]

    # Choose the appropriate RMSE values based on the selected constant variable
    if constant_variable == 'Width':
        rmse_values_S1 = rmse_values_height_S1

        all_predictions_S1 = []
        all_range_of_input = []

        for height_range in height_ranges:
            indices = input_data[(input_data['in:Height [m]'] >= height_range[0]) & (input_data['in:Height [m]'] < height_range[1])].index
            range_of_input = input_data.loc[indices]
            range_of_input_scaled = scaler1.transform(range_of_input)
            predictions_S1 = best_model_S1.predict(range_of_input_scaled)
            predictions_S1 *= s_price
            all_predictions_S1.append(predictions_S1)


            all_range_of_input.append(range_of_input)

        all_predictions_upper_S1 = np.concatenate([arr + rmse for arr, rmse in zip(all_predictions_S1, rmse_values_height_S1)])
        all_predictions_lower_S1 = np.concatenate([arr - rmse for arr, rmse in zip(all_predictions_S1, rmse_values_height_S1)])
        all_heights = np.concatenate([df['in:Height [m]'].values.flatten() for df in all_range_of_input])


    else:
        rmse_values_S1 = rmse_values_width_S1

        all_predictions_S1 = []
        all_range_of_input = []

        for width_range in width_ranges:
```

```
            indices = input_data[(input_data['in:Length [m]'] >= width_range[0]) & (input_data['in:Length [m]'] < width_range[1])].index
            range_of_input = input_data.loc[indices]
            range_of_input_scaled = scaler1.transform(range_of_input)
            predictions_S1 = best_model_S1.predict(range_of_input_scaled)
            predictions_S1 *= s_price
            all_predictions_S1.append(predictions_S1)

            all_range_of_input.append(range_of_input)

        all_predictions_upper_S1 = np.concatenate([arr + rmse for arr, rmse in zip(all_predictions_S1, rmse_values_width_S1)])
        all_predictions_lower_S1 = np.concatenate([arr - rmse for arr, rmse in zip(all_predictions_S1, rmse_values_width_S1)])

        all_widths = np.concatenate([df['in:Length [m]'].values.flatten() for df in all_range_of_input])


    if constant_variable == 'Width':
        constant_value_df = pd.DataFrame({'in:Length [m]': [constant_value]})
        constant_value_df.columns = ['in:Length [m]']
        variable_range_df = pd.DataFrame({'in:Height [m]': [variable_range]})
        variable_range_df.columns = ['in:Height [m]']

        input_data_dot = pd.DataFrame({
            'in:Length [m]': constant_value_df['in:Length [m]'],
            'in:Height [m]': variable_range_df['in:Height [m]']
       })

    else:
        constant_value_df = pd.DataFrame({'in:Height [m]': [constant_value]})
        constant_value_df.columns = ['in:Height [m]']
        variable_range_df = pd.DataFrame({'in:Length [m]': [variable_range]})
        variable_range_df.columns = ['in:Length [m]']

        input_data_dot = pd.DataFrame({
            'in:Length [m]': variable_range_df['in:Length [m]'],
            'in:Height [m]': constant_value_df['in:Height [m]']
        })


    # Extract the relevant columns for plotting
    if constant_variable == 'Width':
        X_filtered = input_data['in:Height [m]']
    else:
        X_filtered = input_data['in:Length [m]']

    # Create subplots
    plt.figure(figsize=(12, 12))


    # Plot for Total Costs per GFA
    plt.subplot(2, 1, 1)

    #plt.xlim(0, max(X_filtered) + 20)
    #plt.ylim(0, max(p_total_costs_S1) + 50)

    plt.plot(X_filtered, p_total_costs_S1, color=color_green, label=f'S1: Prediction ANN S1-A')
    plt.scatter(X_filtered_S1, y_costs_S1, color=color_lightgreen, label='S1: Input Data w = 21.6m', alpha=0.7)

    # Plot upper and lower bands
    if constant_variable == 'Width':
        plt.plot(all_heights, all_predictions_upper_S1, '--', color=color_green, label='Upper Band S1',alpha = 0.2)
        plt.plot(all_heights, all_predictions_lower_S1, '--', color=color_green, label='Lower Band S1', alpha = 0.2)

    else:
        plt.plot(all_widths, all_predictions_upper_S1, '--', color=color_green, label='Upper Band S1',alpha = 0.2)
        plt.plot(all_widths, all_predictions_lower_S1, '--', color=color_green, label='Lower Band S1', alpha = 0.2)

    plt.title(f'Total $\mathbf{{Structural}}$ $\mathbf{{Costs}}$ [€/m2GFA] for {constant_variable}={constant_value}')

    if constant_variable == 'Width':
        plt.xlabel('Height [m]')

    else:
        plt.xlabel('Width [m]')

    plt.ylabel('Total Structural Costs [€/m2GFA]')
    plt.legend(bbox_to_anchor=(1.02, 1), loc='upper left', borderaxespad=0)


    # Adjust layout
    plt.tight_layout()
    plt.ylim(0, 3000)
    plt.xlim(0, 320)
    plt.show()

# Create an interactive output widget
output = widgets.interactive_output(make_predictions_and_plot_S1, {
    'constant_variable': constant_variable_dropdown,
    'constant_value': constant_value_input,
    'variable_range': variable_slider
})


# Create an interactive layout with sliders and output
interactive_layout = widgets.VBox([
    constant_variable_dropdown,
    constant_value_input,
    variable_slider,
    output
])

# Display the interactive layout
display(interactive_layout)
```

In [ ]:
```
# Define constant variables
constant_options = ['Width', 'Height']

# Create interactive sliders and dropdown for constant variables
constant_variable_dropdown = widgets.Dropdown(options=constant_options, value='Width', description='Constant Parameter:')
constant_value_input = widgets.FloatText(value=20, description='Constant Parameter:')
variable_slider = widgets.FloatSlider(value=20, min=5, max=500, step=1, description='Variable Parameter:')


# Update descriptions based on the constant variable
def update_slider_descriptions(constant_variable):
    if constant_variable == 'Height':
        constant_value_input.description = 'Constant Parameter = Height'
        variable_slider.description = 'Variable Parameter = Width'
        variable_slider.min = 10
        variable_slider.max = 60
    else:
        constant_value_input.description = 'Constant Parameter = Width'
        variable_slider.description = 'Variable Parameter = Height'
        variable_slider.min = 60
        variable_slider.max = 300

# Set width for the sliders and dropdown
constant_variable_dropdown.layout.width = '40%'
constant_value_input.layout.width = '40%'
variable_slider.layout.width = '40%'
constant_variable_dropdown.style = {'description_width': '50%'}
constant_value_input.style = {'description_width': '50%'}
variable_slider.style = {'description_width': '50%'}

# Observe changes in the dropdown and update slider descriptions accordingly
widgets.interactive(update_slider_descriptions, constant_variable=constant_variable_dropdown)

# Function to make predictions and plot the graph
def make_predictions_and_plot_S1(constant_variable, constant_value, variable_range):
    # Generate a range of variable values within the specified range
    variable_values = np.linspace(variable_slider.min, variable_slider.max, 100).reshape(-1, 1)

    # Prepare input data
    if constant_variable == 'Width':
        input_data = pd.DataFrame({
            'in:Length [m]': constant_value,
            'in:Height [m]': variable_values.flatten()
        })

    else:
        input_data = pd.DataFrame({
            'in:Length [m]': variable_values.flatten(),
            'in:Height [m]': constant_value
        })

    input_data1 = scaler1.transform(input_data)
    input_data2 = scaler2.transform(input_data)

    # Convert the predictons from mass to costs and embodied carbon
    # Costs
    predictions_S2 = best_model_S2.predict(input_data2)
    predictions_S2[:, 0] *= s_price
    predictions_S2[:, 1] *= pc_rc_price
    p_total_costs_S2 = predictions_S2.sum(axis=1)

    # Upper and lower bands based on RMSE for each system

    rmse_values_width_S2 = rmse_values_S2_diffwidth
    rmse_values_height_S2 = rmse_values_S2_diffheight

    height_ranges = [(50, 100), (100, 150), (150, 200), (200, 250), (250,300)]
    width_ranges = [(10, 20), (20, 30), (30, 40), (40, 50), (50,60)]

    # Choose the appropriate RMSE values based on the selected constant variable
    if constant_variable == 'Width':
        rmse_values_S2 = rmse_values_height_S2
        all_predictions_S2 = []
        all_range_of_input = []

        for height_range in height_ranges:
            indices = input_data[(input_data['in:Height [m]'] >= height_range[0]) & (input_data['in:Height [m]'] < height_range[1])].index
            range_of_input = input_data.loc[indices]
            range_of_input_scaled2 = scaler2.transform(range_of_input)
            predictions_S2 = best_model_S2.predict(range_of_input_scaled2)
            predictions_S2[:, 0] *= s_price
            predictions_S2[:, 1] *= pc_rc_price
            all_predictions_S2.append(predictions_S2.sum(axis=1))

            all_range_of_input.append(range_of_input)

        all_predictions_upper_S2 = np.concatenate([arr + rmse for arr, rmse in zip(all_predictions_S2, rmse_values_height_S2)])
        all_predictions_lower_S2 = np.concatenate([arr - rmse for arr, rmse in zip(all_predictions_S2, rmse_values_height_S2)])

        all_heights = np.concatenate([df['in:Height [m]'].values.flatten() for df in all_range_of_input])


    else:
        rmse_values_S2 = rmse_values_width_S2
        all_predictions_S2 = []
        all_range_of_input = []

        for width_range in width_ranges:
            indices = input_data[(input_data['in:Length [m]'] >= width_range[0]) & (input_data['in:Length [m]'] < width_range[1])].index
            range_of_input = input_data.loc[indices]

            range_of_input_scaled2 = scaler2.transform(range_of_input)
```

```
            predictions_S2 = best_model_S2.predict(range_of_input_scaled2)
            predictions_S2[:, 0] *= s_price
            predictions_S2[:, 1] *= pc_rc_price
            all_predictions_S2.append(predictions_S2.sum(axis=1))

            all_range_of_input.append(range_of_input)

        all_predictions_upper_S2 = np.concatenate([arr + rmse for arr, rmse in zip(all_predictions_S2, rmse_values_width_S2)])
        all_predictions_lower_S2 = np.concatenate([arr - rmse for arr, rmse in zip(all_predictions_S2, rmse_values_width_S2)])

        all_widths = np.concatenate([df['in:Length [m]'].values.flatten() for df in all_range_of_input])


    if constant_variable == 'Width':
        constant_value_df = pd.DataFrame({'in:Length [m]': [constant_value]})
        constant_value_df.columns = ['in:Length [m]']
        variable_range_df = pd.DataFrame({'in:Height [m]': [variable_range]})
        variable_range_df.columns = ['in:Height [m]']

        input_data_dot = pd.DataFrame({
            'in:Length [m]': constant_value_df['in:Length [m]'],
            'in:Height [m]': variable_range_df['in:Height [m]']
       })

    else:
        constant_value_df = pd.DataFrame({'in:Height [m]': [constant_value]})
        constant_value_df.columns = ['in:Height [m]']
        variable_range_df = pd.DataFrame({'in:Length [m]': [variable_range]})
        variable_range_df.columns = ['in:Length [m]']

        input_data_dot = pd.DataFrame({
            'in:Length [m]': variable_range_df['in:Length [m]'],
            'in:Height [m]': constant_value_df['in:Height [m]']
        })


    # Extract the relevant columns for plotting
    if constant_variable == 'Width':
        X_filtered = input_data['in:Height [m]']
    else:
        X_filtered = input_data['in:Length [m]']

    # Create subplots
    plt.figure(figsize=(12, 12))


    # Plot for Total Costs per GFA
    plt.subplot(2, 1, 1)

    plt.plot(X_filtered, p_total_costs_S2, color=color_darkblue, label=f'S2: Prediction ANN S2-A')
    plt.scatter(X_filtered_S2, y_costs_S2, color=color_lightblue, label='S1: Input Data w = 14.4m', alpha=0.7)


    # Plot upper and lower bands
    if constant_variable == 'Width':
        plt.plot(all_heights, all_predictions_upper_S2, '--', color=color_darkblue, label='Upper Band S2',alpha = 0.2)
        plt.plot(all_heights, all_predictions_lower_S2, '--', color=color_darkblue, label='Lower Band S2', alpha = 0.2)

    else:
        plt.plot(all_widths, all_predictions_upper_S2, '--', color=color_darkblue, label='Upper Band S2',alpha = 0.2)
        plt.plot(all_widths, all_predictions_lower_S2, '--', color=color_darkblue, label='Lower Band S2', alpha = 0.2)

    plt.title(f'Total $\mathbf{{Structural}}$ $\mathbf{{Costs}}$ [€/m2GFA] for {constant_variable}={constant_value}')

    if constant_variable == 'Width':
        plt.xlabel('Height [m]')

    else:
        plt.xlabel('Width [m]')

    plt.ylabel('Total Structural Costs [€/m2GFA]')
    plt.legend(bbox_to_anchor=(1.02, 1), loc='upper left', borderaxespad=0)


    # Adjust Layout
    plt.tight_layout()
    plt.ylim(0, 3000)
    plt.xlim(0, 320)
    plt.show()

# Create an interactive output widget
output = widgets.interactive_output(make_predictions_and_plot_S1, {
    'constant_variable': constant_variable_dropdown,
    'constant_value': constant_value_input,
    'variable_range': variable_slider
})


# Create an interactive layout with sliders and output
interactive_layout = widgets.VBox([
    constant_variable_dropdown,
    constant_value_input,
    variable_slider,
    output
])

# Display the interactive layout
display(interactive_layout)
```

RMSE for different height ranges

S1: Braced Framed Tube

Different height ranges

In [ ]:
```
# Function to calculate RMSE for different height ranges
height_ranges = [(50, 100), (100, 150), (150, 200), (200, 250), (250,300)]  # Define the height ranges
rmse_values_S1_diffheight = []

for height_range in height_ranges:
    indices_train = X_train[(X_train['in:Height [m]'] >= height_range[0]) & (X_train['in:Height [m]'] < height_range[1])].index
    train_values = y_train.loc[indices_train]
    train_X = X_train.loc[indices_train]

    indices_test = X_test[(X_test['in:Height [m]'] >= height_range[0]) & (X_test['in:Height [m]'] < height_range[1])].index
    test_values = y_test.loc[indices_test]
    test_X = X_test.loc[indices_test]

    test_X_scaled = scaler1.transform(test_X)
    predictions_values = best_model_S1.predict(test_X_scaled)

    rmse = np.sqrt(mean_squared_error(test_values, predictions_values))
    rmse_values_S1_diffheight.append(rmse)


rmse_values_S1_diffheight = [rmse * s_price for rmse in rmse_values_S1_diffheight]

# Print the RMSE values for different height ranges
for i, height_range in enumerate(height_ranges):
    print(f"RMSE for the range of height: {height_range[0]}-{height_range[1]} for S1: {round(rmse_values_S1_diffheight[i],1)}" f" [€/GFA]"
```

Different height ranges

In [ ]:
```
width_ranges = [(10, 20), (20, 30), (30, 40), (40, 50), (50,60)]  # Define the width ranges
rmse_values_S1_diffwidth = []

for width_range in width_ranges:
    indices_train = X_train[(X_train['in:Length [m]'] >= width_range[0]) & (X_train['in:Length [m]'] < width_range[1])].index
    train_values = y_train.loc[indices_train]
    train_X = X_train.loc[indices_train]

    indices_test = X_test[(X_test['in:Length [m]'] >= width_range[0]) & (X_test['in:Length [m]'] < width_range[1])].index
    test_values = y_test.loc[indices_test]
    test_X = X_test.loc[indices_test]

    test_X_scaled = scaler1.transform(test_X)
    predictions_values = best_model_S1.predict(test_X_scaled)

    rmse = np.sqrt(mean_squared_error(test_values, predictions_values))
    rmse_values_S1_diffwidth.append(rmse)

rmse_values_S1_diffwidth = [rmse * s_price for rmse in rmse_values_S1_diffwidth]

# Print the RMSE values for different width ranges
for i, width_range in enumerate(width_ranges):
    print(f"RMSE for the range of width: {width_range[0]}-{width_range[1]} for S1: {round(rmse_values_S1_diffwidth[i],1)}" f" [€/GFA]")
```

S2: Outrigger

Different height ranges

In [ ]:
```
rmse_values_S2_diffheight_steel = []
rmse_values_S2_diffheight_concrete = []

for height_range in height_ranges:
    indices_train = X_train2[(X_train['in:Height [m]'] >= height_range[0]) & (X_train2['in:Height [m]'] < height_range[1])].index
    train_values = y_train2.loc[indices_train]
    train_X = X_train2.loc[indices_train]

    indices_test = X_test2[(X_test2['in:Height [m]'] >= height_range[0]) & (X_test2['in:Height [m]'] < height_range[1])].index
    test_values = y_test2.loc[indices_test]
    test_X = X_test2.loc[indices_test]

    test_X_scaled = scaler2.transform(test_X)
    predictions_values = best_model_S2.predict(test_X_scaled)

    predictions_values_S2_1 = predictions_values[:, 0]
    predictions_values_S2_2 = predictions_values[:, 1]

    # Separate y_true for each output of S2
    test_values_S2_1 = test_values.iloc[:, 0]
    test_values_S2_2 = test_values.iloc[:, 1]

    # Calculate RMSE for each output of S2
    rmse_S2_1 = np.sqrt(mean_squared_error(test_values_S2_1, predictions_values_S2_1))
    rmse_S2_2 = np.sqrt(mean_squared_error(test_values_S2_2, predictions_values_S2_2))

    rmse_values_S2_diffheight_steel.append(rmse_S2_1)
    rmse_values_S2_diffheight_concrete.append(rmse_S2_2)

    rmse = np.sqrt(mean_squared_error(test_values, predictions_values))
    rmse_values_S1_diffheight.append(rmse)

rmse_values_S1_diffheight_steel = [rmse * s_price for rmse in rmse_values_S2_diffheight_steel]
rmse_values_S1_diffheight_concrete = [rmse * pc_rc_price for rmse in rmse_values_S2_diffheight_concrete]

rmse_values_S2_diffheight = [a + b for a, b in zip(rmse_values_S2_diffheight_steel, rmse_values_S2_diffheight_concrete)]

# Print the RMSE values for different height ranges for the first output of S2
for i, height_range in enumerate(height_ranges):
    print(f"RMSE for the range of height: {height_range[0]}-{height_range[1]} (S2: Steel & Output Concrete): "
          f"{round(rmse_values_S2_diffheight_steel[i],1)} and {round(rmse_values_S2_diffheight_concrete[i],1)}"  f" [€/GFA]")

print("")
for i, height_range in enumerate(height_ranges):
    print(f"RMSE for the range of heigt: {height_range[0]}-{height_range[1]} for S2: "
          f"{round(rmse_values_S2_diffheight[i],1)}"  f" [€/GFA]")
```

Different width ranges

In [ ]:
```
rmse_values_S2_diffwidth_steel = []
rmse_values_S2_diffwidth_concrete = []

for width_range in width_ranges:
    indices_train = X_train2[(X_train2['in:Length [m]'] >= width_range[0]) & (X_train2['in:Length [m]'] < width_range[1])].index
    train_values = y_train2.loc[indices_train]
    train_X = X_train2.loc[indices_train]

    indices_test = X_test2[(X_test2['in:Length [m]'] >= width_range[0]) & (X_test2['in:Length [m]'] < width_range[1])].index
    test_values = y_test2.loc[indices_test]
    test_X = X_test2.loc[indices_test]

    test_X_scaled = scaler2.transform(test_X)
    predictions_values = best_model_S2.predict(test_X_scaled)

    predictions_values_S2_1 = predictions_values[:, 0]
    predictions_values_S2_2 = predictions_values[:, 1]

    # Separate y_true for each output of S2
    test_values_S2_1 = test_values.iloc[:, 0]
    test_values_S2_2 = test_values.iloc[:, 1]

    # Calculate RMSE for each output of S2
    rmse_S2_1 = np.sqrt(mean_squared_error(test_values_S2_1, predictions_values_S2_1))
    rmse_S2_2 = np.sqrt(mean_squared_error(test_values_S2_2, predictions_values_S2_2))

    rmse_values_S2_diffwidth_steel.append(rmse_S2_1)
    rmse_values_S2_diffwidth_concrete.append(rmse_S2_2)

rmse_values_S2_diffwidth_steel = [rmse * s_price for rmse in rmse_values_S2_diffwidth_steel]
rmse_values_S2_diffwidth_concrete = [rmse * pc_rc_price for rmse in rmse_values_S2_diffwidth_concrete]

rmse_values_S2_diffwidth = [a + b for a, b in zip(rmse_values_S2_diffwidth_steel, rmse_values_S2_diffwidth_concrete)]

# Print the RMSE values for different height ranges for the first output of S2
for i, width_range in enumerate(width_ranges):
    print(f"RMSE for the range of width: {width_range[0]}-{width_range[1]} (S2: Steel & Output Concrete): "
          f"{round(rmse_values_S2_diffwidth_steel[i],1)} and {round(rmse_values_S2_diffwidth_concrete[i],1)}"  f" [€/GFA]")

print("")
for i, width_range in enumerate(width_ranges):
    print(f"RMSE for the range of width: {width_range[0]}-{width_range[1]} for S2:: "
          f"{round(rmse_values_S2_diffwidth[i],1)}"  f" [€/GFA]")
```

7) COMPARISON WITH LITERATURE

A) Division of number of stories

In [ ]:
```
import math
```

In [ ]:
```
# Function to make predictions and plot boxplots
def make_predictions_EmCO2_boxplot(length_range, height_range1, height_range2, height_range3, height_range4):
    # Prepare input data
    lengths = np.arange(length_range[0], length_range[1] + 1)
    predictions_range1 = []
    predictions_range2 = []
    predictions_range3 = []
    predictions_range4 = []

    for length in lengths:
        heights1 = np.arange(height_range1[0], height_range1[1] + 1, 3)
        heights2 = np.arange(height_range2[0], height_range2[1] + 1, 3)
        heights3 = np.arange(height_range3[0], height_range3[1] + 1, 3)
        heights4 = np.arange(height_range4[0], height_range4[1] + 1, 3)

        for height1 in heights1:
            input_data_range1 = pd.DataFrame({'in:Length [m]': [length], 'in:Height [m]': [height1]})
            input_data1_range1 = scaler1.transform(input_data_range1)
            input_data2_range1 = scaler2.transform(input_data_range1)

            prediction_S1 = best_model_S1.predict(input_data1_range1) * s_EmCO2
            prediction_S2 = best_model_S2.predict(input_data2_range1)[:, 0] * s_EmCO2 + best_model_S2.predict(input_data2_range1)[:, 1] *
            if np.isnan(prediction_S1):
                prediction_S1 == 0
            if np.isnan(prediction_S2):
                prediction_S2 == 0
            min_prediction = np.min([prediction_S1, prediction_S2]) + 65.23
            predictions_range1.append(min_prediction)

        for height2 in heights2:
            input_data_range2 = pd.DataFrame({'in:Length [m]': [length], 'in:Height [m]': [height2]})
            input_data1_range2 = scaler1.transform(input_data_range2)
            input_data2_range2 = scaler2.transform(input_data_range2)

            prediction_S1 = best_model_S1.predict(input_data1_range2) * s_EmCO2
            prediction_S2 = best_model_S2.predict(input_data2_range2)[:, 0] * s_EmCO2 + best_model_S2.predict(input_data2_range2)[:, 1] *
            if math.isnan(prediction_S1):
                prediction_S1 == 0
            if math.isnan(prediction_S2):
                prediction_S2 == 0
            min_prediction = np.min([prediction_S1, prediction_S2]) + 65.23
            predictions_range2.append(min_prediction)

        for height3 in heights3:
            input_data_range3 = pd.DataFrame({'in:Length [m]': [length], 'in:Height [m]': [height3]})
            input_data1_range3 = scaler1.transform(input_data_range3)
            input_data2_range3 = scaler2.transform(input_data_range3)

            prediction_S1 = best_model_S1.predict(input_data1_range3) * s_EmCO2
            prediction_S2 = best_model_S2.predict(input_data2_range3)[:, 0] * s_EmCO2 + best_model_S2.predict(input_data2_range3)[:, 1] *
            if np.isnan(prediction_S1):
                prediction_S1 == 0
            if np.isnan(prediction_S2):
                prediction_S2 == 0
            min_prediction = np.min([prediction_S1, prediction_S2]) + 65.23
            predictions_range3.append(min_prediction)

        for height4 in heights4:
            input_data_range4 = pd.DataFrame({'in:Length [m]': [length], 'in:Height [m]': [height4]})
            input_data1_range4 = scaler1.transform(input_data_range4)
            input_data2_range4 = scaler2.transform(input_data_range4)

            prediction_S1 = best_model_S1.predict(input_data1_range4) * s_EmCO2
            prediction_S2 = best_model_S2.predict(input_data2_range4)[:, 0] * s_EmCO2 + best_model_S2.predict(input_data2_range4)[:, 1] *
            if np.isnan(prediction_S1):
                prediction_S1 == 0
            if np.isnan(prediction_S2):
                prediction_S2 == 0
            min_prediction = np.min([prediction_S1, prediction_S2]) + 65.23
            predictions_range4.append(min_prediction)

    predictions_range1 = np.array(predictions_range1)
    predictions_range2 = np.array(predictions_range2)
    predictions_range3 = np.array(predictions_range3)
    predictions_range4 = np.array(predictions_range4)

    return predictions_range1, predictions_range2, predictions_range3, predictions_range4

length_range = (15, 60)
height_range1 = (48, 150)
height_range2 = (153, 201)
height_range3 = (204, 252)
height_range4 = (255, 303)

predictions_range1, predictions_range2, predictions_range3, predictions_range4 = make_predictions_EmCO2_boxplot(length_range, height_range
```

In [ ]:
```
predictions_range1
```

In [ ]:
```
data = [predictions_range1, predictions_range2, predictions_range3, predictions_range4]
medians = [np.median(x) for x in data]
q1 = [np.percentile(x, 25) for x in data]
q3 = [np.percentile(x, 75) for x in data]
iqr = [q3[i] - q1[i] for i in range(len(data))]
upper_whisker = [q3[i] + 1.5 * iqr[i] for i in range(len(data))]
lower_whisker = [q3[i] - 1.5 * iqr[i] for i in range(len(data))]
```

In [ ]:
```
plt.figure(figsize=(14, 7))
bp = plt.boxplot(data, patch_artist=True, showfliers=False, labels=['11 - 50 stories', '51 - 70 stories', '71 - 90 stories', '91 - 100 sto

# Customizing box colors
colors = [color_green,color_green, color_green,color_green]
for box, color in zip(bp['boxes'], colors):
    box.set(color='black', linewidth=0.5)
    box.set(facecolor=color)

for median in bp['medians']:
    median.set(color='black')

# Add median, q25, and q75 annotations
for i in range(len(data)):
    plt.text(i + 1.2, medians[i], f'median: {int(medians[i])}', horizontalalignment='left', verticalalignment='center', fontsize=11)
    plt.text(i + 1.2, q1[i] - 20, f'Q1: {int(q1[i])}', horizontalalignment='left', verticalalignment='top', fontsize=11)
    plt.text(i + 1.2, q3[i] + 20, f'Q3: {int(q3[i])}', horizontalalignment='left', verticalalignment='bottom', fontsize=11)
    #plt.text(i + 0.7, upper_whisker[i] + 20 , f'Max: {int(upper_whisker[i])}', horizontalalignment='left', verticalalignment='bottom', fo
    plt.text(i + 0.7, lower_whisker[i] - 20, f'Min: {int(lower_whisker[i])}', horizontalalignment='left', verticalalignment='top', fontsiz

plt.title('Embodied Carbon (Cradle-to-Gate) for Different Ranges of Stories', fontsize = 14)
plt.xlabel('Number of Stories', fontsize = 14)
plt.ylabel('Embodied Carbon (kgCO2eq/m2-GFA)', fontsize = 14)
plt.xticks(fontsize=13)  # Increase fontsize of tick labels
plt.grid(True)
plt.show()
```

B) Division of floor area

In [ ]:
```
length_range = (15, 60)
lengths = np.arange(length_range[0], length_range[1] + 1)

height_range = (48, 300)
heights = np.arange(height_range[0], height_range[1] + 1, 3)

floor_area_array = []
for length in lengths:
    for height in heights:
        floors = height / 3
        floor_area = floors * length * length
        floor_area_array.append((length, height, floor_area))

smaller_than_10k = []
between_10k_and_100k = []
bigger_than_100k = []

for entry in floor_area_array:
    if entry[2] < 10000:
        smaller_than_10k.append(entry)
    elif 10000 <= entry[2] < 100000:
        between_10k_and_100k.append(entry)
    else:
        bigger_than_100k.append(entry)

print("Number of floor areas smaller than 10,000:", len(smaller_than_10k))
print("Number of floor areas between 10,000 and 100,000:", len(between_10k_and_100k))
print("Number of floor areas bigger than 100,000:", len(bigger_than_100k))
```

In [ ]:
```
range1 = smaller_than_10k
range2 = between_10k_and_100k
range3 = bigger_than_100k
```

In [ ]:
```
predictions_range1_FA = []
for i in range(len(range1)):
    if range1[i][1] / range1[i][0] < 7.9:
        input_data_range1 = pd.DataFrame({'in:Length [m]': [range1[i][0]], 'in:Height [m]': [range1[i][1]]})
        input_data1_range1 = scaler1.transform(input_data_range1)
        input_data2_range1 = scaler2.transform(input_data_range1)

        prediction_S1 = best_model_S1.predict(input_data1_range1) * s_EmCO2
        prediction_S2 = best_model_S2.predict(input_data2_range1)[:, 0] * s_EmCO2 + best_model_S2.predict(input_data2_range1)[:, 1] * pc_r
        min_prediction = np.nanmin([prediction_S1, prediction_S2]) + 65.23
        predictions_range1_FA = np.append(predictions_range1_FA, min_prediction)

print(len(predictions_range1_FA))

predictions_range2_FA = []
for i in range(len(range2)):
    if range2[i][1] / range2[i][0] < 7.9:
        input_data_range2 = pd.DataFrame({'in:Length [m]': [range2[i][0]], 'in:Height [m]': [range2[i][1]]})
        input_data1_range2 = scaler1.transform(input_data_range2)
        input_data2_range2 = scaler2.transform(input_data_range2)

        prediction_S1 = best_model_S1.predict(input_data1_range2) * s_EmCO2
        prediction_S2 = best_model_S2.predict(input_data2_range2)[:, 0] * s_EmCO2 + best_model_S2.predict(input_data2_range2)[:, 1] * pc_r
        min_prediction = np.nanmin([prediction_S1, prediction_S2]) + 65.23
        predictions_range2_FA = np.append(predictions_range2_FA, min_prediction)

print(len(predictions_range2_FA))

predictions_range3_FA = []
for i in range(len(range3)):
    if range3[i][1] / range3[i][0] < 7.9:
        input_data_range3 = pd.DataFrame({'in:Length [m]': [range3[i][0]], 'in:Height [m]': [range3[i][1]]})
        input_data1_range3 = scaler1.transform(input_data_range3)
        input_data2_range3 = scaler2.transform(input_data_range3)

        prediction_S1 = best_model_S1.predict(input_data1_range3) * s_EmCO2
        prediction_S2 = best_model_S2.predict(input_data2_range3)[:, 0] * s_EmCO2 + best_model_S2.predict(input_data2_range3)[:, 1] * pc_r
        min_prediction = np.nanmin([prediction_S1, prediction_S2]) + 65.23
        predictions_range3_FA = np.append(predictions_range3_FA, min_prediction)

print(len(predictions_range3_FA))
```

In [ ]:
```
data_FA = [predictions_range1_FA, predictions_range2_FA, predictions_range3_FA]
medians = [np.median(x) for x in data_FA]
q1 = [np.percentile(x, 25) for x in data_FA]

q3 = [np.percentile(x, 75) for x in data_FA]
iqr = [q3[i] - q1[i] for i in range(len(data_FA))]

upper_whisker = [q3[i] + 1.5 * iqr[i] for i in range(len(data_FA))]

minimum = [min(data_FA[i]) for i in range(len(data_FA))]

lower_whisker = [q1[i] - 1.5 * iqr[i] for i in range(len(data_FA))]
lower_whisker = [max(minimum[i], lower_whisker[i]) for i in range(len(data_FA))]

print(lower_whisker)
print(q1)
print(medians)
print(q3)
print(upper_whisker)
```

In [ ]:
```
plt.figure(figsize=(13, 5))
bp = plt.boxplot(data_FA, patch_artist=True, showfliers=False, labels=['< 10,000 m2 (n=125)', '10,000 - 100,000 m2 (n=2391)', '> 100,000 m

# Customizing box colors
colors = [color_green,color_green, color_green,color_green]
for box, color in zip(bp['boxes'], colors):
    box.set(color='black', linewidth=0.5)
    box.set(facecolor=color)

for median in bp['medians']:
    median.set(color='black')

plt.title('Embodied Carbon (Cradle-to-Gate) for Different Ranges of Floor Area', fontsize = 14)
plt.xlabel('Floor Area (m2)', fontsize = 14)
plt.ylabel('Embodied Carbon (kgCO2eq/m2-GFA)', fontsize = 14)
plt.xticks(fontsize=13)  # Increase fontsize of tick labels
plt.ylim(0, 1200)
plt.grid(True)
plt.show()
```

8) IMPACT OPTIMIZING STABILITY FRAMEWORK

In [ ]:
```
length_range = (15, 60)
lengths = np.arange(length_range[0], length_range[1] + 1)

height_range1 = (48, 103)
heights1 = np.arange(height_range1[0], height_range1[1], 3)

height_range2 = (103, 153)
heights2 = np.arange(height_range2[0], height_range2[1], 3)

height_range3 = (153, 203)
heights3 = np.arange(height_range3[0], height_range3[1], 3)

height_range4 = (203, 253)
heights4 = np.arange(height_range4[0], height_range4[1], 3)

height_range5 = (253, 303)
heights5 = np.arange(height_range5[0], height_range5[1], 3)

input_array1 = []
for length in lengths:
    for height in heights1:
        input_array1.append((length, height))

input_array2 = []
for length in lengths:
    for height in heights2:
        input_array2.append((length, height))

input_array3 = []
for length in lengths:
    for height in heights3:
        input_array3.append((length, height))

input_array4 = []
for length in lengths:
    for height in heights4:
        input_array4.append((length, height))

input_array5 = []
for length in lengths:
    for height in heights5:
        input_array5.append((length, height))

print('Length of the input data set of height range 11-50 stories is', len(input_array1))
print('Length of the input data set of height range 51-100 stories is', len(input_array2))
print('Length of the input data set of height range 101-150 stories is', len(input_array3))
print('Length of the input data set of height range 151-200 stories is', len(input_array4))
print('Length of the input data set of height range 201-250 stories is', len(input_array5))
```

In [ ]:
```
floor_impact = 65.23

diff_range1 = []
for i in range(len(input_array1)):
    if input_array1[i][1] / input_array1[i][0] < 11:
        input_data_range1 = pd.DataFrame({'in:Length [m]': [input_array1[i][0]], 'in:Height [m]': [input_array1[i][1]]})
        input_data1_range1 = scaler1.transform(input_data_range1)
        input_data2_range1 = scaler2.transform(input_data_range1)

        prediction_S1 = best_model_S1.predict(input_data1_range1) * s_EmCO2
        prediction_S2 = best_model_S2.predict(input_data2_range1)[:, 0] * s_EmCO2 + best_model_S2.predict(input_data2_range1)[:, 1] * pc_r
        difference = abs(prediction_S1 - prediction_S2)
        total_best_option = min(prediction_S1, prediction_S2) + floor_impact
        share = (difference / total_best_option) * 100
        diff_range1 = np.append(diff_range1, share)

diff_range2 = []
for i in range(len(input_array2)):
    if input_array2[i][1] / input_array2[i][0] < 11:
        input_data_range2 = pd.DataFrame({'in:Length [m]': [input_array2[i][0]], 'in:Height [m]': [input_array2[i][1]]})
        input_data1_range2 = scaler1.transform(input_data_range2)
        input_data2_range2 = scaler2.transform(input_data_range2)

        prediction_S1 = best_model_S1.predict(input_data1_range2) * s_EmCO2
        prediction_S2 = best_model_S2.predict(input_data2_range2)[:, 0] * s_EmCO2 + best_model_S2.predict(input_data2_range2)[:, 1] * pc_r
        difference = abs(prediction_S1 - prediction_S2)
        total_best_option = min(prediction_S1, prediction_S2) + floor_impact
        share = (difference / total_best_option) * 100
        diff_range2 = np.append(diff_range2, share)

diff_range3 = []
for i in range(len(input_array3)):
    if input_array3[i][1] / input_array3[i][0] < 11:
        input_data_range3 = pd.DataFrame({'in:Length [m]': [input_array3[i][0]], 'in:Height [m]': [input_array3[i][1]]})
        input_data1_range3 = scaler1.transform(input_data_range3)
        input_data2_range3 = scaler2.transform(input_data_range3)

        prediction_S1 = best_model_S1.predict(input_data1_range3) * s_EmCO2
        prediction_S2 = best_model_S2.predict(input_data2_range3)[:, 0] * s_EmCO2 + best_model_S2.predict(input_data2_range3)[:, 1] * pc_r
        difference = abs(prediction_S1 - prediction_S2)
        total_best_option = min(prediction_S1, prediction_S2) + floor_impact
        share = (difference / total_best_option) * 100
        diff_range3 = np.append(diff_range3, share)

diff_range4 = []
for i in range(len(input_array4)):
    if input_array4[i][1] / input_array4[i][0] < 11:
        input_data_range4 = pd.DataFrame({'in:Length [m]': [input_array4[i][0]], 'in:Height [m]': [input_array4[i][1]]})
        input_data1_range4 = scaler1.transform(input_data_range4)
        input_data2_range4 = scaler2.transform(input_data_range4)

        prediction_S1 = best_model_S1.predict(input_data1_range4) * s_EmCO2
        prediction_S2 = best_model_S2.predict(input_data2_range4)[:, 0] * s_EmCO2 + best_model_S2.predict(input_data2_range4)[:, 1] * pc_r
        difference = abs(prediction_S1 - prediction_S2)
        total_best_option = min(prediction_S1, prediction_S2) + floor_impact
        share = (difference / total_best_option) * 100
        diff_range4 = np.append(diff_range4, share)

diff_range5 = []
for i in range(len(input_array5)):
    if input_array5[i][1] / input_array5[i][0] < 11:
        input_data_range5 = pd.DataFrame({'in:Length [m]': [input_array5[i][0]], 'in:Height [m]': [input_array5[i][1]]})
        input_data1_range5 = scaler1.transform(input_data_range5)
        input_data2_range5 = scaler2.transform(input_data_range5)

        prediction_S1 = best_model_S1.predict(input_data1_range5) * s_EmCO2
        prediction_S2 = best_model_S2.predict(input_data2_range5)[:, 0] * s_EmCO2 + best_model_S2.predict(input_data2_range5)[:, 1] * pc_r
        if math.isnan(prediction_S1):
            prediction_S1 == np.nan
        if math.isnan(prediction_S2):
            prediction_S2 == np.nan
        difference = abs(prediction_S1 - prediction_S2)
        total_best_option = min(prediction_S1, prediction_S2) + floor_impact
        share = (difference / total_best_option) * 100
        diff_range5 = np.append(diff_range5, share)
```

In [ ]:
```
diff_range1 = diff_range1[~np.isnan(diff_range1)]
diff_range2 = diff_range2[~np.isnan(diff_range2)]
diff_range3 = diff_range3[~np.isnan(diff_range3)]
diff_range4 = diff_range4[~np.isnan(diff_range4)]
diff_range5 = diff_range5[~np.isnan(diff_range5)]

print(len(diff_range1))
print(len(diff_range2))
print(len(diff_range3))
print(len(diff_range4))
print(len(diff_range5))
```

In [ ]:

In [ ]:
```
data_diff = [diff_range1, diff_range2, diff_range3, diff_range4, diff_range5]
medians = [np.median(x) for x in data_diff]
q1 = [np.percentile(x, 25) for x in data_diff]
print(medians)

q3 = [np.percentile(x, 75) for x in data_diff]
iqr = [q3[i] - q1[i] for i in range(len(data_diff))]
upper_whisker = [q3[i] + 1.0 * iqr[i] for i in range(len(data_diff))]
minimum = [min(data_diff[i]) for i in range(len(data_diff))]
lower_whisker = [q1[i] - 1.0 * iqr[i] for i in range(len(data_diff))]
lower_whisker = [max(minimum[i], lower_whisker[i]) for i in range(len(data_diff))]
```

In [ ]:
```
plt.figure(figsize=(14, 6))
bp = plt.boxplot(data_diff, patch_artist=True, showfliers=False, labels=['48 - 100 m', '100 - 150 m', '150 - 200 m', '200 - 250 m', '250 -

# Customizing box colors
colors = [color_grey, color_grey, color_grey, color_grey, color_grey]
for box, color in zip(bp['boxes'], colors):
    box.set(color='black', linewidth=0.5)
    box.set(facecolor=color)

for median in bp['medians']:
    median.set(color='black')

plt.title('Percentage of the Gain of the Best Stability System over the Total Embodied Carbon (Framework + Floors)', fontsize = 14)
plt.xlabel('Height of the Structure', fontsize = 14)
plt.ylabel('Percentage [%]', fontsize = 14)
plt.xticks(fontsize=13)
plt.ylim(0,100)
plt.grid(True)
plt.show()
```

In [ ]:
```

```

PREDICTION OF IMPACT OF DIFFERENT ELEMENTS WITHIN STABILITY FRAMEWORK

COMPARISON BETWEEN BRACED FRAMED TUBE AND OUTRIGGER SYSTEM BASED ON COSTS OR EMBODIED CARBON FOR EACH STRUCTURAL ELEMENT IN EACH HEIGHT ZONE

Emma Potijk (4715802)
MSc Thesis Structural Engineering
University: TU Delft
Company: Buro Happold
Period: 08/2023 - 05/2024

Import standard functions

In [ ]:
```
import pandas as pd
import numpy as np
from sklearn.model_selection import KFold
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_squared_error
from sklearn.neural_network import MLPRegressor
from geneticalgorithm import geneticalgorithm as ga
from sklearn.model_selection import train_test_split

import matplotlib.pyplot as plt
import seaborn as sns
sns.set(style="whitegrid")

import ipywidgets as widgets
from IPython.display import display
from ipywidgets import interact
import copy
import warnings
from mpl_toolkits.mplot3d import Axes3D

from sklearn.base import BaseEstimator
from keras.callbacks import EarlyStopping, ModelCheckpoint
from scipy.stats import norm
import math
```

In [ ]:
```
warnings.filterwarnings(action='ignore', category=UserWarning)
```

In [ ]:
```
color_orange = '#FF914D'
color_lightblue = '#38B6FF'
color_green = '#589D62'
color_purple = '#CB6CE6'
color_darkblue = '#417DB9'
color_yellow = '#FFF3C2'
color_grey = '#A6A6A6'
color_lightgreen = '#8BE198'
```

--- CONTENT ---

1) Data preprocessing

2) Modelling procedure

2a) Braced Framed Tube
2b) Outrigger

3) Make predictions

3a) Braced Framed Tube
3b) Outrigger

--- CODE ---

1) DATA PREPROCESSING

S1 = Braced Framed Tube System (grid size = 3.6 m)

S2 = Outrigger System (grid size = 1.8 m)

IMPORT DATA

i) S1: Braced Framed Tube

In [ ]:
```
file_path = r'C:\Users\epotijk\OneDrive - Buro Happold\MSc Thesis\Data Sets GH\Colibri_Braced_Framed_Tube\data.csv'
dataset_S1 = pd.read_csv(file_path, delimiter = ',')

dataset_S1 = dataset_S1.drop(columns = ['img'])
dataset_S1 = dataset_S1.loc[(dataset_S1 >= 0).all(axis=1)]
```

In [ ]:
```
dataset_S1
```

In [ ]:
```
# Assuming 'df' is your DataFrame

ranges = [(2, 8), (8, 14), (14, 20), (20, 26)]

def check_descending(row):
    for start, end in ranges:
        for i in range(end-1, start-1, -1):  # Iterate in reverse order
            if i > start and row[i] > row[i-1]:  # Check if values are not descending
                row[i-1] = row[i]  # Replace smaller value with larger one
    return row

dataset_S1_v1 = dataset_S1.copy()
dataset_S1 = dataset_S1_v1.apply(check_descending, axis=1)
dataset_S1
```

In [ ]:
```
def check_descending(row):
    for start, end in ranges:
        if not all(row[i] >= row[i+1] for i in range(start, end-1)):
            return False
    return True

is_descending = dataset_S1.apply(check_descending, axis=1)
print(is_descending)
```

ii) S1: Outrigger

In [ ]:
```
file_path = r'C:\Users\epotijk\OneDrive - Buro Happold\MSc Thesis\Data Sets GH\Colibri_Outrigger_DivisionColumns_v3\data.csv'
dataset_S2 = pd.read_csv(file_path, delimiter = ',')

dataset_S2 = dataset_S2.drop(columns = ['img'])
dataset_S2 = dataset_S2.loc[(dataset_S2 >= 0).all(axis=1)]
```

In [ ]:
```
dataset_S2 = dataset_S2.copy()
```

In [ ]:
```
mask = dataset_S2['out:Displacement Framework [cm]'] > dataset_S2['out:Max allowed displacement [cm]'] + 3
dataset_S2 = dataset_S2[~mask]
```

In [ ]:
```
dataset_S2.replace(-999.000000, 0.000000)
indices_to_drop6 = dataset_S2[dataset_S2['in:Number of grids'] == 6].index
indices_to_drop8 = dataset_S2[dataset_S2['in:Number of grids'] == 8].index

dataset_S2.drop(indices_to_drop6, inplace=True)
dataset_S2.drop(indices_to_drop8, inplace=True)
```

CONVERTING INPUT FEATURES

Geometry Variables

In [ ]:
```
#Convertion into length instead of number of grids or number of floors
grid_size_S1 = 3.6
grid_size_S2 = 1.8
floor_to_floor = 3.0

dataset_S1['in:Number of grids'] = dataset_S1['in:Number of grids'] * grid_size_S1
dataset_S1['in:Number of floors'] = dataset_S1['in:Number of floors'] * floor_to_floor

dataset_S2['in:Number of grids'] = dataset_S2['in:Number of grids'] * grid_size_S2
dataset_S2['in:Number of floors'] = dataset_S2['in:Number of floors'] * floor_to_floor

dataset_S1 = dataset_S1.rename(columns={'in:Number of grids' : 'in:Length [m]'})
dataset_S1 = dataset_S1.rename(columns={'in:Number of floors' : 'in:Height [m]'})

dataset_S2 = dataset_S2.rename(columns={'in:Number of grids' : 'in:Length [m]'})
dataset_S2 = dataset_S2.rename(columns={'in:Number of floors' : 'in:Height [m]'})
```

In [ ]:
```
mass_S1 = dataset_S1.copy()
mass_S2 = dataset_S2.copy()
```

Dropping additional floor beams columns since floor beams are equal over total height (equal every height zone)

In [ ]:
```
mass_S1 = mass_S1.drop(mass_S1.columns[21:26], axis = 1)
mass_S2 = mass_S2.drop(mass_S2.columns[27:32], axis = 1)
```

In [ ]:
```
mass_S2
```

Performance Parameters Variables

Costs

In [ ]:
```
#Prefab Concrete Reinforced
pc_rc_price_area = 850                              # concrete price [€/m3]
density_pc_rc = 2500                                # density prefab reinforced concrete [kg/m3]
pc_rc_price = pc_rc_price_area / density_pc_rc      # concrete price [€/kg]

#Hollow Core
hc_price_area = 110                                 # hollow core slab price (h=320mm) [€/m2]
density_hc = 429                                    # hollow core slab mass per area (h=320mm) [kg/m2]
hc_price = hc_price_area / density_hc               # hollow core slab price (h=320mm) [€/kg]

#Steel
s_price = 3.5                                       # steel price S355 [€/kg]


print('Price: prefab reinforced concrete =', round(pc_rc_price,3), '€/kg')
print('Price: hollow core concrete =', round(hc_price,3), '€/kg')
print('Price: steel =', round(s_price,3), '€/kg')
```

Embodied Carbon

In [ ]:
```
CO2_sprice = 0.05         # shaduw price CO2 eq [€/kgCO2eq]

pc_rc_EmCO2 = 0.238       # EmCO2 precast reinforced [kgCO2eq/kg]
hc_EmCO2 = 0.155          # EmCO2 hollow core [kgCO2eq/kg]
s_EmCO2 = 1.12            # EmCO2 steel S355 [kgCO2eq/kg]

print('Embodied Carbon: prefab reinforced concrete =', round(pc_rc_EmCO2,3), 'kgCO2eq/kg')
print('Embodied Carbon: hollow core concrete =', round(hc_EmCO2,3), 'kgCO2eq/kg')
print('Embodied Carbon: steel =', round(s_EmCO2,3), 'kgCO2eq/kg')
```

Floor System

In [ ]:
```
floor_mass = mass_S2.iloc[1, 3]
costs_floors = floor_mass * hc_price
costs_floors = round(costs_floors,2)
print('The costs of the floor sytem (hollow core slab) is', costs_floors, '€/m2-GFA')

EmCO2_floors = floor_mass * hc_EmCO2 * CO2_sprice
EmCO2_floors = round(EmCO2_floors,2)
print('The EmCO2 of the floor sytem (hollow core slab) is', EmCO2_floors, '€/m2-GFA')
```

2a) MODELLING PROCEDURE : BRACED FRAMED TUBE

SPLITTING AND SCALING DATA

In [ ]:
```
# Assuming df is your DataFrame and X, y are features and target variable
# Clean data and split into features (X) and target variable (y)

X = mass_S1[['in:Length [m]', 'in:Height [m]']]
y = mass_S1.iloc[:, 8:28]
```

In [ ]:
```
min_OB = np.min(y.iloc[:,0])
print("Minimum value of OB", min_OB)

min_OC = np.min(y.iloc[:,6])
print("Minimum value of OC", min_OC)

min_FB = np.min(y.iloc[:,12])
print("Minimum value of FB", min_FB)

min_IC = np.min(y.iloc[:,13])
print("Minimum value of IC", min_IC)
```

Splitting data into training and test sets & scaling the sets

In [ ]:
```
# Split data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
X_train_df = pd.DataFrame(X_train, columns=['in:Length [m]', 'in:Height [m]'])
X_test_df = pd.DataFrame(X_test, columns=['in:Length [m]', 'in:Height [m]'])

scaler = MinMaxScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_scaled = scaler.transform(X)
X_test_scaled = scaler.transform(X_test_df)
```

k-fold cross-validation

In [ ]:
```
# Split data with k-fold cross-validation
kf = KFold(n_splits=5, shuffle=True, random_state=42)

for train_index, val_index in kf.split(X_train_df):
    X_train_fold, X_val_fold = X_train_df.iloc[train_index], X_train_df.iloc[val_index]
    y_train_fold, y_val_fold = y_train.iloc[train_index], y_train.iloc[val_index]

X_val_fold_scaled = scaler.transform(X_val_fold)
X_train_fold_scaled = scaler.transform(X_train_fold)
```

Check the shape of the input data

In [ ]:
```
class ANNModel_S1(BaseEstimator):
    def __init__(self, neurons, kernel_initializer, bias_initializer, alpha=0.01, learning_rate_init=0.01):
        self.neurons = neurons
        self.alpha = alpha
        self.learning_rate_init = learning_rate_init
        self.kernel_initializer = kernel_initializer
        self.bias_initializer = bias_initializer

        self.model = MLPRegressor(
            hidden_layer_sizes=(neurons, neurons),
            activation='relu',
            solver='adam',
            max_iter=10000,
            n_iter_no_change=10,
            alpha=self.alpha,
            learning_rate_init=self.learning_rate_init,
            tol=1e-4,
            verbose=False,
            warm_start=False
        )

        self.losses = {'training_loss': [], 'validation_loss': []}

    def partial_fit(self, X, y):
        self.model.partial_fit(X, y)

    def fit(self, X, y, X_val=None, y_val=None):
        if X_val is not None and y_val is not None:
            self.model.validation_set = (X_val, y_val)
        else:
            self.model.validation_set = None

        for epoch in range(self.model.max_iter):
            self.model.partial_fit(X, y)
            training_loss = self.model.loss_
            self.losses['training_loss'].append(training_loss)

            if self.model.validation_set is not None:
                val_predictions = self.model.predict(X_val)
                val_loss = np.mean((val_predictions - y_val) ** 2, axis=0)
                self.losses['validation_loss'].append(np.mean(val_loss))

            if self.model.n_iter_no_change is not None and len(self.losses['training_loss']) > self.model.n_iter_no_change:
                recent_losses = self.losses['training_loss'][-self.model.n_iter_no_change:]
                if max(recent_losses) - min(recent_losses) < self.model.tol:
                    break

    def determine_wind_zones(self, X):
        if isinstance(X, pd.DataFrame):
            b = X.iloc[:, 0].values
            h = X.iloc[:, 1].values
        else:
            b = X[:, 0]
            h = X[:, 1]

        wind_zones = np.zeros_like(b, dtype=int)

        for i in range(len(b)):
            if h[i] <= b[i]:
                wind_zones[i] = 1
            elif b[i] <= h[i] < 2 * b[i]:
                wind_zones[i] = 2
            elif h[i] > 2 * b[i]:
                wind_zones[i] = 3

        return wind_zones

    def determine_h_wind_zones(self, wind_zones, X):
        if isinstance(X, pd.DataFrame):
            b = X.iloc[:, 0].values
            h = X.iloc[:, 1].values
        else:
            b = X[:, 0]
            h = X[:, 1]

        z_e1 = np.zeros_like(wind_zones, dtype=float)
        z_e2 = np.zeros_like(wind_zones, dtype=float)
        z_e3 = np.zeros_like(wind_zones, dtype=float)

        floor_to_floor = 3

        for i, zone in enumerate(wind_zones):
            if zone == 1:
                z_e1[i] = h[i]
                z_e2[i] = 0
                z_e3[i] = 0
            elif zone == 2:
                z_e1[i] = b[i]
                z_e2[i] = h[i]
                z_e3[i] = 0
            elif zone == 3:
                z_e1[i] = (b[i] // floor_to_floor) * floor_to_floor
                z_e2[i] = h[i] - ((b[i] // floor_to_floor) * floor_to_floor)
                z_e3[i] = h[i]

        z_e1 = np.round(z_e1, 1)
        z_e2 = np.round(z_e2, 1)
        z_e3 = np.round(z_e3, 1)


        height_wind_zones = np.column_stack((z_e1, z_e2, z_e3))
        return height_wind_zones
```

```
    def determine_height_zones(self, wind_zones, X, height_wind_zones):
        num_zones = np.zeros_like(wind_zones, dtype=float)

        def distribute_floors(total_floors, num_zones):
            if num_zones == 0:
                return [], 0

            base_floors_per_zone = total_floors // num_zones
            remaining_floors = total_floors % num_zones

            floors_per_zone = [base_floors_per_zone] * num_zones

            for i in range(int(remaining_floors)):
                floors_per_zone[i] += 1

            return floors_per_zone, len(floors_per_zone)

        for i in range(len(wind_zones)):
            if wind_zones[i] == 1:
                num_zones[i] = 1

            elif wind_zones[i] == 2:
                num_zones[i] = 2

            elif wind_zones[i] == 3:
                floors_2 = height_wind_zones[i, 1] - height_wind_zones[i, 0]
                ratio = (height_wind_zones[i, 1] - height_wind_zones[i, 0]) / height_wind_zones[i, 0]

                if ratio == 1.5:
                    ratio = 2
                if ratio == 3.5:
                    ratio = 4
                if ratio == 2.5:
                    ratio = 3
                else:
                    ratio = round(ratio)
                    if ratio > 4:
                        ratio = 4
                    else:
                        floors_distribution_windzone2, outcome_length = distribute_floors(floors_2, int(ratio))

                if ratio <= 1:
                    num_zones[i] = 3

                elif ratio > 1:
                    additional_zones = ratio
                    additional_zones = min(additional_zones, 4)
                    num_zones[i] = additional_zones + 2

        return num_zones

    def predict(self, X, X_unscaled):
        wind_zones = self.determine_wind_zones(X_unscaled)
        height_wind_zones = self.determine_h_wind_zones(wind_zones, X_unscaled)
        num_zones = self.determine_height_zones(wind_zones, X_unscaled, height_wind_zones)

        predictions = self.model.predict(X)

        predictions[predictions < 0] = 0

        ranges = [(0, 6), (6, 12), (12, 13), (13, 19)]

        for start, end in ranges:
            for i in range(predictions.shape[0]):
                mask = (predictions[i, start:end] == 0)
                indices = np.where(mask)[0]
                if len(indices) > 0:
                    if start == 0:
                        predictions[i, start:end][indices[:6]] = min_OB
                    elif start == 6:
                        predictions[i, start:end][indices[:12]] = min_OC
                    elif start == 12:
                        predictions[i, start:end][indices[:13]] = min_FB
                    elif start == 13:
                        predictions[i, start:end][indices[:19]] = min_IC

        slenderness = None
        if isinstance(X_unscaled, pd.DataFrame):
            slenderness = X_unscaled.iloc[:, 1] / X_unscaled.iloc[:, 0]
        else:
            slenderness = X_unscaled[:, 1] / X_unscaled[:, 0]

        predictions[slenderness > 11.67] = np.nan

        predicted_num_zones = num_zones

        height_zone_masks = np.arange(1, 7).reshape(1, -1) <= predicted_num_zones.reshape(-1,1)
        height_zone_masks_expanded = height_zone_masks[:, :predictions.shape[1]]

        unused_height_zone_columns_6 = [5, 11, 18]
        unused_height_zone_columns_5 = [4, 5, 10, 11, 17, 18]
        unused_height_zone_columns_4 = [3, 4, 5, 9, 10, 11, 16, 17, 18]
        unused_height_zone_columns_3 = [2, 3, 4, 5, 8, 9, 10, 11, 15, 16, 17, 18]
        unused_height_zone_columns_2 = [1, 2, 3, 4, 5, 7, 8, 9, 10, 11, 14, 15, 16, 17, 18]

        for i, num_zone in enumerate(num_zones):
            if num_zone == 5:
                predictions[i, unused_height_zone_columns_6] = 0
            if num_zone == 4:
                predictions[i, unused_height_zone_columns_5] = 0
            if num_zone == 3:
```

```
                predictions[i, unused_height_zone_columns_4] = 0
            if num_zone == 2:
                predictions[i, unused_height_zone_columns_3] = 0
            if num_zone == 1:
                predictions[i, unused_height_zone_columns_2] = 0

        for i in range(predictions.shape[0]):
            for start, end in ranges:
                for j in range(start+1, end):
                    if predictions[i, j] > predictions[i, j-1]:
                        predictions[i, j] = predictions[i, j-1]

        return predictions, num_zones
```

DEFINE FITNESS MODEL AND SET UP GA

In [ ]:
```
# Define the fitness function
def fitness_func(solution):
    neurons = int(solution[0])
    kernel_initializer_option = solution[1]
    bias_initializer_option = solution[2]
    alpha_option = solution[3]
    learning_rate_option = solution[4]

    alpha = 0.01 if alpha_option == 1 else 0.001
    learning_rate = 0.01 if learning_rate_option == 1 else 0.001
    kernel_initializer = 'random_uniform' if kernel_initializer_option == 0 else 'constant' if kernel_initializer_option == 1 else 'zeros'
    bias_initializer = 'random_uniform' if bias_initializer_option == 0 else 'constant' if bias_initializer_option == 1 else 'zeros'

    # Create and train the model
    model = ANNModel_S1(neurons, alpha=alpha, learning_rate_init=learning_rate,
                        kernel_initializer=kernel_initializer, bias_initializer=bias_initializer)

    model.fit(X_train_scaled, y_train, X_val_fold_scaled, y_val_fold)

    # Predict on the validation set
    y_pred_fold, num_zones = model.predict(X_val_fold_scaled, X_val_fold)

    # Calculate mean squared error
    mse = mean_squared_error(y_val_fold, y_pred_fold)

    # Print the neurons and mse value
    print("Neurons:", neurons, "Alpha:", alpha, "Learning Rate:", learning_rate,
          "Kernel Initializer:", kernel_initializer, "Bias Initializer", bias_initializer)
    print("MSE:", mse)

    return mse
```

Define the parameters bounds and algorithm parameters for GA

In [ ]:
```
# Define the parameter bounds for the genetic algorithm
varbound = np.array([[1, 50], [0, 2], [0, 2], [0, 1], [0, 1]])

# Define the algorithm parameters
algorithm_param = {
    'max_num_iteration': 30,
    'population_size': 100,
    'mutation_probability': 0.1,
    'parents_portion': 0.3,
    'elit_ratio': 0.1,
    'crossover_probability': 0.8,
    'crossover_type': 'uniform',
    'max_iteration_without_improv': 3
}

# Create the genetic algorithm
modelGA = ga(function=fitness_func, dimension=5, variable_type='int', variable_boundaries=varbound,
             algorithm_parameters=algorithm_param, function_timeout = 100)

# Run the genetic algorithm
modelGA.run()
```

Run ANN model on the test data with the best parameters

In [ ]:
```
#Neurons: 45 Alpha: 0.01 Learning Rate: 0.01 MSE: 29.59881285138531
# Get the best solution found by the genetic algorithm
optimum_solution = modelGA.output_dict['variable']
best_fitness = modelGA.output_dict['function']

# Extract the values for number of neurons, alpha, and learning rate from the solution
optimum_neurons = int(optimum_solution[0])
optimum_alpha = 0.01 if optimum_solution[3] == 1 else 0.001
optimum_learning_rate = 0.01 if optimum_solution[4] == 1 else 0.001
optimum_kernel_initializer = 'random_uniform' if optimum_solution[1] == 0 else 'constant' if optimum_solution[1] == 1 else 'zeros'
optimum_bias_initializer = 'random_uniform' if optimum_solution[2] == 0 else 'constant' if optimum_solution[2] == 1 else 'zeros'

print("Best solution (number of neurons):", optimum_neurons)
print("Best alpha option:", optimum_alpha)
print("Best learning rate option:", optimum_learning_rate)
print("Best kernel initializer", optimum_kernel_initializer)
print("Best bias initializer", optimum_bias_initializer)
print("Corresponding fitness (MSE):", best_fitness)

# Create a new instance of the ANN model with the best parameters
best_model_S1 = ANNModel_S1(neurons=optimum_neurons, kernel_initializer=optimum_kernel_initializer,
                        bias_initializer=optimum_bias_initializer,alpha=optimum_alpha,
                        learning_rate_init=optimum_learning_rate)
```

Calculate the Mean Squared Error for the validation fold with the best parameters

In [ ]:
```
#best_model_S1 = ANNModel_S1(neurons=47, kernel_initializer='constant', bias_initializer='zeros',
#                         alpha=0.01, learning_rate_init=0.01)

best_model_S1 = ANNModel_S1(neurons=49, kernel_initializer='zeros',
                        bias_initializer='random_uniform',alpha=0.01,
                        learning_rate_init=0.01)

# Train the model on the current fold
best_model_S1.fit(X_train_scaled, y_train, X_val_fold_scaled, y_val_fold)

# Predict on the test set for this fold
y_pred_fold, num_zones = best_model_S1.predict(X_val_fold_scaled, X_val_fold)

# Calculate mean squared error for this fold
mse_fold = mean_squared_error(y_val_fold, y_pred_fold)

print("Mean Squared Error for this fold:", mse_fold)
```

In [ ]:
```
X_train_fold.reset_index(drop=True, inplace=True)
y_train_fold.reset_index(drop=True, inplace=True)
```

Calculate MSE

In [ ]:
```
# Calculate mean squared error for this fold
y_pred_val, num_zones = best_model_S1.predict(X_val_fold_scaled, X_val_fold)
mse_valfold = mean_squared_error(y_val_fold, y_pred_val)
print(mse_valfold)
```

In [ ]:
```
# Calculate mean squared error for the test data
y_pred_test, num_zones = best_model_S1.predict(X_test_scaled, X_test)
mse_test = mean_squared_error(y_test, y_pred_test)
print(mse_test)
```

COMPARING MODEL WITH TEST & TRAIN DATA

Actual test data and predicted test data based on the scaled input data (X test)

In [ ]:
```
X_test_array = np.array(X_test)
X_test_array_pd = pd.DataFrame(X_test_array)

y_test_array = np.array(y_test)
y_test_array_pd = pd.DataFrame(y_test_array)

y_pred_test, num_zones = best_model_S1.predict(X_test_scaled, X_test)
y_pred_test_pd = pd.DataFrame(y_pred_test)
```

Difference between the predicted test data and the actual test data

In [ ]:
```
diff_test = y_test_array - y_pred_test
diff_test_pd = pd.DataFrame(diff_test)
```

Difference in percentage between the predicted test data and the actual data

In [ ]:
```
diff_percentage = (diff_test / y_test_array) * 100
diff_percentage = np.array(diff_percentage)
diff_percentage = pd.DataFrame(diff_percentage)
diff_percentage = diff_percentage.fillna(0)
```

In [ ]:
```
diff_percentage = diff_percentage.values.ravel()
```

Illustrating the distribution of the absolute percentages between actual and predicted values (test data)

In [ ]:
```
# Create bins for the histogram
bins = np.arange(-101, 101, 2)  # Adjust the range and width of bins as needed

# Create the histogram
plt.hist(diff_percentage, bins=bins, color=color_lightgreen, density=True)

# Calculate the mean and standard deviation of your data
mu = np.mean(diff_percentage)
std = np.std(diff_percentage)

# Generate the x values for your normal distribution
xmin, xmax = plt.xlim()
x = np.linspace(xmin, xmax, 100)

# Generate the y values (probability density function) for your normal distribution
p = norm.pdf(x, mu, std)

# Plot the normal distribution
plt.plot(x, p, 'k', linewidth=2, color=color_green)

# Add labels and title
plt.xlabel('Relative error between Actual and Predicted Values')
plt.ylabel('Probability Density')
plt.title('Distribution of relative error of Braced Framed Tube (S1)')

# Show the plot
plt.show()

#rmse_ANN = np.sqrt(np.mean((y_test_array - y_pred_test)**2))
#print("The RMSE is equal to", round(rmse_ANN,3), 'kg')
```

ANALYSIS OF PREDICTIONS

RMSE for each output

In [ ]:
```
n_outputs = y_test_array_pd.shape[1]  # Get the number of outputs
rmse_list_S1 = []

for i in range(n_outputs):
    y_test_i = y_test_array_pd.iloc[:, i]  # Get the true values for the i-th output
    y_pred_i = y_pred_test_pd.iloc[:, i]  # Get the predicted values for the i-th output

    rmse_i = np.sqrt(mean_squared_error(y_test_i, y_pred_i))  # Calculate RMSE for the i-th output
    rmse_list_S1.append(rmse_i)  # Append the RMSE to the list

# Now rmse_list contains the RMSE for each output separately
print("RMSE for each output separately:")
for i, rmse_i in enumerate(rmse_list_S1):
    print(f"Output {i + 1}: {rmse_i}")
```

MAPE for each output

In [ ]:
```
# Define lists to store MAPE values for each output
mape_list_S1 = []

# Iterate over each output
for i in range(n_outputs):
    # Get the true values for the i-th output
    y_test_i = y_test_array_pd.iloc[:, i]

    # Get the predicted values for the i-th output
    y_pred_i = y_pred_test_pd.iloc[:, i]

    # Calculate absolute errors
    abs_errors = np.abs(y_test_i - y_pred_i)

    # Calculate percentage errors
    perc_errors = abs_errors / y_test_i * 100

    # Calculate mean absolute percentage error (MAPE) for the i-th output
    mape_i = np.mean(perc_errors)

    # Append the MAPE to the list
    mape_list_S1.append(mape_i)

# Print the MAPE for each output separately
print("Mean Absolute Percentage Error (MAPE) for each output separately:")
for i, mape_i in enumerate(mape_list_S1):
    print(f"Output {i + 1}: {mape_i}")

mean_mape = np.mean(mape_list_S1)

print(mean_mape)
```

Plot of convergence

In [ ]:
```
fig, axes = plt.subplots(1, 2, figsize=(12, 5))  # Create a figure with 1 row and 2 columns

# Plot the original plot on the first subplot
axes[0].plot(best_model_S1.losses['training_loss'], label='Training Loss', color = color_darkblue)
axes[0].plot(best_model_S1.losses['validation_loss'], label='Validation Loss',  color = color_orange)
axes[0].set_title('Convergence Plot for Braced Framed Tube (S1)')
axes[0].set_xlabel('Epochs')
axes[0].set_ylabel('Loss [MSE]')
axes[0].legend()

ymax = np.max(best_model_S1.losses['validation_loss'])

axes[0].set_ylim(bottom=-50, top=ymax+100)

# Plot the zoomed-in plot on the second subplot
axes[1].plot(best_model_S1.losses['training_loss'], label='Training Loss', color = color_darkblue)
axes[1].plot(best_model_S1.losses['validation_loss'], label='Validation Loss', color = color_orange)
axes[1].set_title('Zoomed-in Convergence Plot for Braced Framed Tube (S1)')
axes[1].set_xlabel('Epochs')
axes[1].set_ylabel('Loss [MSE]')
axes[1].legend()

# Set limits for better visualization on the zoomed-in plot
#axes[1].set_xlim(left=0-50, right=1500)
axes[1].set_xlim(left=17500, right=20000)
axes[1].set_ylim(bottom=-50, top=500)
#axes[1].set_ylim(bottom=-50, top=ymax+100)

plt.tight_layout()
plt.show()
```

2b) MODELLING PROCEDURE: Outrigger

SPLITTING AND SCALING DATA

In [ ]:
```
mass_S2.iloc[:, 8:41]
```

In [ ]:
```
# Assuming df is your DataFrame and X, y are features and target variable
# Clean data and split into features (X) and target variable (y)

X2 = mass_S2[['in:Length [m]', 'in:Height [m]']]
y2 = mass_S2.iloc[:, 8:41]
```

In [ ]:
```
min_OB2 = y2.iloc[:, 0:6].min(axis=0)
min_OB2 = min_OB2[0]
print("Minimum value of OB:", min_OB2)

min_OC_Out2 = y2.iloc[:, 6:12].min(axis=0)
min_OC_Out2 = min_OC_Out2[0]
print("Minimum value of OC connected to BellTruss:", min_OC_Out2)

min_OC_noOut2 = y2.iloc[:, 12:18].min(axis=0)
min_OC_noOut2 = min_OC_noOut2[0]
print("Minimum value of OC not connected to BellTruss:", min_OC_noOut2)

min_FB2 = y2.iloc[:, 18].min(axis=0)
min_FB2 = min_FB2
print("Minimum value of FB:", min_FB2)

min_IC_Out2 = y2.iloc[:, 19:25].min(axis=0)
min_IC_Out2 = min_IC_Out2[0]
print("Minimum value of IC connected to BellTruss:", min_IC_Out2)

min_IC_noOut2 = y2.iloc[:, 25:31].min(axis=0)
min_IC_noOut2 = min_IC_noOut2[0]
print("Minimum value of IC not connected to BellTruss:", min_IC_noOut2)
```

Splitting data into training and test sets & scaling the sets

In [ ]:
```
# Split data into training and test sets
X_train2, X_test2, y_train2, y_test2 = train_test_split(X2, y2, test_size=0.2, random_state=42)
X_train_df2 = pd.DataFrame(X_train2, columns=['in:Length [m]', 'in:Height [m]'])
X_test_df2 = pd.DataFrame(X_test2, columns=['in:Length [m]', 'in:Height [m]'])

scaler2 = MinMaxScaler()
X_train_scaled2 = scaler2.fit_transform(X_train_df2)
X_scaled2 = scaler2.transform(X2)
X_test_scaled2 = scaler2.transform(X_test_df2)
```

k-fold cross-validation

In [ ]:
```
# Split data with k-fold cross-validation
kf = KFold(n_splits=5, shuffle=True, random_state=42)

for train_index, val_index in kf.split(X_train_df2):
    X_train_fold2, X_val_fold2 = X_train_df2.iloc[train_index], X_train_df2.iloc[val_index]
    y_train_fold2, y_val_fold2 = y_train2.iloc[train_index], y_train2.iloc[val_index]

X_val_fold_scaled2 = scaler2.transform(X_val_fold2)
X_train_fold_scaled2 = scaler2.transform(X_train_fold2)
```

Check the shape of the input data

In [ ]:
```
print("Shape of X_train_fold:", X_train_fold2.shape)
print("Shape of y_train_fold:", y_train_fold2.shape)
print("Shape of y_train:", y_train2.shape)
print("Shape of X_train:", X_train_scaled2.shape)
#print("Index values of X_train_fold:", X_train_fold.index)
#print("Index values of y_train_fold:", y_train_fold.index)
print("Shape of y_val:", y_val_fold2.shape)
print("Shape of X_val_scaled:", X_val_fold_scaled2.shape)
```

In [ ]:
```
import math
```

In [ ]:
```
class ANNModel_S2(BaseEstimator):
    def __init__(self, neurons, kernel_initializer, bias_initializer, alpha=0.01, learning_rate_init=0.01):
        self.neurons = neurons
        self.alpha = alpha
        self.learning_rate_init = learning_rate_init
        self.kernel_initializer = kernel_initializer
        self.bias_initializer = bias_initializer

        self.model = MLPRegressor(
            hidden_layer_sizes=(neurons, neurons),
            activation='relu',
            solver='adam',
            max_iter=10000,
            n_iter_no_change=10,
            alpha=self.alpha,
            learning_rate_init=self.learning_rate_init,
            tol=1e-4,
            verbose=False,
            warm_start=False
        )

        self.losses = {'training_loss': [], 'validation_loss': []}

    def partial_fit(self, X, y):
        self.model.partial_fit(X, y)

    def fit(self, X, y, X_val=None, y_val=None):
        if X_val is not None and y_val is not None:
            self.model.validation_set = (X_val, y_val)
        else:
            self.model.validation_set = None

        for epoch in range(self.model.max_iter):
            self.model.partial_fit(X, y)
            training_loss = self.model.loss_
            self.losses['training_loss'].append(training_loss)

            if self.model.validation_set is not None:
                val_predictions = self.model.predict(X_val)
                val_loss = np.mean((val_predictions - y_val) ** 2, axis=0)
                self.losses['validation_loss'].append(np.mean(val_loss))

            if self.model.n_iter_no_change is not None and len(self.losses['training_loss']) > self.model.n_iter_no_change:
                recent_losses = self.losses['training_loss'][-self.model.n_iter_no_change:]
                if max(recent_losses) - min(recent_losses) < self.model.tol:
                    break

    def determine_wind_zones(self, X):
        if isinstance(X, pd.DataFrame):
            b = X.iloc[:, 0].values
            h = X.iloc[:, 1].values
        else:
            b = X[:, 0]
            h = X[:, 1]

        wind_zones = np.zeros_like(b, dtype=int)

        for i in range(len(b)):
            if h[i] <= b[i]:
                wind_zones[i] = 1
            elif b[i] <= h[i] < 2 * b[i]:
                wind_zones[i] = 2
            elif h[i] > 2 * b[i]:
                wind_zones[i] = 3

        return wind_zones

    def determine_h_wind_zones(self, wind_zones, X):
        if isinstance(X, pd.DataFrame):
            b = X.iloc[:, 0].values
            h = X.iloc[:, 1].values
        else:
            b = X[:, 0]
            h = X[:, 1]

        z_e1 = np.zeros_like(wind_zones, dtype=float)
        z_e2 = np.zeros_like(wind_zones, dtype=float)
        z_e3 = np.zeros_like(wind_zones, dtype=float)

        floor_to_floor = 3

        for i, zone in enumerate(wind_zones):
            if zone == 1:
                z_e1[i] = h[i]
                z_e2[i] = 0
                z_e3[i] = 0
            elif zone == 2:
                z_e1[i] = b[i]
                z_e2[i] = h[i]
                z_e3[i] = 0
            elif zone == 3:
                z_e1[i] = math.ceil(b[i] / floor_to_floor) * floor_to_floor
                z_e2[i] = h[i] - (math.ceil(b[i] / floor_to_floor) * floor_to_floor)
                z_e3[i] = h[i]

        z_e1 = np.round(z_e1, 1)
        z_e2 = np.round(z_e2, 1)
        z_e3 = np.round(z_e3, 1)


        height_wind_zones = np.column_stack((z_e1, z_e2, z_e3))
        return height_wind_zones
```

```
    def determine_height_zones(self, wind_zones, X, height_wind_zones):
        num_zones = np.zeros_like(wind_zones, dtype=float)

        def distribute_floors(total_floors, num_zones):
            if num_zones == 0:
                return [], 0

            base_floors_per_zone = total_floors // num_zones
            remaining_floors = total_floors % num_zones

            floors_per_zone = [base_floors_per_zone] * num_zones

            for i in range(int(remaining_floors)):
                floors_per_zone[i] += 1


            return floors_per_zone, len(floors_per_zone)

        for i in range(len(wind_zones)):
            if wind_zones[i] == 1:
                num_zones[i] = 1

            elif wind_zones[i] == 2:
                num_zones[i] = 2

            elif wind_zones[i] == 3:
                floors_2 = height_wind_zones[i, 1] - height_wind_zones[i, 0]
                ratio = (height_wind_zones[i, 1] - height_wind_zones[i, 0]) / height_wind_zones[i, 0]

                if ratio == 1.5:
                    ratio = 2
                if ratio == 3.5:
                    ratio = 4
                if ratio == 2.5:
                    ratio = 3
                else:
                    ratio = round(ratio)
                    if ratio > 4:
                        ratio = 4
                    else:
                        floors_distribution_windzone2, outcome_length = distribute_floors(floors_2, int(ratio))

                if ratio <= 1:
                    num_zones[i] = 3

                elif ratio > 1:
                    additional_zones = ratio
                    additional_zones = min(additional_zones, 4)
                    num_zones[i] = additional_zones + 2

        return num_zones

    def predict(self, X, X_unscaled):
        wind_zones = self.determine_wind_zones(X_unscaled)
        height_wind_zones = self.determine_h_wind_zones(wind_zones, X_unscaled)
        num_zones = self.determine_height_zones(wind_zones, X_unscaled, height_wind_zones)

        predictions = self.model.predict(X)

        predictions[predictions < 0] = 0

        ranges = [(0, 6), (6, 12), (12, 18), (18, 19), (19, 25), (25, 31)]

        for start, end in ranges:
            for i in range(predictions.shape[0]):
                mask = (predictions[i, start:end] == 0)
                indices = np.where(mask)[0]
                if len(indices) > 0:
                    if start == 0:
                        predictions[i, start:end][indices[:6]] = min_OB2
                    elif start == 6:
                        predictions[i, start:end][indices[:12]] = min_OC_Out2
                    elif start == 12:
                        predictions[i, start:end][indices[:18]] = min_OC_noOut2
                    elif start == 18:
                        predictions[i, start:end][indices[:19]] = min_FB2
                    elif start == 19:
                        predictions[i, start:end][indices[:25]] = min_IC_Out2
                    elif start == 25:
                        predictions[i, start:end][indices[:31]] = min_IC_noOut2

        predicted_num_zones = num_zones

        height_zone_masks = np.arange(1, 7).reshape(1, -1) <= predicted_num_zones.reshape(-1,1)
        height_zone_masks_expanded = height_zone_masks[:, :predictions.shape[1]]

        unused_height_zone_columns_6 = [5, 11, 17, 24, 30]
        unused_height_zone_columns_5 = [4, 5, 10, 11, 16, 17, 23, 24, 29, 30]
        unused_height_zone_columns_4 = [3, 4, 5, 9, 10, 11, 15, 16, 17, 22, 23, 24, 28, 29, 30]
        unused_height_zone_columns_3 = [2, 3, 4, 5, 8, 9, 10, 11, 14, 15, 16, 17, 21, 22, 23, 24, 27, 28, 29, 30]
        unused_height_zone_columns_2 = [1, 2, 3, 4, 5, 7, 8, 9, 10, 11, 13, 14, 15, 16, 17, 20, 21, 22, 23, 24,
                                        26, 27, 28, 29, 30]

        for i, num_zone in enumerate(num_zones):
            if num_zone == 5:
                predictions[i, unused_height_zone_columns_6] = 0
            if num_zone == 4:
                predictions[i, unused_height_zone_columns_5] = 0
            if num_zone == 3:
                predictions[i, unused_height_zone_columns_4] = 0
            if num_zone == 2:
```

```
                predictions[i, unused_height_zone_columns_3] = 0
            if num_zone == 1:
                predictions[i, unused_height_zone_columns_2] = 0

        for i in range(predictions.shape[0]):
            for start, end in ranges:
                for j in range(start+1, end):
                    if predictions[i, j] > predictions[i, j-1]:
                        predictions[i, j] = predictions[i, j-1]

        return predictions, num_zones
```

DEFINE THE FITNESS FUNCTION AND SET UP GA

In [ ]:
```
# Define the fitness function
def fitness_func(solution):
    neurons = int(solution[0])
    kernel_initializer_option = solution[1]
    bias_initializer_option = solution[2]
    alpha_option = solution[3]
    learning_rate_option = solution[4]

    alpha = 0.01 if alpha_option == 1 else 0.001
    learning_rate = 0.01 if learning_rate_option == 1 else 0.001
    kernel_initializer = 'random_uniform' if kernel_initializer_option == 0 else 'constant' if kernel_initializer_option == 1 else 'zeros'
    bias_initializer = 'random_uniform' if bias_initializer_option == 0 else 'constant' if bias_initializer_option == 1 else 'zeros'

    # Create and train the model
    model2 = ANNModel_S2(neurons, alpha=alpha, learning_rate_init=learning_rate,
                         kernel_initializer=kernel_initializer, bias_initializer=bias_initializer)

    model2.fit(X_train_scaled2, y_train2, X_val_fold_scaled2, y_val_fold2)

    # Predict on the validation set
    y_pred_fold2, num_zones = model2.predict(X_val_fold_scaled2, X_val_fold2)

    # Calculate mean squared error
    mse2 = mean_squared_error(y_val_fold2, y_pred_fold2)

    # Print the neurons and mse value
    print("Neurons:", neurons, "Alpha:", alpha, "Learning Rate:", learning_rate,
          "Kernel Initializer:", kernel_initializer, "Bias Initializer", bias_initializer)
    print("MSE:", mse2)

    return mse2
```

Define the parameters bounds and algorithm parameters for GA

In [ ]:
```
# Define the parameter bounds for the genetic algorithm
varbound = np.array([[1, 50], [0, 2], [0, 2], [0, 1], [0, 1]])

# Define the algorithm parameters
algorithm_param = {
    'max_num_iteration': 30,
    'population_size': 100,
    'mutation_probability': 0.1,
    'parents_portion': 0.3,
    'elit_ratio': 0.1,
    'crossover_probability': 0.8,
    'crossover_type': 'uniform',
    'max_iteration_without_improv': 3
}

# Create the genetic algorithm
modelGA = ga(function=fitness_func, dimension=5, variable_type='int', variable_boundaries=varbound,
             algorithm_parameters=algorithm_param, function_timeout = 1000)

# Run the genetic algorithm
modelGA.run()
```

Run ANN model on the test data with the best parameters

In [ ]:
```
# Get the best solution found by the genetic algorithm
optimum_solution = modelGA.output_dict['variable']
best_fitness = modelGA.output_dict['function']

# Extract the values for number of neurons, alpha, and learning rate from the solution
optimum_neurons = int(optimum_solution[0])
optimum_alpha = 0.01 if optimum_solution[3] == 1 else 0.001
optimum_learning_rate = 0.01 if optimum_solution[4] == 1 else 0.001
optimum_kernel_initializer = 'random_uniform' if optimum_solution[1] == 0 else 'constant' if optimum_solution[1] == 1 else 'zeros'
optimum_bias_initializer = 'random_uniform' if optimum_solution[2] == 0 else 'constant' if optimum_solution[2] == 1 else 'zeros'

print("Best solution (number of neurons):", optimum_neurons)
print("Best alpha option:", optimum_alpha)
print("Best learning rate option:", optimum_learning_rate)
print("Best kernel initializer", optimum_kernel_initializer)
print("Best bias initializer", optimum_bias_initializer)
print("Corresponding fitness (MSE):", best_fitness)

# Create a new instance of the ANN model with the best parameters
best_model_S2 = ANNModel_S2(neurons=optimum_neurons, kernel_initializer=optimum_kernel_initializer,
                        bias_initializer=optimum_bias_initializer,alpha=optimum_alpha,
                        learning_rate_init=optimum_learning_rate)
```

Calculate the Mean Squared Error for the validation with the best parameters

In [ ]:
```
#Neurons: 37 Alpha: 0.001 Learning Rate: 0.01 Kernel Initializer: zeros Bias Initializer constant
# MSE: 11.351525442185526

best_model_S2 = ANNModel_S2(neurons=47, alpha=0.01,
                        kernel_initializer='zeros', bias_initializer='random_uniform',
                        learning_rate_init=0.01)

#best_model_S2 = ANNModel_S2(neurons=37, alpha=0.001,
#                        kernel_initializer='zeros', bias_initializer='constant',
#                        learning_rate_init=0.01)

# Train the model on the current fold
best_model_S2.fit(X_train_scaled2, y_train2, X_val_fold_scaled2, y_val_fold2)

# Predict on the test set for this fold
y_pred_fold2, num_zones = best_model_S2.predict(X_val_fold_scaled2, X_val_fold2)

# Calculate mean squared error for this fold
mse_fold2 = mean_squared_error(y_val_fold2, y_pred_fold2)

print("Mean Squared Error for this fold:", mse_fold2)
```

In [ ]:
```
X_train_fold2.reset_index(drop=True, inplace=True)
y_train_fold2.reset_index(drop=True, inplace=True)
```

Calculate the MSE

In [ ]:
```
# Calculate mean squared error for this fold
y_pred_val2, num_zones = best_model_S2.predict(X_val_fold_scaled2, X_val_fold2)
mse_valfold2 = mean_squared_error(y_val_fold2, y_pred_val2)
print(mse_valfold2)
```

In [ ]:
```
# Calculate mean squared error for the test data
y_pred_test2, num_zones = best_model_S2.predict(X_test_scaled2, X_test2)
mse_test2 = mean_squared_error(y_test2, y_pred_test2)
print(mse_test2)
```

COMPARING MODEL WITH TEST & TRAIN DATA

Actual test data and predicted test data based on the scaled input data (X test)

In [ ]:
```
X_test_array = np.array(X_test2)
X_test_array_pd = pd.DataFrame(X_test_array)
X_test_array_pd

y_test_array = np.array(y_test2)
y_test_array_pd = pd.DataFrame(y_test_array)
y_test_array_pd

y_pred_test, num_zones = best_model_S2.predict(X_test_scaled2, X_test2)
y_pred_test_pd = pd.DataFrame(y_pred_test)
y_pred_test_pd
```

Difference between the predicted test data and the actual test data

In [ ]:
```
pd.set_option('display.max_rows', None)
pd.set_option('display.max_columns', None)
```

In [ ]:
```
diff_test = y_pred_test - y_test_array
diff_test_pd = pd.DataFrame(diff_test)
```

Difference in percentage between the predicted test data and the actual data

In [ ]:
```
#diff_percentage = (diff_test / y_test_array) * 100
with np.errstate(divide='ignore', invalid='ignore'):
    diff_percentage = np.nan_to_num(np.divide(diff_test, y_test_array) * 100)

diff_percentage = np.array(diff_percentage)
diff_percentage = pd.DataFrame(diff_percentage)
diff_percentage = diff_percentage.fillna(0)
```

In [ ]:
```
diff_percentage = diff_percentage.values.ravel()
```

Illustrating the distribution of the absolute percentages between actual and predicted values (test data)

In [ ]:
```
# Create bins for the histogram
bins = np.arange(-201, 201, 5)  # Adjust the range and width of bins as needed

# Create the histogram
plt.hist(diff_percentage, bins=bins, color=color_lightblue, density=True)

# Calculate the mean and standard deviation of your data
mu = np.mean(diff_percentage)
std = np.std(diff_percentage)

# Generate the x values for your normal distribution
xmin, xmax = plt.xlim()
x = np.linspace(xmin, xmax, 200)

# Generate the y values (probability density function) for your normal distribution
p = norm.pdf(x, mu, std)

# Plot the normal distribution
plt.plot(x, p, 'k', linewidth=2, color=color_darkblue)

# Add labels and title
plt.xlabel('Percentage Difference between Actual and Predicted Values')
plt.ylabel('Probability Density')
plt.title('Distribution of Percentage Differences of Outrigger (S2)')

# Show the plot
plt.show()

print("The mean is equal to", round(mu,3), '%')
```

ANALYSIS OF PREDICTIONS

RMSE for each output

In [ ]:
```
n_outputs = y_test_array_pd.shape[1]  # Get the number of outputs
rmse_list_S2 = []

for i in range(n_outputs):
    y_test_i = y_test_array_pd.iloc[:, i]  # Get the true values for the i-th output
    y_pred_i = y_pred_test_pd.iloc[:, i]  # Get the predicted values for the i-th output

    rmse_i = np.sqrt(mean_squared_error(y_test_i, y_pred_i))  # Calculate RMSE for the i-th output
    rmse_list_S2.append(rmse_i)  # Append the RMSE to the list

# Now rmse_list contains the RMSE for each output separately
print("RMSE for each output separately:")
for i, rmse_i in enumerate(rmse_list_S2):
    print(f"Output {i + 1}: {rmse_i}")
```

MAPE of each output

In [ ]:
```
# Define lists to store MAPE values for each output
mape_list_S2 = []

epsilon = 1e-6  # Small constant to avoid division by zero

# Iterate over each output
for i in range(n_outputs):
    # Get the true values for the i-th output
    y_test_i = y_test_array_pd.iloc[:, i]

    # Get the predicted values for the i-th output
    y_pred_i = y_pred_test_pd.iloc[:, i]

    # Calculate absolute errors
    abs_errors = np.abs(y_test_i - y_pred_i)

    # Calculate percentage errors, adding epsilon to denominator
    perc_errors = abs_errors / (y_test_i + epsilon) * 100

    # Calculate mean absolute percentage error (MAPE) for the i-th output
    mape_i = np.mean(perc_errors)

    # Append the MAPE to the list
    mape_list_S2.append(mape_i)

# Print the MAPE for each output separately
print("Mean Absolute Percentage Error (MAPE) for each output separately:")
for i, mape_i in enumerate(mape_list_S2):
    print(f"Output {i + 1}: {mape_i}")

np.mean(mape_list_S2)
```

Plot of convergence

In [ ]:
```
fig, axes = plt.subplots(1, 2, figsize=(12, 5))  # Create a figure with 1 row and 2 columns

# Plot the original plot on the first subplot
axes[0].plot(best_model_S2.losses['training_loss'], label='Training Loss', color = color_darkblue)
axes[0].plot(best_model_S2.losses['validation_loss'], label='Validation Loss',  color = color_orange)
axes[0].set_title('Convergence Plot for Outrigger (S2)')
axes[0].set_xlabel('Epochs')
axes[0].set_ylabel('Loss [MSE]')
axes[0].legend()

ymax = np.max(best_model_S2.losses['validation_loss'])

axes[0].set_ylim(bottom=-10000, top=ymax+100)

# Plot the zoomed-in plot on the second subplot
axes[1].plot(best_model_S2.losses['training_loss'], label='Training Loss', color = color_darkblue)
axes[1].plot(best_model_S2.losses['validation_loss'], label='Validation Loss', color = color_orange)
axes[1].set_title('Zoomed-in Convergence Plot for Outrigger (S2)')
axes[1].set_xlabel('Epochs')
axes[1].set_ylabel('Loss [MSE]')
axes[1].legend()

# Set limits for better visualization on the zoomed-in plot
axes[1].set_xlim(left=0-50, right=800)
axes[1].set_ylim(bottom=-10000, top=ymax+100)

plt.tight_layout()
plt.show()
```

3a) MAKE PREDICTIONS : BRACED FRAMED TUBE

In [ ]:
```
scaler.get_feature_names_out()
```

A) ELEMENT & HEIGHT ZONE SPLIT

In [ ]:
```
colors = [color_blue1, color_blue2, color_blue3, color_blue4, color_green1, color_green2]
```

In [ ]:
```
from ipywidgets import fixed
```

In [ ]:
```
# Initialize rmse_list_S1 outside the function
rmse_list_S1 = rmse_list_S1

# Function to make predictions
def make_predictions_perHZ_S1(floorplan_length, height, rmse_list_S1):
    # Prepare input data
    input_data = pd.DataFrame({
        'in:Length [m]': [floorplan_length],
        'in:Height [m]': [height]
    })

    input_data_scaled = scaler.transform(input_data)

    # Make predictions using the best_model
    predictions_S1, num_zones = best_model_S1.predict(input_data_scaled, input_data)


    # Convert the predictions from mass to costs and embodied carbon
    # Costs
    predictions_S1[:, 0:20] *= s_price
    p_total_costs_S1 = predictions_S1[:, 0:20].sum(axis=1)

    # COSTS
    # Extracting values for each height zone
    diagrid1 = predictions_S1[0][19]
    diagrid2 = predictions_S1[0][19] if predictions_S1[0][1] != 0 else 0
    diagrid3 = predictions_S1[0][19] if predictions_S1[0][2] != 0 else 0
    diagrid4 = predictions_S1[0][19] if predictions_S1[0][3] != 0 else 0
    diagrid5 = predictions_S1[0][19] if predictions_S1[0][4] != 0 else 0
    diagrid6 = predictions_S1[0][19] if predictions_S1[0][5] != 0 else 0

    floorbeams1 = predictions_S1[0][12]
    floorbeams2 = predictions_S1[0][12] if predictions_S1[0][1] != 0 else 0
    floorbeams3 = predictions_S1[0][12] if predictions_S1[0][2] != 0 else 0
    floorbeams4 = predictions_S1[0][12] if predictions_S1[0][3] != 0 else 0
    floorbeams5 = predictions_S1[0][12] if predictions_S1[0][4] != 0 else 0
    floorbeams6 = predictions_S1[0][12] if predictions_S1[0][5] != 0 else 0

    costs_floors1 = costs_floors
    costs_floors2 = costs_floors if predictions_S1[0][1] != 0 else 0
    costs_floors3 = costs_floors if predictions_S1[0][2] != 0 else 0
    costs_floors4 = costs_floors if predictions_S1[0][3] != 0 else 0
    costs_floors5 = costs_floors if predictions_S1[0][4] != 0 else 0
    costs_floors6 = costs_floors if predictions_S1[0][5] != 0 else 0

    # Creating a DataFrame for costs
    hz_S1_costs = {
        'Category': ['Height Zone 1', 'Height Zone 2', 'Height Zone 3', 'Height Zone 4', 'Height Zone 5', 'Height Zone 6'],
        'FloorBeams': [floorbeams1, floorbeams2, floorbeams3, floorbeams4, floorbeams5, floorbeams6],
        'OuterBeams': [predictions_S1[0][0], predictions_S1[0][1], predictions_S1[0][2], predictions_S1[0][3], predictions_S1[0][4], predi
        'Floor System': [costs_floors1, costs_floors2, costs_floors3, costs_floors4, costs_floors5, costs_floors6],
        'InnerColumns': [predictions_S1[0][13], predictions_S1[0][14], predictions_S1[0][15], predictions_S1[0][16], predictions_S1[0][17]
        'OuterColumns': [predictions_S1[0][6], predictions_S1[0][7], predictions_S1[0][8], predictions_S1[0][9], predictions_S1[0][10], pr
        'Bracing': [diagrid1, diagrid2, diagrid3, diagrid4, diagrid5, diagrid6],
    }

    HZ1_grav = floorbeams1 + predictions_S1[0][0] + costs_floors1 + predictions_S1[0][13]
    HZ1_stab = predictions_S1[0][6] + diagrid1
    ratio_stab_HZ1 = np.round(HZ1_stab / (HZ1_stab + HZ1_grav) * 100, 1)

    HZ2_grav = floorbeams2 + predictions_S1[0][1] + costs_floors2 + predictions_S1[0][14]
    HZ2_stab = predictions_S1[0][7] + diagrid2
    ratio_stab_HZ2 = np.round(HZ2_stab / (HZ2_stab + HZ2_grav) * 100, 1)

    HZ3_grav = floorbeams3 + predictions_S1[0][2] + costs_floors3 + predictions_S1[0][15]
    HZ3_stab = predictions_S1[0][8] + diagrid3
    if HZ3_grav == 0:
        ratio_stab_HZ3 = 0
    else:
        ratio_stab_HZ3 = np.round(HZ3_stab / (HZ3_stab + HZ3_grav) * 100, 1)

    HZ4_grav = floorbeams4 + predictions_S1[0][3] + costs_floors4 + predictions_S1[0][16]
    HZ4_stab = predictions_S1[0][9] + diagrid4
    if HZ4_grav == 0:
        ratio_stab_HZ4 = 0
    else:
        ratio_stab_HZ4 = np.round(HZ4_stab / (HZ4_stab + HZ4_grav) * 100, 1)

    HZ5_grav = floorbeams5 + predictions_S1[0][4] + costs_floors5 + predictions_S1[0][17]
    HZ5_stab = predictions_S1[0][10] + diagrid5
    if HZ5_grav == 0:
        ratio_stab_HZ5 = 0
    else:
        ratio_stab_HZ5 = np.round(HZ5_stab / (HZ5_stab + HZ5_grav) * 100, 1)

    HZ6_grav = floorbeams6 + predictions_S1[0][5] + costs_floors6 + predictions_S1[0][18]
    HZ6_stab = predictions_S1[0][11] + diagrid6
    if HZ6_grav == 0:
        ratio_stab_HZ6 = 0
    else:
        ratio_stab_HZ6 = np.round(HZ6_stab / (HZ6_stab + HZ6_grav) * 100, 1)

    print('HZ1: Ratio of Structural Costs coming from', '\033[1;32m\033[1mStability System\033[0m', ratio_stab_HZ1, '%')
    print('HZ2: Ratio of Structural Costs coming from', '\033[1;32m\033[1mStability System\033[0m', ratio_stab_HZ2, '%')
    print('HZ3: Ratio of Structural Costs coming from', '\033[1;32m\033[1mStability System\033[0m', ratio_stab_HZ3, '%')
    print('HZ4: Ratio of Structural Costs coming from', '\033[1;32m\033[1mStability System\033[0m', ratio_stab_HZ4, '%')
    print('HZ5: Ratio of Structural Costs coming from', '\033[1;32m\033[1mStability System\033[0m', ratio_stab_HZ5, '%')
    print('HZ6: Ratio of Structural Costs coming from', '\033[1;32m\033[1mStability System\033[0m', ratio_stab_HZ6, '%')


    df_hz_costs = pd.DataFrame(hz_S1_costs)
    df_hz_costs.set_index('Category', inplace=True)
```

```
    # Plotting costs for each height zone

    ax_costs = df_hz_costs.plot(kind='bar', stacked=True, figsize=(7, 5), color=colors)
    ax_costs.set_title('Structural Costs Predictions for each Height Zone')
    ax_costs.set_xlabel('Height Zones')
    ax_costs.set_ylabel('Predicted Costs [€/m2GFA]')
    ax_costs.set_xticklabels(ax_costs.get_xticklabels(), rotation=45, ha='right')
    ax_costs.legend(loc='lower left', bbox_to_anchor=(1, 0))
    ax_costs.set_ylim([0, 1000])

    # Display the plot
    plt.show()

    predictions_S1, num_zones = best_model_S1.predict(input_data_scaled, input_data)
    predictions_S1[:, 0:25] *= s_EmCO2 * CO2_sprice

    diagrid1 = predictions_S1[0][19]
    diagrid2 = predictions_S1[0][19] if predictions_S1[0][1] != 0 else 0
    diagrid3 = predictions_S1[0][19] if predictions_S1[0][2] != 0 else 0
    diagrid4 = predictions_S1[0][19] if predictions_S1[0][3] != 0 else 0
    diagrid5 = predictions_S1[0][19] if predictions_S1[0][4] != 0 else 0
    diagrid6 = predictions_S1[0][19] if predictions_S1[0][5] != 0 else 0

    floorbeams1 = predictions_S1[0][12]
    floorbeams2 = predictions_S1[0][12] if predictions_S1[0][1] != 0 else 0
    floorbeams3 = predictions_S1[0][12] if predictions_S1[0][2] != 0 else 0
    floorbeams4 = predictions_S1[0][12] if predictions_S1[0][3] != 0 else 0
    floorbeams5 = predictions_S1[0][12] if predictions_S1[0][4] != 0 else 0
    floorbeams6 = predictions_S1[0][12] if predictions_S1[0][5] != 0 else 0

    EmCO2_floors1 = EmCO2_floors
    EmCO2_floors2 = EmCO2_floors if predictions_S1[0][1] != 0 else 0
    EmCO2_floors3 = EmCO2_floors if predictions_S1[0][2] != 0 else 0
    EmCO2_floors4 = EmCO2_floors if predictions_S1[0][3] != 0 else 0
    EmCO2_floors5 = EmCO2_floors if predictions_S1[0][4] != 0 else 0
    EmCO2_floors6 = EmCO2_floors if predictions_S1[0][5] != 0 else 0

    # Calculate predicted embodied carbon for each height zone
    pred_EmCO2_HZ1 = predictions_S1[0][0] + predictions_S1[0][6] + floorbeams1 + predictions_S1[0][13] + diagrid1
    pred_EmCO2_HZ2 = predictions_S1[0][1] + predictions_S1[0][7] + floorbeams1 + predictions_S1[0][14] + diagrid2
    pred_EmCO2_HZ3 = predictions_S1[0][2] + predictions_S1[0][8] + floorbeams1 + predictions_S1[0][15] + diagrid3
    pred_EmCO2_HZ4 = predictions_S1[0][3] + predictions_S1[0][9] + floorbeams1 + predictions_S1[0][16] + diagrid4
    pred_EmCO2_HZ5 = predictions_S1[0][4] + predictions_S1[0][10] + floorbeams1 + predictions_S1[0][17] + diagrid5
    pred_EmCO2_HZ6 = predictions_S1[0][5] + predictions_S1[0][11] + floorbeams1 + predictions_S1[0][18] + diagrid6

    pred_EmCO2 = [pred_EmCO2_HZ1, pred_EmCO2_HZ2, pred_EmCO2_HZ3, pred_EmCO2_HZ4, pred_EmCO2_HZ5, pred_EmCO2_HZ6]

    # Creating a DataFrame for embodied carbon
    hz_S1_EmCO2 = {
        'Category': ['Height Zone 1', 'Height Zone 2', 'Height Zone 3', 'Height Zone 4', 'Height Zone 5', 'Height Zone 6'],
        'FloorBeams': [floorbeams1, floorbeams2, floorbeams3, floorbeams4, floorbeams5, floorbeams6],
        'OuterBeams': [predictions_S1[0][0], predictions_S1[0][1], predictions_S1[0][2], predictions_S1[0][3], predictions_S1[0][4], predi
        'Floor System': [EmCO2_floors1, EmCO2_floors2, EmCO2_floors3, EmCO2_floors4, EmCO2_floors5, EmCO2_floors6],
        'InnerColumns': [predictions_S1[0][13], predictions_S1[0][14], predictions_S1[0][15], predictions_S1[0][16], predictions_S1[0][17]
        'OuterColumns': [predictions_S1[0][6], predictions_S1[0][7], predictions_S1[0][8], predictions_S1[0][9], predictions_S1[0][10], pr
        'Bracing': [diagrid1, diagrid2, diagrid3, diagrid4, diagrid5, diagrid6],
    }

    df_hz_EmCO2 = pd.DataFrame(hz_S1_EmCO2)
    df_hz_EmCO2.set_index('Category', inplace=True)

    # Plotting embodied carbon for each height zone
    ax_EmCO2 = df_hz_EmCO2.plot(kind='bar', stacked=True, figsize=(7, 5), color=colors)
    ax_EmCO2.set_title('Environmental Costs Predictions for each Height Zone')
    ax_EmCO2.set_xlabel('Height Zones')
    ax_EmCO2.set_ylabel('Predicted Environmental Costs [€/m2GFA]')
    ax_EmCO2.set_xticklabels(ax_EmCO2.get_xticklabels(), rotation=45, ha='right')
    ax_EmCO2.legend(loc='lower left', bbox_to_anchor=(1, 0))
    ax_EmCO2.set_ylim([0, 15])

    # Display the plot
    plt.show()


# Create interactive sliders for floorplan length and height
floorplan_length_slider = widgets.FloatSlider(value=20, min=5,
                                              max=75, step=1,
                                              description='Width of the Floorplan:')
floorplan_length_slider.layout.width = '40%'
floorplan_length_slider.style = {'description_width': '50%'}

height_slider = widgets.FloatSlider(value=100, min=mass_S1['in:Height [m]'].min(),
                                    max=500, step=1,
                                    description='Height:')
height_slider.layout.width = '40%'
height_slider.style = {'description_width': '50%'}

# Create an interactive output widget
output = widgets.interactive_output(make_predictions_perHZ_S1, {'floorplan_length': floorplan_length_slider, 'height': height_slider, 'rms

# Display the sliders and output
display(floorplan_length_slider, height_slider, output)
```

In [ ]:
```
# Function to make predictions
def make_predictions_perHZ(floorplan_length, height):
    # Prepare input data
    input_data = pd.DataFrame({
    'in:Length [m]': [floorplan_length],
    'in:Height [m]': [height]
    })

    input_data_scaled = scaler.transform(input_data)
    predictions_S1, num_zones = best_model_S1.predict(input_data_scaled, input_data)

    #Determination height per height zone
    if num_zones == 1:
        hz1 = height
        pred_OB = predictions_S1[0][0]
        pred_OC = predictions_S1[0][6]
        pred_FB = predictions_S1[0][12]
        pred_IC = predictions_S1[0][13]
        pred_Diagrid = predictions_S1[0][19]

    if num_zones == 2:
        hz1 = (floorplan_length // 3) * 3
        hz2 = height - hz1

        pred_OB = (predictions_S1[0][0]*hz1 + predictions_S1[0][1]*hz2) / height
        pred_OC = (predictions_S1[0][6]*hz1 + predictions_S1[0][7]*hz2) / height
        pred_FB = predictions_S1[0][12]
        pred_IC = (predictions_S1[0][13]*hz1 + predictions_S1[0][14]*hz2) / height
        pred_Diagrid = predictions_S1[0][19]

    if num_zones == 3:
        hz1 = (floorplan_length // 3) * 3
        hz3 = (floorplan_length // 3) * 3
        hz2 = height - hz1 - hz3

        pred_OB = (predictions_S1[0][0]*hz1 + predictions_S1[0][1]*hz2 + predictions_S1[0][2]*hz3) / height
        pred_OC = (predictions_S1[0][6]*hz1 + predictions_S1[0][7]*hz2 + predictions_S1[0][8]*hz3) / height
        pred_FB = predictions_S1[0][12]
        pred_IC = (predictions_S1[0][13]*hz1 + predictions_S1[0][14]*hz2 + predictions_S1[0][15]*hz3) / height
        pred_Diagrid = predictions_S1[0][19]

    if num_zones == 4:
        hz1 = (floorplan_length // 3) * 3
        hz4 = (floorplan_length // 3) * 3
        hz23 = height - hz1 - hz4
        hz2 = hz23 / 2
        hz3 = hz23 / 2

        pred_OB = (predictions_S1[0][0]*hz1 + predictions_S1[0][1]*hz2 + predictions_S1[0][2]*hz3 + predictions_S1[0][3]*hz4) / height
        pred_OC = (predictions_S1[0][6]*hz1 + predictions_S1[0][7]*hz2 + predictions_S1[0][8]*hz3 + predictions_S1[0][9]*hz4) / height
        pred_FB = predictions_S1[0][12]
        pred_IC = (predictions_S1[0][13]*hz1 + predictions_S1[0][14]*hz2 + predictions_S1[0][15]*hz3 + predictions_S1[0][16]*hz4) / height
        pred_Diagrid = predictions_S1[0][19]


    if num_zones == 5:
        hz1 = (floorplan_length // 3) * 3
        hz5 = (floorplan_length // 3) * 3
        hz234 = height - hz1 - hz5
        hz2 = hz234 / 3
        hz3 = hz234 / 3
        hz4 = hz234 / 3

        pred_OB = (predictions_S1[0][0]*hz1 + predictions_S1[0][1]*hz2 + predictions_S1[0][2]*hz3 + predictions_S1[0][3]*hz4 + predictions
        pred_OC = (predictions_S1[0][6]*hz1 + predictions_S1[0][7]*hz2 + predictions_S1[0][8]*hz3 + predictions_S1[0][9]*hz4 + predictions
        pred_FB = predictions_S1[0][12]
        pred_IC = (predictions_S1[0][13]*hz1 + predictions_S1[0][14]*hz2 + predictions_S1[0][15]*hz3 + predictions_S1[0][16]*hz4 + predict
        pred_Diagrid = predictions_S1[0][19]


    if num_zones == 6:
        hz1 = (floorplan_length // 3) * 3
        hz6 = (floorplan_length // 3) * 3
        hz2345 = height - hz1 - hz6
        hz2 = hz2345 / 5
        hz3 = hz2345 / 5
        hz4 = hz2345 / 5
        hz5 = hz2345 / 5

        pred_OB = (predictions_S1[0][0]*hz1 + predictions_S1[0][1]*hz2 + predictions_S1[0][2]*hz3 + predictions_S1[0][3]*hz4 + predictions
        pred_OC = (predictions_S1[0][6]*hz1 + predictions_S1[0][7]*hz2 + predictions_S1[0][8]*hz3 + predictions_S1[0][9]*hz4 + predictions
        pred_FB = predictions_S1[0][12]
        pred_IC = (predictions_S1[0][13]*hz1 + predictions_S1[0][14]*hz2 + predictions_S1[0][15]*hz3 + predictions_S1[0][16]*hz4 + predict
        pred_Diagrid = predictions_S1[0][19]

    pred_Costs_OB = pred_OB * s_price
    pred_Costs_OC = pred_OC * s_price
    pred_Costs_FB = pred_FB * s_price
    pred_Costs_IC = pred_IC * s_price
    pred_Costs_diagrid = pred_Diagrid * s_price

    ## Creating a DataFrame
    StructuralElement_Costs_data = {
            'Category': ['FloorBeams', 'OuterBeams', 'Floor System', 'InnerColumns', 'Bracing', 'Outer Columns'],
            'EmCO2 per element': [pred_Costs_FB, pred_Costs_OB, costs_floors, pred_Costs_IC, pred_Costs_diagrid, pred_Costs_OC]
        }

    ## Creating DataFrame
    df_SE_Costs = pd.DataFrame(StructuralElement_Costs_data)

    df_SE_Costs.set_index('Category', inplace = True)
```

```
    colors = [color_blue1,  # FloorBeams
              color_blue2,  # OuterBeams
              color_blue3,  # Floor System
              color_blue4,  # InnerColumns
              color_green1,     # Bracing
             color_green2]  # Outer Columns

    ax = df_SE_Costs.plot(kind='bar', stacked=True, figsize=(7, 5), color='white', legend=None)

    # Set colors for each part of the bar
    for p, col in zip(ax.patches, colors):
        p.set_facecolor(col)

    ax.set_title('Predicted Structural Costs: Structural Element Split')
    ax.set_xlabel('Structural Elements')
    ax.set_ylabel('Predicted Structural Costs [€/m2GFA]')
    ax.set_xticklabels(ax.get_xticklabels(), rotation=45, ha='right')
    ax.set_ylim([0, 250])

    for p in ax.patches:
        total_value = p.get_height()
        ax.annotate(f'{total_value:.1f}', (p.get_x() + p.get_width() / 2., total_value),
                ha='center', va='center', xytext=(0, 10), textcoords='offset points')

    pred_Costs_stability = pred_Costs_OC + pred_Costs_diagrid
    pred_Costs_gravity = pred_Costs_OB + pred_Costs_FB + pred_Costs_IC + costs_floors
    part_stability = (pred_Costs_stability / (pred_Costs_stability+pred_Costs_gravity))*100

    print('Total Structural Costs of','\033[1;34m\033[1mGravity System\033[0m', 'is', np.round(pred_Costs_gravity, 0), '€/m2GFA')
    print('Total Structural Costs of', '\033[1;32m\033[1mStability System\033[0m', 'is', np.round(pred_Costs_stability, 0), '€/m2GFA')
    print('Ratio of Structural Costs coming from Stability System:', np.round(part_stability,1), '%')

    # Display the plot
    plt.show()


    # EMBODIED CARBON
    pred_EmCO2_OB = pred_OB * s_EmCO2*CO2_sprice
    pred_EmCO2_OC = pred_OC * s_EmCO2*CO2_sprice
    pred_EmCO2_FB = pred_FB * s_EmCO2*CO2_sprice
    pred_EmCO2_IC = pred_IC * s_EmCO2*CO2_sprice
    pred_EmCO2_diagrid = pred_Diagrid * s_EmCO2*CO2_sprice

    ## Creating a DataFrame for HZ1 and HZ2 predictions
    StructuralElement_EmCO2_data = {
            'Category': ['FloorBeams', 'OuterBeams', 'Floor System', 'InnerColumns', 'Bracing', 'Outer Columns'],
            'EmCO2 per element': [pred_EmCO2_FB, pred_EmCO2_OB, EmCO2_floors, pred_EmCO2_IC, pred_EmCO2_diagrid, pred_EmCO2_OC]
        }

    ## Creating DataFrame
    df_SE_EmCO2 = pd.DataFrame(StructuralElement_EmCO2_data)

    df_SE_EmCO2.set_index('Category', inplace = True)

    colors = [color_blue1,  # FloorBeams
              color_blue2,  # OuterBeams
              color_blue3,  # Floor System
              color_blue4,  # InnerColumns
              color_green1,     # Bracing
             color_green2]  # Outer Columns

    pred_EmCO2_stability = pred_EmCO2_OC + pred_EmCO2_diagrid
    pred_EmCO2_gravity = pred_EmCO2_OB + pred_EmCO2_FB + pred_EmCO2_IC + EmCO2_floors
    part_stability = (pred_EmCO2_stability / (pred_EmCO2_stability+pred_EmCO2_gravity))*100

    print('Total Environmental Costs of','\033[1;34m\033[1mGravity System\033[0m', 'is', np.round(pred_EmCO2_gravity, 0), '€/m2GFA')
    print('Total Environmental Costs of', '\033[1;32m\033[1mStability System\033[0m', 'is', np.round(pred_EmCO2_stability, 0), '€/m2GFA')
    print('Ratio of Environmental Costs coming from Stability System:', np.round(part_stability,1), '%')


    ax = df_SE_EmCO2.plot(kind='bar', stacked=True, figsize=(7, 5), color='white', legend=None)  # Initialize with white color

    # Set colors for each part of the bar
    for p, col in zip(ax.patches, colors):
        p.set_facecolor(col)

    ax.set_title('Predicted Environmental Costs: Structural Element Split')
    ax.set_xlabel('Structural Elements')
    ax.set_ylabel('Predicted Environmental Costs [€/m2GFA]')
    ax.set_xticklabels(ax.get_xticklabels(), rotation=45, ha='right')
    ax.set_ylim([0, 5])

    for p in ax.patches:
        total_value = p.get_height()
        ax.annotate(f'{total_value:.1f}', (p.get_x() + p.get_width() / 2., total_value),
                ha='center', va='center', xytext=(0, 10), textcoords='offset points')

    # Display the plot
    plt.show()


# Create interactive sliders for floorplan length and height
floorplan_length_slider = widgets.FloatSlider(value=20, min=5,
                                              max=75, step=1,
                                              description='Width of the Floorplan:')

floorplan_length_slider.layout.width = '40%'
floorplan_length_slider.style = {'description_width': '50%'}

height_slider = widgets.FloatSlider(value=100, min=mass_S1['in:Height [m]'].min(),
                                    max=500, step=1,
                                    description='Height of the Structure:')
```

In [ ]:
```
constant_variable = 'in:Length [m]'
constant_value_width = (21.6) # Set the constant value of width

# Filter data for the specified constant value
filtered_data_S1 = mass_S1[mass_S1[constant_variable] == constant_value_width]

constant_variable = 'in:Height [m]'
constant_value_height = (150) # Set the constant value of height

# Filter data for the specified constant value
filtered_data_S1 = filtered_data_S1[filtered_data_S1[constant_variable] == constant_value_height]
masses_data_S1 = filtered_data_S1.iloc[:, 8:28]
num_zones = int(filtered_data_S1.iloc[:, 28])
floorplan_length = constant_value_width
height = constant_value_height

if num_zones == 1:
    hz1 = height
    mass_OB = masses_data_S1.iloc[0, 0]
    mass_OC = masses_data_S1.iloc[0, 6]
    mass_FB = masses_data_S1.iloc[0, 12]
    mass_IC = masses_data_S1.iloc[0, 13]
    mass_Diagrid = masses_data_S1.iloc[0, 19]

if num_zones == 2:
    hz1 = (floorplan_length // 3) * 3
    hz2 = height - hz1

    mass_OB = (masses_data_S1.iloc[0, 0]*hz1 + masses_data_S1.iloc[0, 1]*hz2) / height
    mass_OC = (masses_data_S1.iloc[0, 6]*hz1 + masses_data_S1.iloc[0, 7]*hz2) / height
    mass_FB = masses_data_S1.iloc[0, 12]
    mass_IC = (masses_data_S1.iloc[0, 13]*hz1 + masses_data_S1.iloc[0, 14]*hz2) / height
    mass_Diagrid = masses_data_S1.iloc[0, 19]

if num_zones == 3:
    hz1 = (floorplan_length // 3) * 3
    hz3 = (floorplan_length // 3) * 3
    hz2 = height - hz1 - hz3

    mass_OB = (masses_data_S1.iloc[0, 0]*hz1 + masses_data_S1.iloc[0, 1]*hz2 + masses_data_S1.iloc[0, 2]*hz3) / height
    mass_OC = (masses_data_S1.iloc[0, 6]*hz1 + masses_data_S1.iloc[0, 7]*hz2 + masses_data_S1.iloc[0, 8]*hz3) / height
    mass_FB = masses_data_S1.iloc[0, 12]
    mass_IC = (masses_data_S1.iloc[0, 13]*hz1 + masses_data_S1.iloc[0, 14]*hz2 + masses_data_S1.iloc[0, 15]*hz3) / height
    mass_Diagrid = masses_data_S1.iloc[0, 19]

if num_zones == 4:
    hz1 = (floorplan_length // 3) * 3
    hz4 = (floorplan_length // 3) * 3
    hz23 = height - hz1 - hz4
    hz2 = hz23 / 2
    hz3 = hz23 / 2

    mass_OB = (masses_data_S1.iloc[0, 0]*hz1 + masses_data_S1.iloc[0, 1]*hz2 + masses_data_S1.iloc[0, 2]*hz3 + masses_data_S1.iloc[0, 3]*h
    mass_OC = (masses_data_S1.iloc[0, 6]*hz1 + masses_data_S1.iloc[0, 7]*hz2 + masses_data_S1.iloc[0, 8]*hz3 + masses_data_S1.iloc[0, 9]*h
    mass_FB = masses_data_S1.iloc[0, 12]
    mass_IC = (masses_data_S1.iloc[0, 13]*hz1 + masses_data_S1.iloc[0, 14]*hz2 + masses_data_S1.iloc[0, 15]*hz3 + masses_data_S1.iloc[0, 1
    mass_Diagrid = masses_data_S1.iloc[0, 19]


if num_zones == 5:
    hz1 = (floorplan_length // 3) * 3
    hz5 = (floorplan_length // 3) * 3
    hz234 = height - hz1 - hz5
    hz2 = hz234 / 3
    hz3 = hz234 / 3
    hz4 = hz234 / 3

    mass_OB = (masses_data_S1.iloc[0, 0]*hz1 + masses_data_S1.iloc[0, 1]*hz2 + masses_data_S1.iloc[0, 2]*hz3 + masses_data_S1.iloc[0, 3]*h
    mass_OC = (masses_data_S1.iloc[0, 6]*hz1 + masses_data_S1.iloc[0, 7]*hz2 + masses_data_S1.iloc[0, 8]*hz3 + masses_data_S1.iloc[0, 9]*h
    mass_FB = masses_data_S1.iloc[0, 12]
    mass_IC = (masses_data_S1.iloc[0, 13]*hz1 + masses_data_S1.iloc[0, 14]*hz2 + masses_data_S1.iloc[0, 15]*hz3 + masses_data_S1.iloc[0, 1
    mass_Diagrid = masses_data_S1.iloc[0, 19]


if num_zones == 6:
    hz1 = (floorplan_length // 3) * 3
    hz6 = (floorplan_length // 3) * 3
    hz2345 = height - hz1 - hz6
    hz2 = hz2345 / 5
    hz3 = hz2345 / 5
    hz4 = hz2345 / 5
    hz5 = hz2345 / 5

    mass_OB = (masses_data_S1.iloc[0, 0]*hz1 + masses_data_S1.iloc[0, 1]*hz2 + masses_data_S1.iloc[0, 2]*hz3 + masses_data_S1.iloc[0, 3]*h
    mass_OC = (masses_data_S1.iloc[0, 6]*hz1 + masses_data_S1.iloc[0, 7]*hz2 + masses_data_S1.iloc[0, 8]*hz3 + masses_data_S1.iloc[0, 9]*h
    mass_FB = masses_data_S1.iloc[0, 12]
    mass_IC = (masses_data_S1.iloc[0, 13]*hz1 + masses_data_S1.iloc[0, 14]*hz2 + masses_data_S1.iloc[0, 15]*hz3 + masses_data_S1.iloc[0, 1
    mass_Diagrid = masses_data_S1.iloc[0, 19]

input_Costs_OB = mass_OB * s_price
input_Costs_OC = mass_OC * s_price
input_Costs_FB = mass_FB * s_price
input_Costs_IC = mass_IC * s_price
input_Costs_diagrid = mass_Diagrid * s_price


# Extract the relevant columns
X_filtered_S1 = filtered_data_S1['in:Height [m]']
```

In [ ]:
```
# Function to make predictions
def make_predictions_perHZ(floorplan_length, height):
    # Prepare input data
    input_data = pd.DataFrame({
    'in:Length [m]': [floorplan_length],
    'in:Height [m]': [height]
    })

    input_data_scaled = scaler.transform(input_data)
    predictions_S1, num_zones = best_model_S1.predict(input_data_scaled, input_data)

    #Determination height per height zone
    if num_zones == 1:
        hz1 = height
        pred_OB = predictions_S1[0][0]
        pred_OC = predictions_S1[0][6]
        pred_FB = predictions_S1[0][12]
        pred_IC = predictions_S1[0][13]
        pred_Diagrid = predictions_S1[0][19]

    if num_zones == 2:
        hz1 = (floorplan_length // 3) * 3
        hz2 = height - hz1

        pred_OB = (predictions_S1[0][0]*hz1 + predictions_S1[0][1]*hz2) / height
        pred_OC = (predictions_S1[0][6]*hz1 + predictions_S1[0][7]*hz2) / height
        pred_FB = predictions_S1[0][12]
        pred_IC = (predictions_S1[0][13]*hz1 + predictions_S1[0][14]*hz2) / height
        pred_Diagrid = predictions_S1[0][19]

    if num_zones == 3:
        hz1 = (floorplan_length // 3) * 3
        hz3 = (floorplan_length // 3) * 3
        hz2 = height - hz1 - hz3

        pred_OB = (predictions_S1[0][0]*hz1 + predictions_S1[0][1]*hz2 + predictions_S1[0][2]*hz3) / height
        pred_OC = (predictions_S1[0][6]*hz1 + predictions_S1[0][7]*hz2 + predictions_S1[0][8]*hz3) / height
        pred_FB = predictions_S1[0][12]
        pred_IC = (predictions_S1[0][13]*hz1 + predictions_S1[0][14]*hz2 + predictions_S1[0][15]*hz3) / height
        pred_Diagrid = predictions_S1[0][19]

    if num_zones == 4:
        hz1 = (floorplan_length // 3) * 3
        hz4 = (floorplan_length // 3) * 3
        hz23 = height - hz1 - hz4
        hz2 = hz23 / 2
        hz3 = hz23 / 2

        pred_OB = (predictions_S1[0][0]*hz1 + predictions_S1[0][1]*hz2 + predictions_S1[0][2]*hz3 + predictions_S1[0][3]*hz4) / height
        pred_OC = (predictions_S1[0][6]*hz1 + predictions_S1[0][7]*hz2 + predictions_S1[0][8]*hz3 + predictions_S1[0][9]*hz4) / height
        pred_FB = predictions_S1[0][12]
        pred_IC = (predictions_S1[0][13]*hz1 + predictions_S1[0][14]*hz2 + predictions_S1[0][15]*hz3 + predictions_S1[0][16]*hz4) / height
        pred_Diagrid = predictions_S1[0][19]


    if num_zones == 5:
        hz1 = (floorplan_length // 3) * 3
        hz5 = (floorplan_length // 3) * 3
        hz234 = height - hz1 - hz5
        hz2 = hz234 / 3
        hz3 = hz234 / 3
        hz4 = hz234 / 3

        pred_OB = (predictions_S1[0][0]*hz1 + predictions_S1[0][1]*hz2 + predictions_S1[0][2]*hz3 + predictions_S1[0][3]*hz4 + predictions
        pred_OC = (predictions_S1[0][6]*hz1 + predictions_S1[0][7]*hz2 + predictions_S1[0][8]*hz3 + predictions_S1[0][9]*hz4 + predictions
        pred_FB = predictions_S1[0][12]
        pred_IC = (predictions_S1[0][13]*hz1 + predictions_S1[0][14]*hz2 + predictions_S1[0][15]*hz3 + predictions_S1[0][16]*hz4 + predict
        pred_Diagrid = predictions_S1[0][19]


    if num_zones == 6:
        hz1 = (floorplan_length // 3) * 3
        hz6 = (floorplan_length // 3) * 3
        hz2345 = height - hz1 - hz6
        hz2 = hz2345 / 5
        hz3 = hz2345 / 5
        hz4 = hz2345 / 5
        hz5 = hz2345 / 5

        pred_OB = (predictions_S1[0][0]*hz1 + predictions_S1[0][1]*hz2 + predictions_S1[0][2]*hz3 + predictions_S1[0][3]*hz4 + predictions
        pred_OC = (predictions_S1[0][6]*hz1 + predictions_S1[0][7]*hz2 + predictions_S1[0][8]*hz3 + predictions_S1[0][9]*hz4 + predictions
        pred_FB = predictions_S1[0][12]
        pred_IC = (predictions_S1[0][13]*hz1 + predictions_S1[0][14]*hz2 + predictions_S1[0][15]*hz3 + predictions_S1[0][16]*hz4 + predict
        pred_Diagrid = predictions_S1[0][19]

    pred_Costs_OB = pred_OB * s_price
    pred_Costs_OC = pred_OC * s_price
    pred_Costs_FB = pred_FB * s_price
    pred_Costs_IC = pred_IC * s_price
    pred_Costs_diagrid = pred_Diagrid * s_price

    ## Creating a DataFrame
    StructuralElement_Costs_pred = {
            'Category': ['FloorBeams', 'OuterColumns', 'InnerColumns', 'Bracing', 'OuterColumns'],
            'Costs per Element': [pred_Costs_FB, pred_Costs_OC, pred_Costs_IC, pred_Costs_diagrid, pred_Costs_OC]
        }

    df_pred_Costs = pd.DataFrame(StructuralElement_Costs_pred)
    df_pred_Costs.set_index('Category', inplace=True)

    StructuralElement_Costs_data = {
        'Category': ['FloorBeams', 'OuterColumns', 'InnerColumns', 'Bracing', 'OuterColumns'],
```

```
        'Costs per Element': [input_Costs_FB, input_Costs_OC, input_Costs_IC, input_Costs_diagrid, input_Costs_OC]
    }

    df_SE_Costs = pd.DataFrame(StructuralElement_Costs_data)
    df_SE_Costs.set_index('Category', inplace=True)

    # Plotting the bar chart
    ax = df_pred_Costs.plot(kind='bar', stacked=True, figsize=(7, 5), color='white', legend=None)

    colors = [color_blue1,
              color_blue2,
              color_blue4,
              color_green1,
              color_green2]

    # Set colors for each part of the bar
    for p, col in zip(ax.patches, colors):
        p.set_facecolor(col)

    # Add scatter points for the costs of structural elements
    ax.scatter(np.arange(len(df_SE_Costs)), df_SE_Costs['Costs per Element'], color=color_orange, zorder=5, label = 'S1: Input Data w = 43

    # Adding annotations for the scatter points
    #for i, cost in enumerate(df_SE_Costs['Costs per element']):
    #    ax.annotate(f'{cost:.1f}', (i, cost), xytext=(0, 5), textcoords='offset points', ha='center')

    ax.set_title('Predicted Costs: Structural Element Split')
    ax.set_xlabel('Structural Elements')
    ax.set_ylabel('Predicted Costs [€/GFA]')
    ax.set_xticklabels(ax.get_xticklabels(), rotation=45, ha='right')
    ax.legend(bbox_to_anchor=(1.02, 1), loc='upper left', borderaxespad=0)
    ax.set_ylim([0, 320])

    #for p in ax.patches:
    #    total_value = p.get_height()
    #    ax.annotate(f'{total_value:.1f}', (p.get_x() + p.get_width() / 2., total_value),
    #            ha='center', va='center', xytext=(0, 10), textcoords='offset points')


    # Display the plot
    plt.show()

# Create interactive sliders for floorplan length and height
floorplan_length_slider = widgets.FloatSlider(value=20, min=5,
                                              max=75, step=1,
                                              description='Width of the Floorplan:')

floorplan_length_slider.layout.width = '40%'
floorplan_length_slider.style = {'description_width': '50%'}

height_slider = widgets.FloatSlider(value=100, min=mass_S1['in:Height [m]'].min(),
                                    max=500, step=1,
                                    description='Height of the Structure:')

height_slider.layout.width = '40%'
height_slider.style = {'description_width': '50%'}


# Create an interactive output widget
output = widgets.interactive_output(make_predictions_perHZ, {'floorplan_length': floorplan_length_slider, 'height': height_slider})

# Display the sliders and output
display(floorplan_length_slider, height_slider, output)
```

3b) MAKE PREDICTIONS : OUTRIGGER SYSTEM

A) ELEMENT & HEIGHT ZONE SPLIT

In [ ]:
```
color_blue1 = '#D2EAF8'
color_blue2 = '#A1DDFF'
color_blue3 = '#38B6FF'
color_blue4 = '#417DB9'
color_blue5 = '#2192D4'
color_green1 = '#D7F1DB'
color_green2 = '#A6D8AE'
color_green3 = '#629F6B'
color_green4 = '#277B34'
colors = [color_blue1, color_blue2, color_blue3, color_blue4, color_blue5, color_green1, color_green2, color_green3, color_green4]
```

In [ ]:
```
EmCO2_floors
```

In [ ]:
```
# Function to make predictions
def make_predictions_perHZ_S2(floorplan_length, height):
    # Prepare input data
    input_data = pd.DataFrame({
    'in:Length [m]': [floorplan_length],
    'in:Height [m]': [height]
    })

    input_data_scaled = scaler2.transform(input_data)

    # Make predictions using the best_model
    predictions_S2, num_zones = best_model_S2.predict(input_data_scaled, input_data)

    # Convert the predictons from mass to costs and embodied carbon

    # COSTS
    predictions_S2[:, 0:32] *= s_price
    predictions_S2[:, 32] *= pc_rc_price

    Outrigger1 = predictions_S2[0][31]
    Core1 = predictions_S2[0][32]
    costs_floors1 = costs_floors
    FloorBeams1 = predictions_S2[0][18]


    if predictions_S2[0][1] == 0:
        Outrigger2 = 0
        FloorBeams2 = 0
        Core2 = 0
        costs_floors2 = 0
    else:
        Outrigger2 = predictions_S2[0][31]
        FloorBeams2 = predictions_S2[0][18]
        Core2 = predictions_S2[0][32]
        costs_floors2 = costs_floors

    if predictions_S2[0][2] == 0:
        Outrigger3 = 0
        FloorBeams3 = 0
        Core3 = 0
        costs_floors3 = 0
    else:
        Outrigger3 = predictions_S2[0][31]
        FloorBeams3 = predictions_S2[0][18]
        Core3 = predictions_S2[0][32]
        costs_floors3 = costs_floors

    if predictions_S2[0][3] == 0:
        Outrigger4 = 0
        FloorBeams4 = 0
        Core4 = 0
        costs_floors4 = 0
    else:
        Outrigger4 = predictions_S2[0][31]
        FloorBeams4 = predictions_S2[0][18]
        Core4 = predictions_S2[0][32]
        costs_floors4 = costs_floors

    if predictions_S2[0][4] == 0:
        Outrigger5 = 0
        FloorBeams5 = 0
        Core5 = 0
        costs_floors5 = 0
    else:
        Outrigger5 = predictions_S2[0][31]
        FloorBeams5 = predictions_S2[0][18]
        Core5 = predictions_S2[0][32]
        costs_floors5 = costs_floors

    if predictions_S2[0][5] == 0:
        Outrigger6 = 0
        FloorBeams6 = 0
        Core6 = 0
        costs_floors6 = 0
    else:
        Outrigger6 = predictions_S2[0][31]
        FloorBeams6 = predictions_S2[0][18]
        Core6 = predictions_S2[0][32]
        costs_floors6 = costs_floors


    ## Creating a DataFrame
    hz_S2_costs = {
            'Category': ['Height Zone 1', 'Height Zone 2', 'Height Zone 3', 'Height Zone 4', 'Height Zone 5', 'Height Zone 6'],
            'FloorBeams': [FloorBeams1, FloorBeams2, FloorBeams3, FloorBeams4, FloorBeams5, FloorBeams6],
            'OuterBeams': [predictions_S2[0][0], predictions_S2[0][1], predictions_S2[0][2], predictions_S2[0][3], predictions_S2[0][4], p
            'Floor System': [costs_floors1, costs_floors2, costs_floors3, costs_floors4, costs_floors5, costs_floors6],
            'InnerColumns_Grav': [(predictions_S2[0][25]*2), (predictions_S2[0][26]*2), (predictions_S2[0][27]*2), (predictions_S2[0][28]*
            'OuterColumns_Grav' : [predictions_S2[0][12], predictions_S2[0][13], predictions_S2[0][14], predictions_S2[0][15], predictions
            'OutriggerTruss': [Outrigger1, Outrigger2, Outrigger3, Outrigger4, Outrigger5, Outrigger6],
            'Core': [Core1, Core2, Core3, Core4, Core5, Core6],
            'InnerColumns_Stab': [(predictions_S2[0][19]-predictions_S2[0][25]), (predictions_S2[0][20]-predictions_S2[0][26]), (predictio
            'OuterColumns_Stab ': [predictions_S2[0][6], predictions_S2[0][7], predictions_S2[0][8], predictions_S2[0][9], predictions_S2[
        }

    HZ1_grav = FloorBeams1 + predictions_S2[0][0] + costs_floors1 + (predictions_S2[0][25]*2) + predictions_S2[0][12]
    HZ1_stab = Outrigger1 + Core1 + (predictions_S2[0][19]-predictions_S2[0][25]) + predictions_S2[0][6]
    ratio_stab_HZ1 = np.round(HZ1_stab / (HZ1_stab + HZ1_grav) * 100, 1)

    HZ2_grav = FloorBeams2 + predictions_S2[0][1] + costs_floors2 + (predictions_S2[0][26]*2) + predictions_S2[0][13]
    HZ2_stab = Outrigger2 + Core2 + (predictions_S2[0][20]-predictions_S2[0][26]) + predictions_S2[0][7]
    ratio_stab_HZ2 = np.round(HZ2_stab / (HZ2_stab + HZ2_grav) * 100, 1)
```

```
HZ3_grav = FloorBeams3 + predictions_S2[0][2] + costs_floors3 + (predictions_S2[0][27]*2) + predictions_S2[0][14]
HZ3_stab = Outrigger3 + Core3 + (predictions_S2[0][21]-predictions_S2[0][27]) + predictions_S2[0][8]
if HZ3_grav == 0:
    ratio_stab_HZ3 = 0
else:
    ratio_stab_HZ3 = np.round(HZ3_stab / (HZ3_stab + HZ3_grav) * 100, 1)

HZ4_grav = FloorBeams4 + predictions_S2[0][3] + costs_floors4 + (predictions_S2[0][28]*2) + predictions_S2[0][15]
HZ4_stab = Outrigger4 + Core4 + (predictions_S2[0][22]-predictions_S2[0][28]) + predictions_S2[0][9]
if HZ4_grav == 0:
    ratio_stab_HZ4 = 0
else:
    ratio_stab_HZ4 = np.round(HZ4_stab / (HZ4_stab + HZ4_grav) * 100, 1)

HZ5_grav = FloorBeams5 + predictions_S2[0][4] + costs_floors5 + (predictions_S2[0][29]*2) + predictions_S2[0][16]
HZ5_stab = Outrigger5 + Core5 + (predictions_S2[0][23]-predictions_S2[0][29]) + predictions_S2[0][10]
if HZ5_grav == 0:
    ratio_stab_HZ5 = 0
else:
    ratio_stab_HZ5 = np.round(HZ5_stab / (HZ5_stab + HZ5_grav) * 100, 1)

HZ6_grav = FloorBeams6 + predictions_S2[0][5] + costs_floors6 + (predictions_S2[0][30]*2) + predictions_S2[0][17]
HZ6_stab = Outrigger6 + Core6 + (predictions_S2[0][24]-predictions_S2[0][30]) + predictions_S2[0][11]
if HZ6_grav == 0:
    ratio_stab_HZ6 = 0
else:
    ratio_stab_HZ6 = np.round(HZ6_stab / (HZ6_stab + HZ6_grav) * 100, 1)

print('HZ1: Ratio of Structural Costs coming from', '\033[1;32m\033[1mStability System\033[0m', ratio_stab_HZ1, '%')
print('HZ2: Ratio of Structural Costs coming from', '\033[1;32m\033[1mStability System\033[0m', ratio_stab_HZ2, '%')
print('HZ3: Ratio of Structural Costs coming from', '\033[1;32m\033[1mStability System\033[0m', ratio_stab_HZ3, '%')
print('HZ4: Ratio of Structural Costs coming from', '\033[1;32m\033[1mStability System\033[0m', ratio_stab_HZ4, '%')
print('HZ5: Ratio of Structural Costs coming from', '\033[1;32m\033[1mStability System\033[0m', ratio_stab_HZ5, '%')
print('HZ6: Ratio of Structural Costs coming from', '\033[1;32m\033[1mStability System\033[0m', ratio_stab_HZ6, '%')


df_hz_costs = pd.DataFrame(hz_S2_costs)
df_hz_costs.set_index('Category', inplace = True)

## Plotting HZ1 and HZ2
ax_costs = df_hz_costs.plot(kind='bar', stacked=True, figsize=(7, 5), color = colors)
ax_costs.set_title('Structural Costs Predictions for each Height Zone')
ax_costs.set_xlabel('Height Zones')
ax_costs.set_ylabel('Predicted Structural Costs [€/m2GFA]')

ax_costs.set_xticklabels(ax_costs.get_xticklabels(), rotation=45, ha='right')

ax_costs.legend(loc='lower left', bbox_to_anchor=(1, 0))
ax_costs.set_ylim([0, 1000])

# Display the plot
plt.show()



#EMBODIED CARBON
predictions_S2, num_zones= best_model_S2.predict(input_data_scaled, input_data)

predictions_S2[:, 0:32] *= s_EmCO2*CO2_sprice
predictions_S2[:, 32] *= pc_rc_EmCO2*CO2_sprice

Outrigger1 = predictions_S2[0][31]
Core1 = predictions_S2[0][32]
EmCO2_floors1 = EmCO2_floors
FloorBeams1 = predictions_S2[0][18]


if predictions_S2[0][1] == 0:
    Outrigger2 = 0
    FloorBeams2 = 0
    Core2 = 0
    EmCO2_floors2 = 0
else:
    Outrigger2 = predictions_S2[0][31]
    FloorBeams2 = predictions_S2[0][18]
    Core2 = predictions_S2[0][32]
    EmCO2_floors2 = EmCO2_floors

if predictions_S2[0][2] == 0:
    Outrigger3 = 0
    FloorBeams3 = 0
    Core3 = 0
    EmCO2_floors3 = 0
else:
    Outrigger3 = predictions_S2[0][31]
    FloorBeams3 = predictions_S2[0][18]
    Core3 = predictions_S2[0][32]
    EmCO2_floors3 = EmCO2_floors

if predictions_S2[0][3] == 0:
    Outrigger4 = 0
    FloorBeams4 = 0
    Core4 = 0
    EmCO2_floors4 = 0
else:
    Outrigger4 = predictions_S2[0][31]
    FloorBeams4 = predictions_S2[0][18]
    Core4 = predictions_S2[0][32]
    EmCO2_floors4 = EmCO2_floors

if predictions_S2[0][4] == 0:
```

```
        Outrigger5 = 0
        FloorBeams5 = 0
        Core5 = 0
        EmCO2_floors5 = 0
    else:
        Outrigger5 = predictions_S2[0][31]
        FloorBeams5 = predictions_S2[0][18]
        Core5 = predictions_S2[0][32]
        EmCO2_floors5 = EmCO2_floors

    if predictions_S2[0][5] == 0:
        Outrigger6 = 0
        FloorBeams6 = 0
        Core6 = 0
        EmCO2_floors6 = 0
    else:
        Outrigger6 = predictions_S2[0][31]
        FloorBeams6 = predictions_S2[0][18]
        Core6 = predictions_S2[0][32]
        EmCO2_floors6 = EmCO2_floors


    ## Creating a DataFrame
    hz_S2_EmCO2 = {
            'Category': ['Height Zone 1', 'Height Zone 2', 'Height Zone 3', 'Height Zone 4', 'Height Zone 5', 'Height Zone 6'],
            'FloorBeams': [FloorBeams1, FloorBeams2, FloorBeams3, FloorBeams4, FloorBeams5, FloorBeams6],
            'OuterBeams': [predictions_S2[0][0], predictions_S2[0][1], predictions_S2[0][2], predictions_S2[0][3], predictions_S2[0][4], p
            'Floor System': [EmCO2_floors1, EmCO2_floors2, EmCO2_floors3, EmCO2_floors4, EmCO2_floors5, EmCO2_floors6],
            'InnerColumns_Grav': [predictions_S2[0][25]*2, predictions_S2[0][26]*2, predictions_S2[0][27]*2, predictions_S2[0][28]*2, pred
            'OuterColumns_Grav' : [predictions_S2[0][12], predictions_S2[0][13], predictions_S2[0][14], predictions_S2[0][15], predictions
            'OutriggerTruss': [Outrigger1, Outrigger2, Outrigger3, Outrigger4, Outrigger5, Outrigger6],
            'Core': [Core1, Core2, Core3, Core4, Core5, Core6],
            'InnerColumns_Stab': [(predictions_S2[0][19]-predictions_S2[0][25]), (predictions_S2[0][20]-predictions_S2[0][26]), (predictio
            'OuterColumns_Stab ': [predictions_S2[0][6], predictions_S2[0][7], predictions_S2[0][8], predictions_S2[0][9], predictions_S2[
        }

    df_hz_EmCO2 = pd.DataFrame(hz_S2_EmCO2)

    df_hz_EmCO2.set_index('Category', inplace = True)

    ## Plotting HZ1 and HZ2
    ax_EmCO2 = df_hz_EmCO2.plot(kind='bar', stacked=True, figsize=(7, 5), color = colors)
    ax_EmCO2.set_title('Environmental Costs Predictions for each Height Zone')
    ax_EmCO2.set_xlabel('Height Zones')
    ax_EmCO2.set_ylabel('Predicted Environmental Costs [€/m2GFA]')

    ax_EmCO2.set_xticklabels(ax_EmCO2.get_xticklabels(), rotation=45, ha='right')

    ax_EmCO2.legend(loc='lower left', bbox_to_anchor=(1, 0))
    ax_EmCO2.set_ylim([0, 15])


    # Display the plot
    plt.show()


# Plot the graphs and use the defined function
# Create interactive sliders for floorplan length and height
floorplan_length_slider = widgets.FloatSlider(value=20, min=5,
                                              max=75, step=1,
                                              description='Width of the Floorplan:')

floorplan_length_slider.layout.width = '40%'
floorplan_length_slider.style = {'description_width': '50%'}

height_slider = widgets.FloatSlider(value=100, min=mass_S1['in:Height [m]'].min(),
                                    max=500, step=1,
                                    description='Height:')

height_slider.layout.width = '40%'
height_slider.style = {'description_width': '50%'}


# Create an interactive output widget
output = widgets.interactive_output(make_predictions_perHZ_S2, {'floorplan_length': floorplan_length_slider, 'height': height_slider})

# Display the sliders and output
display(floorplan_length_slider, height_slider, output)
```

In [ ]:
```
# Function to make predictions
def make_predictions_perHZ(floorplan_length, height):
    # Prepare input data
    input_data = pd.DataFrame({
    'in:Length [m]': [floorplan_length],
    'in:Height [m]': [height]
    })

    input_data_scaled = scaler2.transform(input_data)
    predictions_S2, num_zones = best_model_S2.predict(input_data_scaled, input_data)

    #Determination height per height zone
    if num_zones == 1:
        hz1 = height
        pred_OB = predictions_S2[0][0]
        pred_OC_OutTruss = predictions_S2[0][6]
        pred_OC_noOutTruss = predictions_S2[0][12]
        pred_FB = predictions_S2[0][18]
        pred_IC_OutTruss = predictions_S2[0][19]
        pred_IC_noOutTruss = predictions_S2[0][25]
        pred_Outrigger = predictions_S2[0][31]
        pred_Core = predictions_S2[0][32]

    if num_zones == 2:
        hz1 = (floorplan_length // 3) * 3
        hz2 = height - hz1

        pred_OB = (predictions_S2[0][0]*hz1 + predictions_S2[0][1]*hz2) / height
        pred_OC_OutTruss = (predictions_S2[0][6]*hz1 + predictions_S2[0][7]*hz2) / height
        pred_OC_noOutTruss = (predictions_S2[0][12]*hz1 + predictions_S2[0][13]*hz2) / height
        pred_FB = predictions_S2[0][18]
        pred_IC_OutTruss = (predictions_S2[0][19]*hz1 + predictions_S2[0][20]*hz2) / height
        pred_IC_noOutTruss = (predictions_S2[0][25]*hz1 + predictions_S2[0][26]*hz2) / height
        pred_Outrigger = predictions_S2[0][31]
        pred_Core = predictions_S2[0][32]

    if num_zones == 3:
        hz1 = (floorplan_length // 3) * 3
        hz3 = (floorplan_length // 3) * 3
        hz2 = height - hz1 - hz3

        pred_OB = (predictions_S2[0][0]*hz1 + predictions_S2[0][1]*hz2 + predictions_S2[0][2]*hz3) / height
        pred_OC_OutTruss = (predictions_S2[0][6]*hz1 + predictions_S2[0][7]*hz2 + predictions_S2[0][8]*hz3) / height
        pred_OC_noOutTruss = (predictions_S2[0][12]*hz1 + predictions_S2[0][13]*hz2 + predictions_S2[0][14]*hz3) / height
        pred_FB = predictions_S2[0][18]
        pred_IC_OutTruss = (predictions_S2[0][19]*hz1 + predictions_S2[0][20]*hz2 + predictions_S2[0][21]*hz3) / height
        pred_IC_noOutTruss = (predictions_S2[0][25]*hz1 + predictions_S2[0][26]*hz2 + predictions_S2[0][27]*hz3) / height
        pred_Outrigger = predictions_S2[0][31]
        pred_Core = predictions_S2[0][32]

    if num_zones == 4:
        hz1 = (floorplan_length // 3) * 3
        hz4 = (floorplan_length // 3) * 3
        hz23 = height - hz1 - hz4
        hz2 = hz23 / 2
        hz3 = hz23 / 2

        pred_OB = (predictions_S2[0][0]*hz1 + predictions_S2[0][1]*hz2 + predictions_S2[0][2]*hz3 + predictions_S2[0][3]*hz4) / height
        pred_OC_OutTruss = (predictions_S2[0][6]*hz1 + predictions_S2[0][7]*hz2 + predictions_S2[0][8]*hz3 + predictions_S2[0][9]*hz4) / h
        pred_OC_noOutTruss = (predictions_S2[0][12]*hz1 + predictions_S2[0][13]*hz2 + predictions_S2[0][14]*hz3 + predictions_S2[0][15]*hz
        pred_FB = predictions_S2[0][18]
        pred_IC_OutTruss = (predictions_S2[0][19]*hz1 + predictions_S2[0][20]*hz2 + predictions_S2[0][21]*hz3 + predictions_S2[0][22]*hz4)
        pred_IC_noOutTruss = (predictions_S2[0][25]*hz1 + predictions_S2[0][26]*hz2 + predictions_S2[0][27]*hz3 + predictions_S2[0][28]*hz
        pred_Outrigger = predictions_S2[0][31]
        pred_Core = predictions_S2[0][32]


    if num_zones == 5:
        hz1 = (floorplan_length // 3) * 3
        hz5 = (floorplan_length // 3) * 3
        hz234 = height - hz1 - hz5
        hz2 = hz234 / 3
        hz3 = hz234 / 3
        hz4 = hz234 / 3

        pred_OB = (predictions_S2[0][0]*hz1 + predictions_S2[0][1]*hz2 + predictions_S2[0][2]*hz3 + predictions_S2[0][3]*hz4 + predictions
        pred_OC_OutTruss = (predictions_S2[0][6]*hz1 + predictions_S2[0][7]*hz2 + predictions_S2[0][8]*hz3 + predictions_S2[0][9]*hz4 + pr
        pred_OC_noOutTruss = (predictions_S2[0][12]*hz1 + predictions_S2[0][13]*hz2 + predictions_S2[0][14]*hz3 + predictions_S2[0][15]*hz
        pred_FB = predictions_S2[0][18]
        pred_IC_OutTruss= (predictions_S2[0][19]*hz1 + predictions_S2[0][20]*hz2 + predictions_S2[0][21]*hz3 + predictions_S2[0][22]*hz4 +
        pred_IC_noOutTruss = (predictions_S2[0][25]*hz1 + predictions_S2[0][26]*hz2 + predictions_S2[0][27]*hz3 + predictions_S2[0][28]*hz
        pred_Outrigger = predictions_S2[0][31]
        pred_Core = predictions_S2[0][32]


    if num_zones == 6:
        hz1 = (floorplan_length // 3) * 3
        hz6 = (floorplan_length // 3) * 3
        hz2345 = height - hz1 - hz6
        hz2 = hz2345 / 5
        hz3 = hz2345 / 5
        hz4 = hz2345 / 5
        hz5 = hz2345 / 5

        pred_OB = (predictions_S2[0][0]*hz1 + predictions_S2[0][1]*hz2 + predictions_S2[0][2]*hz3 + predictions_S2[0][3]*hz4 + predictions
        pred_OC_OutTruss = (predictions_S2[0][6]*hz1 + predictions_S2[0][7]*hz2 + predictions_S2[0][8]*hz3 + predictions_S2[0][9]*hz4 + pr
        pred_OC_noOutTruss = (predictions_S2[0][12]*hz1 + predictions_S2[0][13]*hz2 + predictions_S2[0][14]*hz3 + predictions_S2[0][15]*hz
        pred_FB = predictions_S2[0][18]
        pred_IC_OutTruss = (predictions_S2[0][19]*hz1 + predictions_S2[0][20]*hz2 + predictions_S2[0][21]*hz3 + predictions_S2[0][22]*hz4
        pred_IC_noOutTruss = (predictions_S2[0][25]*hz1 + predictions_S2[0][26]*hz2 + predictions_S2[0][27]*hz3 + predictions_S2[0][28]*hz
        pred_Outrigger = predictions_S2[0][31]
        pred_Core = predictions_S2[0][32]
```

```
pred_Costs_OB = pred_OB * s_price
pred_Costs_OC_OutTruss = pred_OC_OutTruss * s_price
pred_Costs_OC_noOutTruss = pred_OC_noOutTruss * s_price
pred_Costs_FB = pred_FB * s_price
pred_Costs_IC_OutTruss = pred_IC_OutTruss * s_price
pred_Costs_IC_noOutTruss = pred_IC_noOutTruss * s_price
pred_Costs_Outrigger = pred_Outrigger * s_price
pred_Costs_Core = pred_Core * pc_rc_price
remaining_value_IC_costs = pred_Costs_IC_OutTruss - pred_Costs_IC_noOutTruss
pred_Costs_IC_stab = remaining_value_IC_costs
pred_Costs_IC_grav = pred_Costs_IC_noOutTruss * 2

## Creating a DataFrame
StructuralElement_Costs_data = {
        'Category': ['FloorBeams', 'OuterBeams', 'Floor System', 'InnerColumns_Grav', 'OuterColumns_Grav',
                     'InnerColumns_Stab', 'OutriggerTruss', 'Core', 'OuterColumns_Stab'],
        'Costs per element': [pred_Costs_FB, pred_Costs_OB, costs_floors, pred_Costs_IC_grav, pred_Costs_OC_noOutTruss,
                              pred_Costs_IC_stab, pred_Costs_Outrigger, pred_Costs_Core, pred_Costs_OC_OutTruss]}
## Creating DataFrame
df_SE_Costs = pd.DataFrame(StructuralElement_Costs_data)

pred_Costs_stability = pred_Costs_Outrigger + pred_Costs_Core + pred_Costs_IC_stab + pred_Costs_OC_OutTruss
pred_Costs_gravity = pred_Costs_FB + pred_Costs_OB + costs_floors + pred_Costs_IC_grav + pred_Costs_OC_noOutTruss
part_stability = (pred_Costs_stability / (pred_Costs_stability+pred_Costs_gravity))*100

print('Total Structural Costs of','\033[1;34m\033[1mGravity System\033[0m', 'is', np.round(pred_Costs_gravity, 0), '€/m2GFA')
print('Total Structural Costs of', '\033[1;32m\033[1mStability System\033[0m', 'is', np.round(pred_Costs_stability, 0), '€/m2GFA')
print('Ratio of Structural Costs coming from Stability System:', np.round(part_stability,1), '%')

df_SE_Costs.set_index('Category', inplace = True)

colors = [color_blue1, color_blue2, color_blue3, color_blue4, color_blue5, color_green1, color_green2, color_green3, color_green4]

ax = df_SE_Costs.plot(kind='bar', stacked=True, figsize=(7, 5), color='white', legend=None)

# Set colors for each part of the bar
for p, col in zip(ax.patches, colors):
    p.set_facecolor(col)

ax.set_title('Predicted Structural Costs: Structural Element Split')
ax.set_xlabel('Structural Elements')
ax.set_ylabel('Predicted Structural Costs [€/m2GFA]')
ax.set_xticklabels(ax.get_xticklabels(), rotation=45, ha='right')
ax.set_ylim([0, 250])

for p in ax.patches:
    total_value = p.get_height()
    ax.annotate(f'{total_value:.1f}', (p.get_x() + p.get_width() / 2., total_value),
            ha='center', va='center', xytext=(0, 10), textcoords='offset points')


# Display the plot
plt.show()

# EMBODIED CARBON
pred_EmCO2_OB = pred_OB * s_EmCO2*CO2_sprice
pred_EmCO2_OC_OutTruss = pred_OC_OutTruss * s_EmCO2*CO2_sprice
pred_EmCO2_OC_noOutTruss = pred_OC_noOutTruss * s_EmCO2*CO2_sprice
pred_EmCO2_FB = pred_FB * s_EmCO2*CO2_sprice
pred_EmCO2_IC_OutTruss = pred_IC_OutTruss * s_EmCO2*CO2_sprice
pred_EmCO2_IC_noOutTruss = pred_IC_noOutTruss * s_EmCO2*CO2_sprice
pred_EmCO2_Outrigger = pred_Outrigger * s_EmCO2*CO2_sprice
pred_EmCO2_Core = pred_Core * pc_rc_EmCO2*CO2_sprice
remaining_value_IC_EmCO2 = pred_EmCO2_IC_OutTruss - pred_EmCO2_IC_noOutTruss
pred_EmCO2_IC_stab = remaining_value_IC_EmCO2
pred_EmCO2_IC_grav = pred_EmCO2_IC_noOutTruss * 2


## Creating a DataFrame for HZ1 and HZ2 predictions
StructuralElement_EmCO2_data = {
        'Category': ['FloorBeams', 'OuterBeams', 'Floor System', 'InnerColumns_Grav', 'OuterColumns_Grav',
                     'InnerColumns_Stab', 'OutriggerTruss', 'Core', 'OuterColumns_Stab'],
        'Costs per element': [pred_EmCO2_FB, pred_EmCO2_OB, EmCO2_floors, pred_EmCO2_IC_grav, pred_EmCO2_OC_noOutTruss,
                              pred_EmCO2_IC_stab, pred_EmCO2_Outrigger, pred_EmCO2_Core, pred_EmCO2_OC_OutTruss]}


pred_EmCO2_stability = pred_EmCO2_Outrigger + pred_EmCO2_Core + pred_EmCO2_IC_stab + pred_EmCO2_OC_OutTruss
pred_EmCO2_gravity = pred_EmCO2_FB + pred_EmCO2_OB + EmCO2_floors + pred_EmCO2_IC_noOutTruss + pred_EmCO2_OC_noOutTruss
part_stability = (pred_EmCO2_stability / (pred_EmCO2_stability+pred_EmCO2_gravity))*100

print('Total Environmental Costs of','\033[1;34m\033[1mGravity System\033[0m', 'is', np.round(pred_EmCO2_gravity, 0), '€/m2GFA')
print('Total Environmental Costs of', '\033[1;32m\033[1mStability System\033[0m', 'is', np.round(pred_EmCO2_stability, 0), '€/m2GFA')
print('Ratio of Environmental Costs coming from Stability System:', np.round(part_stability,1), '%')

## Creating DataFrame
df_SE_EmCO2 = pd.DataFrame(StructuralElement_EmCO2_data)

df_SE_EmCO2.set_index('Category', inplace = True)

colors = [color_blue1, color_blue2, color_blue3, color_blue4, color_blue5, color_green1, color_green2, color_green3, color_green4]

ax = df_SE_EmCO2.plot(kind='bar', stacked=True, figsize=(7, 5), color='white', legend=None)  # Initialize with white color

# Set colors for each part of the bar
for p, col in zip(ax.patches, colors):
    p.set_facecolor(col)

ax.set_title('Predicted Environmental Costs: Structural Element Split')
ax.set_xlabel('Structural Elements')
ax.set_ylabel('Predicted Environmental Costs [€/m2GFA]')
ax.set_xticklabels(ax.get_xticklabels(), rotation=45, ha='right')
```

```
    ax.set_ylim([0, 5])

    for p in ax.patches:
        total_value = p.get_height()
        ax.annotate(f'{total_value:.1f}', (p.get_x() + p.get_width() / 2., total_value),
                ha='center', va='center', xytext=(0, 10), textcoords='offset points')

    # Display the plot
    plt.show()


# Create interactive sliders for floorplan length and height
floorplan_length_slider = widgets.FloatSlider(value=20, min=5,
                                              max=75, step=1,
                                              description='Width of the Floorplan:')

floorplan_length_slider.layout.width = '40%'
floorplan_length_slider.style = {'description_width': '50%'}

height_slider = widgets.FloatSlider(value=100, min=mass_S1['in:Height [m]'].min(),
                                    max=500, step=1,
                                    description='Height of the Structure:')

height_slider.layout.width = '40%'
height_slider.style = {'description_width': '50%'}


# Create an interactive output widget
output = widgets.interactive_output(make_predictions_perHZ, {'floorplan_length': floorplan_length_slider, 'height': height_slider})

# Display the sliders and output
display(floorplan_length_slider, height_slider, output)
```

Analysis of Errors of S2

In [ ]:
```
constant_variable = 'in:Length [m]'
constant_value_width = (32.4) # Set the constant value of width

# Filter data for the specified constant value
filtered_data_S2 = mass_S2[mass_S2[constant_variable] == constant_value_width]
```

In [ ]:
```
constant_variable = 'in:Length [m]'
constant_value_width = (32.4) # Set the constant value of width

# Filter data for the specified constant value
filtered_data_S2 = mass_S2[mass_S2[constant_variable] == constant_value_width]

constant_variable = 'in:Height [m]'
constant_value_height = (120) # Set the constant value of height

# Filter data for the specified constant value
filtered_data_S2 = filtered_data_S2[filtered_data_S2[constant_variable] == constant_value_height]
masses_data_S2 = filtered_data_S2.iloc[:, 8:34]
num_zones = int(filtered_data_S2.iloc[:, 35])
floorplan_length = constant_value_width
height = constant_value_height

if num_zones == 1:
    hz1 = height
    mass_OB = masses_data_S2.iloc[0, 0]
    mass_OC = masses_data_S2.iloc[0, 6]
    mass_FB = masses_data_S2.iloc[0, 12]
    mass_IC = masses_data_S2.iloc[0, 18]
    mass_Outrigger = masses_data_S2.iloc[0, 24]
    mass_Core = masses_data_S2.iloc[0, 25]

if num_zones == 2:
    hz1 = (floorplan_length // 3) * 3
    hz2 = height - hz1

    mass_OB = (masses_data_S2.iloc[0, 0]*hz1 + masses_data_S2.iloc[0, 1]*hz2) / height
    mass_OC = (masses_data_S2.iloc[0, 6]*hz1 + masses_data_S2.iloc[0, 7]*hz2) / height
    mass_FB = (masses_data_S2.iloc[0, 12]*hz1 + masses_data_S2.iloc[0, 13]*hz2) / height
    mass_IC = (masses_data_S2.iloc[0, 18]*hz1 + masses_data_S2.iloc[0, 19]*hz2) / height
    mass_Outrigger = masses_data_S2.iloc[0, 24]
    mass_Core = masses_data_S2.iloc[0, 25]


if num_zones == 3:
    hz1 = (floorplan_length // 3) * 3
    hz3 = (floorplan_length // 3) * 3
    hz2 = height - hz1 - hz3

    mass_OB = (masses_data_S2.iloc[0, 0]*hz1 + masses_data_S2.iloc[0, 1]*hz2 + masses_data_S2.iloc[0, 2]*hz3) / height
    mass_OC = (masses_data_S2.iloc[0, 6]*hz1 + masses_data_S2.iloc[0, 7]*hz2 + masses_data_S2.iloc[0, 8]*hz3) / height
    mass_FB = (masses_data_S2.iloc[0, 12]*hz1 + masses_data_S2.iloc[0, 13]*hz2 + masses_data_S2.iloc[0, 14]*hz3) / height
    mass_IC = (masses_data_S2.iloc[0, 18]*hz1 + masses_data_S2.iloc[0, 19]*hz2 + masses_data_S2.iloc[0, 20]*hz3) / height
    mass_Outrigger = masses_data_S2.iloc[0, 24]
    mass_Core = masses_data_S2.iloc[0, 25]

if num_zones == 4:
    hz1 = (floorplan_length // 3) * 3
    hz4 = (floorplan_length // 3) * 3
    hz23 = height - hz1 - hz4
    hz2 = hz23 / 2
    hz3 = hz23 / 2

    mass_OB = (masses_data_S2.iloc[0, 0]*hz1 + masses_data_S2.iloc[0, 1]*hz2 + masses_data_S2.iloc[0, 2]*hz3 + masses_data_S2.iloc[0, 3]*h
    mass_OC = (masses_data_S2.iloc[0, 6]*hz1 + masses_data_S2.iloc[0, 7]*hz2 + masses_data_S2.iloc[0, 8]*hz3 + masses_data_S2.iloc[0, 9]*h
    mass_FB = (masses_data_S2.iloc[0, 12]*hz1 + masses_data_S2.iloc[0, 13]*hz2 + masses_data_S2.iloc[0, 14]*hz3 + masses_data_S2.iloc[0, 1
    mass_IC = (masses_data_S2.iloc[0, 18]*hz1 + masses_data_S2.iloc[0, 19]*hz2 + masses_data_S2.iloc[0, 20]*hz3 + masses_data_S2.iloc[0, 2
    mass_Outrigger = masses_data_S2.iloc[0, 24]
    mass_Core = masses_data_S2.iloc[0, 25]


if num_zones == 5:
    hz1 = (floorplan_length // 3) * 3
    hz5 = (floorplan_length // 3) * 3
    hz234 = height - hz1 - hz5
    hz2 = hz234 / 3
    hz3 = hz234 / 3
    hz4 = hz234 / 3

    mass_OB = (masses_data_S2.iloc[0, 0]*hz1 + masses_data_S2.iloc[0, 1]*hz2 + masses_data_S2.iloc[0, 2]*hz3 + masses_data_S2.iloc[0, 3]*h
    mass_OC = (masses_data_S2.iloc[0, 6]*hz1 + masses_data_S2.iloc[0, 7]*hz2 + masses_data_S2.iloc[0, 8]*hz3 + masses_data_S2.iloc[0, 9]*h
    mass_FB = (masses_data_S2.iloc[0, 12]*hz1 + masses_data_S2.iloc[0, 13]*hz2 + masses_data_S2.iloc[0, 14]*hz3 + masses_data_S2.iloc[0, 1
    mass_IC = (masses_data_S2.iloc[0, 18]*hz1 + masses_data_S2.iloc[0, 19]*hz2 + masses_data_S2.iloc[0, 20]*hz3 + masses_data_S2.iloc[0, 2
    mass_Outrigger = masses_data_S2.iloc[0, 24]
    mass_Core = masses_data_S2.iloc[0, 25]


if num_zones == 6:
    hz1 = (floorplan_length // 3) * 3
    hz6 = (floorplan_length // 3) * 3
    hz2345 = height - hz1 - hz6
    hz2 = hz2345 / 5
    hz3 = hz2345 / 5
    hz4 = hz2345 / 5
    hz5 = hz2345 / 5

    mass_OB = (masses_data_S2.iloc[0, 0]*hz1 + masses_data_S2.iloc[0, 1]*hz2 + masses_data_S2.iloc[0, 2]*hz3 + masses_data_S2.iloc[0, 3]*h
    mass_OC = (masses_data_S2.iloc[0, 6]*hz1 + masses_data_S2.iloc[0, 7]*hz2 + masses_data_S2.iloc[0, 8]*hz3 + masses_data_S2.iloc[0, 9]*h
    mass_FB = (masses_data_S2.iloc[0, 12]*hz1 + masses_data_S2.iloc[0, 13]*hz2 + masses_data_S2.iloc[0, 14]*hz3 + masses_data_S2.iloc[0, 1
    mass_IC = (masses_data_S2.iloc[0, 18]*hz1 + masses_data_S2.iloc[0, 19]*hz2 + masses_data_S2.iloc[0, 20]*hz3 + masses_data_S2.iloc[0, 2
    mass_Outrigger = masses_data_S2.iloc[0, 24]
    mass_Core = masses_data_S2.iloc[0, 25]

input_Costs_OB = mass_OB * s_price
input_Costs_OC = mass_OC * s_price
input_Costs_FB = mass_FB * s_price
input_Costs_IC = mass_IC * s_price
input_Costs_Outrigger = mass_Outrigger * s_price
input_Costs_Core = mass_Core * pc_rc_price
```

In [ ]:
```
# Function to make predictions
def make_predictions_perHZ(floorplan_length, height):
    # Prepare input data
    input_data = pd.DataFrame({
    'in:Length [m]': [floorplan_length],
    'in:Height [m]': [height]
    })

    input_data_scaled = scaler2.transform(input_data)
    predictions_S2, num_zones = best_model_S2.predict(input_data_scaled, input_data)

    #Determination height per height zone
    if num_zones == 1:
        hz1 = height
        pred_OB = predictions_S2[0][0]
        pred_OC = predictions_S2[0][6]
        pred_FB = predictions_S2[0][12]
        pred_IC = predictions_S2[0][18:24]
        pred_Outrigger = predictions_S2[0][24]
        pred_Core = predictions_S2[0][25]

    if num_zones == 2:
        hz1 = (floorplan_length // 3) * 3
        hz2 = height - hz1

        pred_OB = (predictions_S2[0][0]*hz1 + predictions_S2[0][1]*hz2) / height
        pred_OC = (predictions_S2[0][6]*hz1 + predictions_S2[0][7]*hz2) / height
        pred_FB = (predictions_S2[0][12]*hz1 + predictions_S2[0][13]*hz2) / height
        pred_IC = (predictions_S2[0][18]*hz1 + predictions_S2[0][19]*hz2) / height
        pred_Outrigger = predictions_S1[0][24]
        pred_Core = predictions_S2[0][25]

    if num_zones == 3:
        hz1 = (floorplan_length // 3) * 3
        hz3 = (floorplan_length // 3) * 3
        hz2 = height - hz1 - hz3

        pred_OB = (predictions_S2[0][0]*hz1 + predictions_S2[0][1]*hz2 + predictions_S2[0][2]*hz3) / height
        pred_OC = (predictions_S2[0][6]*hz1 + predictions_S2[0][7]*hz2 + predictions_S2[0][8]*hz3) / height
        pred_FB = (predictions_S2[0][12]*hz1 + predictions_S2[0][13]*hz2 + predictions_S2[0][14]*hz3) / height
        pred_IC = (predictions_S2[0][18]*hz1 + predictions_S2[0][19]*hz2 + predictions_S2[0][20]*hz3) / height
        pred_Outrigger = predictions_S2[0][24]
        pred_Core = predictions_S2[0][25]

    if num_zones == 4:
        hz1 = (floorplan_length // 3) * 3
        hz4 = (floorplan_length // 3) * 3
        hz23 = height - hz1 - hz4
        hz2 = hz23 / 2
        hz3 = hz23 / 2

        pred_OB = (predictions_S2[0][0]*hz1 + predictions_S2[0][1]*hz2 + predictions_S2[0][2]*hz3 + predictions_S2[0][3]*hz4) / height
        pred_OC = (predictions_S2[0][6]*hz1 + predictions_S2[0][7]*hz2 + predictions_S2[0][8]*hz3 + predictions_S2[0][9]*hz4) / height
        pred_FB = (predictions_S2[0][12]*hz1 + predictions_S2[0][13]*hz2 + predictions_S2[0][14]*hz3 + predictions_S2[0][15]*hz4) / height
        pred_IC = (predictions_S2[0][18]*hz1 + predictions_S2[0][19]*hz2 + predictions_S2[0][20]*hz3 + predictions_S2[0][21]*hz4) / height
        pred_Outrigger = predictions_S2[0][24]
        pred_Core = predictions_S2[0][25]


    if num_zones == 5:
        hz1 = (floorplan_length // 3) * 3
        hz5 = (floorplan_length // 3) * 3
        hz234 = height - hz1 - hz5
        hz2 = hz234 / 3
        hz3 = hz234 / 3
        hz4 = hz234 / 3

        pred_OB = (predictions_S2[0][0]*hz1 + predictions_S2[0][1]*hz2 + predictions_S2[0][2]*hz3 + predictions_S2[0][3]*hz4 + predictions
        pred_OC = (predictions_S2[0][6]*hz1 + predictions_S2[0][7]*hz2 + predictions_S2[0][8]*hz3 + predictions_S2[0][9]*hz4 + predictions
        pred_FB = (predictions_S2[0][12]*hz1 + predictions_S2[0][13]*hz2 + predictions_S2[0][14]*hz3 + predictions_S2[0][15]*hz4 + predict
        pred_IC = (predictions_S2[0][18]*hz1 + predictions_S2[0][19]*hz2 + predictions_S2[0][20]*hz3 + predictions_S2[0][21]*hz4 + predict
        pred_Outrigger = predictions_S2[0][24]
        pred_Core = predictions_S2[0][25]


    if num_zones == 6:
        hz1 = (floorplan_length // 3) * 3
        hz6 = (floorplan_length // 3) * 3
        hz2345 = height - hz1 - hz6
        hz2 = hz2345 / 5
        hz3 = hz2345 / 5
        hz4 = hz2345 / 5
        hz5 = hz2345 / 5

        pred_OB = (predictions_S2[0][0]*hz1 + predictions_S2[0][1]*hz2 + predictions_S2[0][2]*hz3 + predictions_S2[0][3]*hz4 + predictions
        pred_OC = (predictions_S2[0][6]*hz1 + predictions_S2[0][7]*hz2 + predictions_S2[0][8]*hz3 + predictions_S2[0][9]*hz4 + predictions
        pred_FB = (predictions_S2[0][12]*hz1 + predictions_S2[0][13]*hz2 + predictions_S2[0][14]*hz3 + predictions_S2[0][15]*hz4 + predict
        pred_IC = (predictions_S2[0][18]*hz1 + predictions_S2[0][19]*hz2 + predictions_S2[0][20]*hz3 + predictions_S2[0][21]*hz4 + predict
        pred_Outrigger = predictions_S2[0][24]
        pred_Core = predictions_S2[0][25]

    pred_Costs_OB = pred_OB #* s_price
    pred_Costs_OC = pred_OC #* s_price
    pred_Costs_FB = pred_FB #* s_price
    pred_Costs_IC = pred_IC #* s_price
    pred_Costs_Outrigger = pred_Outrigger #* s_price
    pred_Costs_Core = pred_Core #* pc_rc_price

    ## Creating a DataFrame
    StructuralElement_Costs_pred = {
            'Category': ['OuterBeam', 'OuterColumns', 'FloorBeams', 'InnerColumns', 'Outrigger', 'Core'],
            'Costs per Element': [pred_Costs_OB, pred_Costs_OC, pred_Costs_FB, pred_Costs_IC, pred_Costs_Outrigger,
```

```
                                    pred_Costs_Core]
        }

    df_pred_Costs = pd.DataFrame(StructuralElement_Costs_pred)
    df_pred_Costs.set_index('Category', inplace=True)

    StructuralElement_Costs_data = {
        'Category': ['OuterBeam', 'OuterColumns', 'FloorBeams', 'InnerColumns', 'Outrigger', 'Core'],
        'Costs per Element': [input_Costs_OB, input_Costs_OC, input_Costs_FB, input_Costs_IC, input_Costs_Outrigger,
                              input_Costs_Core]
    }

    df_input_Costs = pd.DataFrame(StructuralElement_Costs_data)
    df_input_Costs.set_index('Category', inplace=True)

    # Plotting the bar chart
    ax = df_pred_Costs.plot(kind='bar', stacked=True, figsize=(7, 5), color='white', legend=None)

    colors = [color_green,  # OuterBeams
              color_grey,  # OuterColumns
              color_yellow,  # FloorBeams
              color_darkblue,  # InnerColumns
              color_purple,    # Outrigger
              color_lightblue]  # Core

    # Set colors for each part of the bar
    for p, col in zip(ax.patches, colors):
        p.set_facecolor(col)

    # Add scatter points for the costs of structural elements
    ax.scatter(np.arange(len(df_input_Costs)), df_input_Costs['Costs per Element'], color=color_orange, zorder=5, label = 'S2: Input Data


    ax.set_title('S2: Predicted Costs: Structural Element Split')
    ax.set_xlabel('Structural Elements')
    ax.set_ylabel('Predicted Costs [€/GFA]')
    ax.set_xticklabels(ax.get_xticklabels(), rotation=45, ha='right')
    ax.legend(bbox_to_anchor=(1.02, 1), loc='upper left', borderaxespad=0)
    ax.set_ylim([0, 100])


    # Display the plot
    plt.show()

# Create interactive sliders for floorplan length and height
floorplan_length_slider = widgets.FloatSlider(value=20, min=5,
                                              max=75, step=1,
                                              description='Width of the Floorplan:')

floorplan_length_slider.layout.width = '40%'
floorplan_length_slider.style = {'description_width': '50%'}

height_slider = widgets.FloatSlider(value=100, min=mass_S1['in:Height [m]'].min(),
                                    max=500, step=1,
                                    description='Height of the Structure:')

height_slider.layout.width = '40%'
height_slider.style = {'description_width': '50%'}


# Create an interactive output widget
output = widgets.interactive_output(make_predictions_perHZ, {'floorplan_length': floorplan_length_slider, 'height': height_slider})

# Display the sliders and output
display(floorplan_length_slider, height_slider, output)
```

In [ ]:

In [ ]: