FOSSEE Summer Internship Report

On

Laced Column Module Devlopement

Submitted by

Sanket Sunil Gaikwad

3rd Year B.Tech Student, Department of Computer Engineering

Sanjivani University, Kopargaon

Under the Guidance of

Prof. Siddhartha Ghosh

Department of Civil Engineering

Indian Institute of Technology Bombay

Mentors:

Parth Karia

July 6, 2025

Acknowledgments

- I wish to express my heartfelt gratitude to everyone who supported me throughout the course of this project, especially Dr. Mahendra Gawali, Sir for introducing me to this opportunity. I am truly grateful for the contributions, time, and effort of all those who assisted me in various ways.
- Project staff at the Osdag team, Parth Karia, Ajmal Babu M. S. , and Ajinkya Dahale.
- Osdag Principal Investigator (PI) Prof. Siddhartha Ghosh, Department of Civil Engineering at IIT Bombay
- FOSSEE PI Prof. Kannan M. Moudgalya, FOSSEE Project Investigator, Department of Chemical Engineering, IIT Bombay
- FOSSEE Managers Usha Viswanathan and Vineeta Parmar and their entire team
- I would like to sincerely acknowledge the support provided by the National Mission on Education through Information and Communication Technology (NMEICT), Ministry of Education (MoE), Government of India. Their valuable contribution and initiative in promoting educational innovation and digital learning played a vital role in facilitating the successful execution of this project.
- I sincerely thank my colleagues for making my internship a great experience. I am particularly grateful to Rajesh Dalai and Aditya Donde whose guidance and collaboration enabled me to make a significant contribution to the development of the module.
- I would like to take this opportunity to express my sincere gratitude to the Department of Computer Engineering, Sanjivani University, for their unwavering support

and guidance throughout the course of my studies.

Contents

1 Introduction **5**

1.1 National Mission in Education through ICT 5

1.1.1 ICT Initiatives of MoE 6

1.2 FOSSEE Project 7

1.2.1 Projects and Activities 7

1.2.2 Fellowships 7

1.3 Osdag Software 8

1.3.1 Osdag GUI 9

1.3.2 Features 9

2 Screening Task **10**

2.1 Problem Statement 10

2.2 Introduction: 10

2.3 Data Source and Processing: 10

2.4 Methodology: 11

2.5 Shear Force Diagram (SFD): 11

2.6 Bending Moment Diagram (BMD): 11

2.7 Conclusion: 12

3 Module Development - Compression Members (Laced Columns) **13**

3.1 Problem Statement 13

3.1.1 Inputs 13

3.1.2 Outputs 14

3.1.3 Calculation Procedure Overview 15

3.2 Tasks Done 16

3.2.1 Python Code 16

3.2.2 Python Code 18

3.2.3 Explanation of the Code 19

3.2.4 Full code 19

4 Debugging and Documentation **145**

4.1 Problem Statement 145
4.2 Tasks Done 146

5 Conclusions **147**
5.1 Tasks Accomplished 147
5.2 Skills Developed 148

A Appendix **149**
A.1 Work Reports 149

Bibliography **151**

Chapter 1

Introduction

1.1 National Mission in Education through ICT

The National Mission on Education through ICT (NMEICT) is a scheme under the Department of Higher Education, Ministry of Education, Government of India. It aims to leverage the potential of ICT to enhance teaching and learning in Higher Education Institutions in an anytime-anywhere mode.

The mission aligns with the three cardinal principles of the Education Policy—**access, equity, and quality**—by:

- Providing connectivity and affordable access devices for learners and institutions.
- Generating high-quality e-content free of cost.

NMEICT seeks to bridge the digital divide by empowering learners and teachers in urban and rural areas, fostering inclusivity in the knowledge economy. Key focus areas include:

- Development of e-learning pedagogies and virtual laboratories.
- Online testing, certification, and mentorship through accessible platforms like EduSAT and DTH.
- Training and empowering teachers to adopt ICT-based teaching methods.

For further details, visit the official website: www.nmeict.ac.in.

1.1.1 ICT Initiatives of MoE

The Ministry of Education (MoE) has launched several ICT initiatives aimed at students, researchers, and institutions. The table below summarizes the key details:

No.	Resource	For Students/Researchers	For Institutions
Audio-Video e-content			
1	SWAYAM	Earn credit via online courses	Develop and host courses; accept credits
2	SWAYAMPRABHA	Access 24x7 TV programs	Enable SWAYAMPRABHA viewing facilities
Digital Content Access			
3	National Digital Library	Access e-content in multiple disciplines	List e-content; form NDL Clubs
4	e-PG Pathshala	Access free books and e-content	Host e-books
5	Shodhganga	Access Indian research theses	List institutional theses
6	e-ShodhSindhu	Access full-text e-resources	Access e-resources for institutions
Hands-on Learning			
7	e-Yantra	Hands-on embedded systems training	Create e-Yantra labs with IIT Bombay
8	FOSSEE	Volunteer for open-source software	Run labs with open-source software
9	Spoken Tutorial	Learn IT skills via tutorials	Provide self-learning IT content
10	Virtual Labs	Perform online experiments	Develop curriculum-based experiments
E-Governance			
11	SAMARTH ERP	Manage student lifecycle digitally	Enable institutional e-governance
Tracking and Research Tools			
12	VIDWAN	Register and access experts	Monitor faculty research outcomes
13	Shodh Shuddhi	Ensure plagiarism-free work	Improve research quality and reputation
14	Academic Bank of Credits	Store and transfer credits	Facilitate credit redemption

Table 1.1: Summary of ICT Initiatives by the Ministry of Education

1.2 FOSSEE Project

The FOSSEE (Free/Libre and Open Source Software for Education) project promotes the use of FLOSS tools in academia and research. It is part of the National Mission on Education through Information and Communication Technology (NMEICT), Ministry of Education (MoE), Government of India.

1.2.1 Projects and Activities

The FOSSEE Project supports the use of various FLOSS tools to enhance education and research. Key activities include:

- **Textbook Companion**: Porting solved examples from textbooks using FLOSS.
- **Lab Migration**: Facilitating the migration of proprietary labs to FLOSS alternatives.
- **Niche Software Activities**: Specialized activities to promote niche software tools.
- **Forums**: Providing a collaborative space for users.
- **Workshops and Conferences**: Organizing events to train and inform users.

1.2.2 Fellowships

FOSSEE offers various internship and fellowship opportunities for students:

- Winter Internship
- Summer Fellowship
- Semester-Long Internship

Students from any degree and academic stage can apply for these internships. Selection is based on the completion of screening tasks involving programming, scientific computing, or data collection that benefit the FLOSS community. These tasks are designed to be completed within a week.

For more details, visit the official FOSSEE website.

Figure 1.1: FOSSEE Projects and Activities

1.3 Osdag Software

Osdag (Open steel design and graphics) is a cross-platform, free/libre and open-source software designed for the detailing and design of steel structures based on the Indian Standard IS 800:2007. It allows users to design steel connections, members, and systems through an interactive graphical user interface (GUI) and provides 3D visualizations of designed components. The software enables easy export of CAD models to drafting tools for construction/fabrication drawings, with optimized designs following industry best practices [1, 2, 3]. Built on Python and several Python-based FLOSS tools (e.g., PyQt and PythonOCC), Osdag is licensed under the GNU Lesser General Public License (LGPL) Version 3.

1.3.1 Osdag GUI

The Osdag GUI is designed to be user-friendly and interactive. It consists of

- **Input Dock**: Collects and validates user inputs.
- **Output Dock**: Displays design results after validation.
- **CAD Window**: Displays the 3D CAD model, where users can pan, zoom, and rotate the design.
- **Message Log**: Shows errors, warnings, and suggestions based on design checks.

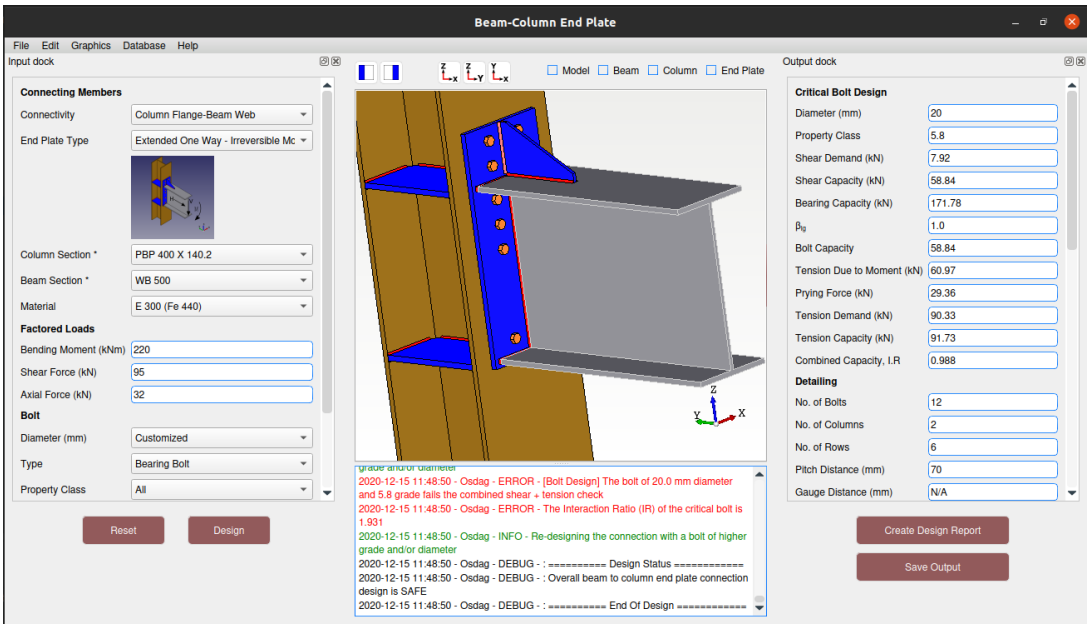


Figure 1.2: Osdag GUI

1.3.2 Features

- **CAD Model**: The 3D CAD model is color-coded and can be saved in multiple formats such as IGS, STL, and STEP.
- **Design Preferences**: Customizes the design process, with advanced users able to set preferences for bolts, welds, and detailing.
- **Design Report**: Creates a detailed report in PDF format, summarizing all checks, calculations, and design details, including any discrepancies.

For more details, visit the official Osdag website.

Chapter 2

Screening Task

2.1 Problem Statement

Shear Force and Bending Moment Diagram

2.2 Introduction:

Shear Force Diagram (SFD) and Bending Moment Diagram (BMD) are fundamental tools in structural analysis, helping engineers determine the effects of forces on beams.This report presents a Python-based approach to generate SFD and BMD using an Excel dataset and Matplotlib for visualization.

2.3 Data Source and Processing:

The dataset is stored in an Excel file (SFS Screening SFDBMD.xlsx) containing:

- **Distance (m):** Position along the beam.
- **Shear Force (kN):** Forces acting along the beam.
- **Bending Moment (kN-m):** Moments causing beam bending.

2.4 Methodology:

- Read the Excel file and extract relevant columns.
- Process data for numerical consistency.
- Plot the SFD and BMD using Matplotlib.

2.5 Shear Force Diagram (SFD):

The Shear Force Diagram represents the variation of shear force along the beam. Features include:

- A blue line with markers indicating shear force values.
- A shaded area highlighting the magnitude of shear force.
- A horizontal reference line (y = 0) for visualization
- **Observations:**
- The shear force is positive at the left support.
- Sudden jumps indicate load applications.
- The force decreases towards the right end.

2.6 Bending Moment Diagram (BMD):

The BMD shows the bending moment variation along the beam. Features include:

- A red line with markers representing bending moments.
- A filled region indicating moment intensity.
- A reference x-axis for zero moments.
- **Observations:**
- The bending moment starts at zero at one end.

- It reaches a minimum point indicating maximum bending.
- The moment increases towards the other end.

2.7 Conclusion:

This project successfully generates SFD and BMD using Python, providing crucial insights for structural analysis. These diagrams help in designing safe and efficient structures.

Chapter 3

Module Development - Compression Members (Laced Columns)

3.1 Problem Statement

The objective is to develop a **design code** for the structural design of **laced columns** as per IS:800-2007 guidelines, using the methodology outlined in the OSDAG Design and Detailing Checklist (DDCL). The code must perform comprehensive calculations to determine the appropriate member sections, design forces, lacing arrangements, and connection details for a compression member subjected to axial load.

3.1.1 Inputs

The following design parameters are to be provided by the user:

- **Section Profile**: (2-channels back-to-back or toe-to-toe)
- **Section Size**: (All,Customized)
- **Material Grade**: (e.g., E250, E350)
- **Axial Load (AF)**: Factored axial load in kN
- **Unsupported Lengths (in mm)**:
  - Along y–y axis
  - Along z–z axis

- **Boundary Conditions** (at both ends for both axes)
- **Lacing Details**:
  - Lacing Pattern (single, double, with/without tie)
  - Lacing Profile (angle, channel, flat)
- **Type of Connection**: Bolted or Welded
- **Design Preferences** (optional):
  - Effective Area Parameter (0.1 to 1.0)
  - Allowable Utilization Ratio
  - Bolt Diameter
  - Weld Size

3.1.2 Outputs

The expected outputs from the program are:

- Effective Length and Slenderness Ratios

  The code outputs effective lengths in both YY and ZZ directions

  Slenderness ratios are calculated and displayed for both YY and ZZ axes

- Section Classification

  Shows section class (Plastic, Semi-Compact, etc.)

- Design Values

  Design compressive stress (fcd)

  Design compressive strength

- Buckling Curve Classification

  Shows buckling curves for both YY and ZZ axes

- Channel and Lacing Details

  Spacing between channels

- Tie Plate Details

  Depth, thickness, and length of tie plates

- Lacing Spacing

  Shows the spacing between lacings (L0)

- Imperfection Factors and Euler Buckling Stress

- Stress Reduction Factors and Design Compressive Stress Values

3.1.3 Calculation Procedure Overview

The code must execute the following stages, aligned with IS:800-2007 and DDCL guidelines:

1. **Initial Section Selection**: Based on factored axial load and required effective area.
2. **Section Classification**: As per IS:800-2007 Table 2.
3. **Buckling Class Identification and Imperfection Factor**: From IS:800-2007 Table 7 and 10.
4. **Slenderness Ratio Calculation**: Using effective length factors from boundary conditions.
5. **Stress and Strength Calculation**:
   - Compute λ_e, λ, ϕ, χ, f_{cd}, and P_d
6. **Geometric Design**: Determine spacing S and tie plate dimensions.
7. **Lacing Design**:
   - Calculate L_0, N_L, angle θ, and compressive force P_{cal}
   - Check slenderness and strength in both compression and tension
8. **Connection Design**:
   - For bolts: evaluate V_{dsb}, V_{dpb} and determine number of bolts
   - For welds: determine strength and required weld length

The final code should modularly implement all the above steps and validate each design parameter against the limits prescribed by IS:800-2007.

3.2 Tasks Done

After a detailed study of the Design and Detailing Checklist (DDCL) to check for errors and understand the algorithm and the functional flow of the original design methodology, I translated each task into discrete Python functions to ensure modularity and clarity while preserving all structural design requirements. I developed a parallel computational flow tailored specifically for Python implementation maintaining the functionality of the original algorithm.

3.2.1 Python Code

This section presents the Python script developed for the design of laced compound columns as per IS:800-2007, integrated within the Osdag framework. The script handles design inputs, performs calculations, and displays output results through the GUI. It includes provisions for design preferences, validation, and report generation.

Description of the Script

The script is implemented as a class `LacedColumn`, which inherits from the base `Member` class in Osdag. It facilitates the GUI-based input of design parameters such as section properties, axial load, boundary conditions, material grade, and lacing configuration.

Key features of the script:

- **Input Parameters:** User-defined section profile, axial load, unsupported lengths, end conditions, lacing type, connection type, material grade, and design preferences (e.g., bolt diameter, weld size).
- **Design Calculations:** Computation of effective lengths, slenderness ratios, imperfection factors, buckling curves, design compressive stress and strength, and lacing force checks as per IS:800-2007 and DDCL methodology.
- **Output:** Outputs include utilization ratio, section classification, stress values, and verification results, all shown in the GUI and included in the PDF report.

Python Code

Listing 3.2: Laced Column Design in Osdag

```
class LacedColumn(Member):
    def __init__(self):
        super().__init__()
        self.design_status = False
        self.result = {}
        self.utilization_ratio = 0
        self.section_designation = None
        self.material_combo = QComboBox()
        self.axial_load_lineedit = QLineEdit()
        self.unsupported_length_yy_lineedit = QLineEdit()
        self.unsupported_length_zz_lineedit = QLineEdit()
        self.lacing_pattern_combo = QComboBox()
        self.connection_combo = QComboBox()
        ...

    def input_values(self, *args, **kwargs):
        options_list = []
        options_list.append((KEY_SEC_PROFILE, "Section
            Profile", TYPE_COMBOBOX, ["Back-to-Back", "Toe-to-
            Toe"], True, 'No Validator'))
        options_list.append((KEY_AXIAL, "Axial Load (kN)",
            TYPE_TEXTBOX, None, True, 'Float Validator'))
        ...
        return options_list

    def output_values(self, flag):
        out_list = []
        out_list.append((KEY_EFF_LEN_YY, "Effective Length (
            YY)", TYPE_TEXTBOX, self.effective_length_yy, True
            ))
        out_list.append((KEY_SLENDER_YY, "Slenderness Ratio (
```

```
            YY)", TYPE_TEXTBOX, self.effective_sr_yy, True))
        out_list.append((KEY_FCD, "Design Compressive Stress"
            , TYPE_TEXTBOX, self.result_fcd, True))
        ...
        return out_list
```

Explanation of the Code

- **Lines 1–13:** Defines the `LacedColumn` class and initializes GUI elements and internal variables.
- **Lines 15–22:** The `input_values` method specifies the GUI form layout for input parameters such as section profile and axial load.
- **Lines 24–30:** The `output_values` method formats the final design results to be displayed in the output dock of the Osdag GUI.

3.2.2 Python Code

The Python script is shown below. Each section is commented for clarity.

Listing 3.1: Laced Column Design in Osdag

```
1
2  import os
3  import sys
4  import math
5  from osdag.design_shear import ShearConnection
6  from osdag.connection_geometry import ConnectionGeometry
7
8  # Input parameters
9  beam_section = "ISMB 300"  # Beam section
10 column_section = "ISWB 500"  # Column section
11 load = 250  # Shear load in kN
12 bolt_diameter = 16  # Diameter of bolts in mm
13 bolt_grade = 8.8  # Grade of bolts
14
15 # Connection design
```

```
connection = ShearConnection(
    beam_section, column_section, load, bolt_diameter, bolt_grade
)
connection.calculate_bolt_group()
connection.verify_end_plate()
connection.verify_bolt_strength()

# Output results
print("Number of bolts required:", connection.number_of_bolts)
print("Bolt arrangement (rows x columns):", connection.
    bolt_arrangement)
print("End plate adequacy:", connection.end_plate_adequacy)
print("Bolt group adequacy:", connection.bolt_group_adequacy)
```

3.2.3 Explanation of the Code

– **Line 1-5**: Imports necessary libraries and Osdag modules for design calculations.

– **Line 7-11**: Defines the input parameters, including the beam and column sections, applied load, and bolt properties.

– **Line 13-18**: Creates a `ShearConnection` object and performs design calculations for the bolt group, end plate, and bolt strength.

– **Line 20-23**: Prints the results, including the number of bolts, bolt arrangement, and adequacy checks for the connection components.

3.2.4 Full code

```
"""
Main module: Design of Compression Member
Sub-module:  Design of column (loaded axially)

@author:Sanket Gaikwad

Reference:
```

```
            1) IS 800: 2007 General construction in steel -
               Code of practice (Third revision)

"""
import logging
import math
import numpy as np
from PyQt5.QtWidgets import QTextEdit, QMessageBox, QLineEdit
   , QComboBox, QDialog, QVBoxLayout, QListWidget,
   QDialogButtonBox, QLabel
from PyQt5.QtCore import QObject, pyqtSignal
from ...Common import *
from ..connection.moment_connection import MomentConnection
from ...utils.common.material import *
from ...utils.common.load import Load
from ...utils.common.component import ISection, Material
from ...utils.common.component import *
from ..member import Member
from ...Report_functions import *
from ...utils.common.common_calculation import *
from ...utils.common import is800_2007
from ...design_report.reportGenerator_latex import
   CreateLatex
from ...Common import TYPE_TAB_4, TYPE_TAB_5
from PyQt5.QtWidgets import QLineEdit, QMainWindow,
   QMessageBox
from PyQt5.QtCore import Qt
from PyQt5.QtGui import QValidator, QDoubleValidator
from PyQt5.QtWidgets import QComboBox
from PyQt5.QtWidgets import QDialog, QVBoxLayout, QLabel,
   QPushButton, QFormLayout, QTableWidget, QTableWidgetItem,
   QListWidget, QHBoxLayout
from PyQt5.QtWidgets import QDialogButtonBox
import os
```

```
import traceback
from ...utils.common.material import Material
from ...Common import KEY_LACING_SECTION_DIM

class LacedColumn(Member):
    def reset_output_state(self):
        self.effective_length_yy = None
        self.effective_length_zz = None
        self.effective_sr_yy = None
        self.effective_sr_zz = None
        self.result_fcd = None
        self.result_capacity = None
        self.result_UR = None
        self.result_section_class = None
        self.result_effective_area = None
        self.result_bc_yy = None
        self.result_bc_zz = None
        self.result_IF_yy = None
        self.result_IF_zz = None
        self.result_ebs_yy = None
        self.result_ebs_zz = None
        self.result_nd_esr_yy = None
        self.result_nd_esr_zz = None
        self.result_phi_yy = None
        self.result_phi_zz = None
        self.result_srf_yy = None
        self.result_srf_zz = None
        self.result_fcd_1_yy = None
        self.result_fcd_1_zz = None
        self.result_fcd_2 = None
        self.result_cost = None
        self.result = {}
        self.optimum_section_ur_results = {}
```

```
        # Any other custom/calculated fields should be reset
            here as well
    def __init__(self):
        super().__init__()
        self.material_property = None  # Defensive: always
            initialize
        self.design_status = False
        self.failed_reason = None  # Track why design failed
        self.result = {}
        self.utilization_ratio = 0
        self.area = 0
        self.epsilon = 1.0
        self.fy = 0
        self.section = None
        self.material = {}
        self.weld_size = ''
        self.weld_type = ''
        self.weld_strength = 0
        self.lacing_incl_angle = 0
        self.lacing_section = ''
        self.lacing_type = ''
        self.allowed_utilization = ''
        self.module = KEY_DISP_COMPRESSION_LacedColumn
        self.mainmodule = 'Member'
        self.section_designation = None
        self.dialogs = []  # Track all dialogs/windows opened
            by this module
        self.design_pref_dialog = None
        self.output_title_fields = {}
        self.design_pref_dictionary = {
            KEY_DISP_LACEDCOL_WELD_SIZE: "5mm",
            KEY_DISP_LACEDCOL_BOLT_DIAMETER: "16mm",
            KEY_DISP_LACEDCOL_EFFECTIVE_AREA: "1.0",
            KEY_DISP_LACEDCOL_ALLOWABLE_UR: "1.0"
```

```
        }
        self.design_pref = {}  # Ensure design_pref is always
            defined
        # Define input line edits for unsupported lengths and
            axial load
        self.unsupported_length_yy_lineedit = QLineEdit()
        self.unsupported_length_zz_lineedit = QLineEdit()
        self.axial_load_lineedit = QLineEdit()
        # Define combo boxes for dropdowns
        self.material_combo = QComboBox()
        self.connection_combo = QComboBox()
        self.lacing_pattern_combo = QComboBox()
        self.section_profile_combo = QComboBox()
        self.section_designation_combo = QComboBox()
        self.flange_class = None
        self.web_class = None
        self.gamma_m0 = 1.1  # As per IS 800:2007, Table 5
           for yield stress
        self.optimum_section_cost_results = {}  # Initialize
           to avoid AttributeError
        self.optimum_section_cost = []  # For cost-based
           optimization, as in other modules

##################################################
# Design Preference Functions Start
##################################################
    def tab_list(self):
        """
        Returns list of tabs for design preferences, matching
            flexure.py exactly.
        """
        tabs = []
        # Column Section tab (use tab_section for flexure-
           like behavior)
```

```
        t1 = (KEY_DISP_COLSEC, TYPE_TAB_1, self.tab_section)
        tabs.append(t1)
        # Weld Preferences tab (keep as is)
        t2 = ("Weld Preferences", TYPE_TAB_4, self.
           all_weld_design_values)
        tabs.append(t2)
        return tabs

    def all_weld_design_values(self, *args):
        return [
            (KEY_DISP_LACEDCOL_LACING_PROFILE_TYPE, "Lacing
               Profile Type", TYPE_COMBOBOX, ["Angle", "
               Channel", "Flat"], True, 'No Validator'),
            (KEY_DISP_LACEDCOL_LACING_PROFILE, "Lacing
               Profile Section", TYPE_COMBOBOX_CUSTOMIZED,
               self.get_lacing_profiles, True, 'No Validator'
               ),
            (KEY_DISP_LACEDCOL_EFFECTIVE_AREA, "Effective
               Area Parameter", TYPE_COMBOBOX, ["1.0", "0.9",
                "0.8", "0.7", "0.6", "0.5", "0.4", "0.3", "
               0.2", "0.1"], True, 'No Validator'),
            (KEY_DISP_LACEDCOL_ALLOWABLE_UR, "Allowable
               Utilization Ratio", TYPE_COMBOBOX, ["1.0", "
               0.95", "0.9", "0.85"], True, 'No Validator'),
            (KEY_DISP_LACEDCOL_BOLT_DIAMETER, "Bolt Diameter"
               , TYPE_COMBOBOX, ["16mm", "20mm", "24mm", "27
               mm"], True, 'No Validator'),
            (KEY_DISP_LACEDCOL_WELD_SIZE, "Weld Size",
               TYPE_COMBOBOX, ["4mm", "5mm", "6mm", "8mm"],
               True, 'No Validator')
        ]

    def tab_value_changed(self):
        """
```

```
        Returns list of tuples for tab value changes,
           matching flexure.py exactly.
        """
        change_tab = []
        # Section material changes (auto-populate fu, fy)
        t1 = (KEY_DISP_COLSEC, [KEY_SEC_MATERIAL], [
           KEY_SEC_FU, KEY_SEC_FY], TYPE_TEXTBOX, self.
           get_fu_fy_I_section)
        change_tab.append(t1)
        # Section properties update (I-section properties)
        t2 = (KEY_DISP_COLSEC, ['Label_1', 'Label_2', '
           Label_3', 'Label_4', 'Label_5'],
              ['Label_11', 'Label_12', 'Label_13', 'Label_14'
                 , 'Label_15', 'Label_16', 'Label_17', '
                Label_18',
               'Label_19', 'Label_20', 'Label_21', 'Label_22'
                 , KEY_IMAGE], TYPE_TEXTBOX, self.
                get_I_sec_properties)
        change_tab.append(t2)
        # SHS/RHS properties update
        t3 = (KEY_DISP_COLSEC, ['Label_HS_1', 'Label_HS_2', '
           Label_HS_3'],
              ['Label_HS_11', 'Label_HS_12', 'Label_HS_13', '
                 Label_HS_14', 'Label_HS_15', 'Label_HS_16',
                 'Label_HS_17', 'Label_HS_18',
               'Label_HS_19', 'Label_HS_20', 'Label_HS_21', '
                  Label_HS_22', KEY_IMAGE], TYPE_TEXTBOX,
                  self.get_SHS_RHS_properties)
        change_tab.append(t3)
        # CHS properties update
        t4 = (KEY_DISP_COLSEC, ['Label_CHS_1', 'Label_CHS_2',
            'Label_CHS_3'],
              ['Label_CHS_11', 'Label_CHS_12', 'Label_CHS_13'
                 , 'Label_HS_14', 'Label_HS_15', 'Label_HS_16
```

```
            ', 'Label_21', 'Label_22',
            KEY_IMAGE], TYPE_TEXTBOX, self.
               get_CHS_properties)
        change_tab.append(t4)
        # Section source update (when section designation
           changes)
        t5 = (KEY_DISP_COLSEC, [KEY_SECSIZE], [KEY_SOURCE],
           TYPE_TEXTBOX, self.change_source)
        change_tab.append(t5)
        return change_tab

    def input_dictionary_design_pref(self):
        """
        Returns list of tuples for design preferences,
           matching flexure.py exactly.
        """
        design_input = []
        # Section material (combobox)
        t1 = (KEY_DISP_COLSEC, TYPE_COMBOBOX, [
           KEY_SEC_MATERIAL])
        design_input.append(t1)
        # Section properties (fu, fy)
        t2 = (KEY_DISP_COLSEC, TYPE_TEXTBOX, [KEY_SEC_FU,
           KEY_SEC_FY])
        design_input.append(t2)
        return design_input

    def input_dictionary_without_design_pref(self, *args, **
       kwargs):
        """
        Returns list of tuples for input dictionary without
           design preferences, matching flexure.py pattern.
        """
        design_input = []
```

```
        # Material input with safe defaults
        t1 = (KEY_MATERIAL, [KEY_SEC_MATERIAL], 'Input Dock')
        design_input.append(t1)

        # Weld preferences with safe defaults
        t2 = (None, [
                KEY_DISP_LACEDCOL_LACING_PROFILE_TYPE,
                KEY_DISP_LACEDCOL_LACING_PROFILE,
                KEY_DISP_LACEDCOL_EFFECTIVE_AREA,
                KEY_DISP_LACEDCOL_ALLOWABLE_UR,
                KEY_DISP_LACEDCOL_BOLT_DIAMETER,
                KEY_DISP_LACEDCOL_WELD_SIZE
            ], '')
        design_input.append(t2)

        return design_input

    def refresh_input_dock(self):
        """
        Returns list of tuples for refreshing input dock,
           matching flexure.py exactly.
        """
        add_buttons = []
        # Add section designation combobox for column section
            tab
        t1 = (KEY_DISP_COLSEC, KEY_SECSIZE, TYPE_COMBOBOX,
           KEY_SECSIZE, None, None, "Columns")
        add_buttons.append(t1)
        return add_buttons

    def get_values_for_design_pref(self, key,
       design_dictionary):
        """
```

```
        Returns default values for design preferences,
            matching flexure.py pattern.
        """
        if not design_dictionary or design_dictionary.get(
            KEY_MATERIAL, 'Select Material') == 'Select
            Material':
             fu = ''
             fy = ''
        else:
             material = Material(design_dictionary[
                KEY_MATERIAL], 41)
             fu = material.fu
             fy = material.fy

        val = {
             KEY_SECSIZE: 'Select Section',
             KEY_DISP_LACEDCOL_LACING_PROFILE_TYPE: "Angle",
             KEY_DISP_LACEDCOL_LACING_PROFILE: "ISA 40x40x5",
             KEY_DISP_LACEDCOL_EFFECTIVE_AREA: "1.0",
             KEY_DISP_LACEDCOL_ALLOWABLE_UR: "1.0",
             KEY_DISP_LACEDCOL_BOLT_DIAMETER: "16mm",
             KEY_DISP_LACEDCOL_WELD_SIZE: "5mm",
             KEY_SEC_FU: fu,
             KEY_SEC_FY: fy,
             KEY_SEC_MATERIAL: design_dictionary.get(
                KEY_MATERIAL, 'Select Material')
        }.get(key, '')

        return val

    def get_lacing_profiles(self, *args):
        """
        Returns lacing profile options based on selected
            lacing pattern.
```

```
        """
        if not args or not args[0]:
            return connectdb('Angles', call_type="popup")

        pattern = args[0]
        if pattern == "Angle":
            return connectdb('Angles', call_type="popup")
        elif pattern == "Channel":
            return connectdb('Channels', call_type="popup")
        elif pattern == "Flat":
            return connectdb('Channels', call_type="popup")
        else:
            return []

    ####################################
    # Design Preference Functions End
    ####################################
    def module_name(self):
        return KEY_DISP_COMPRESSION_COLUMN

    def set_osdaglogger(self, widget_or_key=None):
        import logging
        self.logger = logging.getLogger('Osdag')
        self.logger.setLevel(logging.DEBUG)
        handler = logging.StreamHandler()
        formatter = logging.Formatter(fmt='%(asctime)s - %(
            name)s - %(levelname)s - %(message)s', datefmt='%Y
            -%m-%d %H:%M:%S')
        handler.setFormatter(formatter)
        self.logger.addHandler(handler)
        handler = logging.FileHandler('logging_text.log')
        handler.setFormatter(formatter)
        self.logger.addHandler(handler)
```

```
        # Always use OurLog for colored log messages in the
            log window
        if widget_or_key is not None:
            from ...Common import OurLog
            handler = OurLog(widget_or_key)
            handler.setFormatter(formatter)
            self.logger.addHandler(handler)

    def customized_input(self, *args, **kwargs):
        c_lst = []
        # Section Designation ComboBox with All/Customized
        t1 = (KEY_SECSIZE, self.fn_section_designation)
        c_lst.append(t1)
        return c_lst

    def fn_section_designation(self, *args):
        # This function is called when the Section Size
            ComboBox (All/Customized) changes
        # args[0] is the selected profile, args[1] is 'All'
            or 'Customized'
        if len(args) == 1 and isinstance(args[0], list):
            args = args[0]
        profile = args[0] if len(args) > 0 else None
        mode = args[1] if len(args) > 1 else 'All'

        if mode == 'All':
            # Return all designations from DB for the
                selected profile
            result = self.fn_profile_section(profile)
            if not isinstance(result, list):
                result = [result] if result else []
            # self.logger.info(f"Section designation (All)
                for profile '{profile}': {result}")
            return result
```

```
        elif mode == 'Customized':
            # Open popup dialog for user to select
            # Fetch all designations for the selected profile
                from the database
            section_list = self.fn_profile_section(profile)
            if not section_list:
                return []

            if dialog.exec_() == QDialog.Accepted:
                selected = dialog.get_selected()
                if not isinstance(selected, list):
                    selected = [selected] if selected else []
                # self.logger.info(f"Section designation (
                   Customized) selected: {selected}")
                return selected
            else:
                # self.logger.info("Section designation
                   dialog cancelled; returning previous
                   selection or empty list.")
                return current_selected if current_selected
                   else []
        else:
            return []

    def open_section_designation_dialog(self,
       selected_profile, current_selected=None,
       disabled_values=None):
        if disabled_values is None:
            disabled_values = []
        section_list = connectdb(selected_profile, call_type=
           "popup")
        # Prevent multiple dialogs
        if hasattr(self, 'section_designation_dialog') and
           self.section_designation_dialog is not None:
```

```
            if self.section_designation_dialog.isVisible():
                self.section_designation_dialog.raise_()
                self.section_designation_dialog.
                    activateWindow()
                return None
        self.section_designation_dialog =
            SectionDesignationDialog(section_list)
        self.dialogs.append(self.section_designation_dialog)
        if current_selected:
            self.section_designation_dialog.list_widget.
                clearSelection()
            for i in range(self.section_designation_dialog.
                list_widget.count()):
                if self.section_designation_dialog.
                    list_widget.item(i).text() in
                    current_selected:
                    self.section_designation_dialog.
                        list_widget.item(i).setSelected(True)
        result = None
        if self.section_designation_dialog.exec_() == QDialog
            .Accepted:
            selected = self.section_designation_dialog.
                get_selected()
            self.sec_list = selected
            result = selected
        # Remove from dialog tracking
        if self.section_designation_dialog in self.dialogs:
            self.dialogs.remove(self.
                section_designation_dialog)
        self.section_designation_dialog = None
        return result

    def input_values(self, *args, **kwargs):
        """
```

```
        Function declared in ui_template.py line 566
        Function to return a list of tuples to be displayed
           as the UI (Input Dock)
        """
        self.module = KEY_DISP_LACEDCOL
        options_list = []

        # Module title and name
        options_list.append((KEY_DISP_LACEDCOL, "Laced Column
           ", TYPE_MODULE, [], True, 'No Validator'))

        # Section
        options_list.append(("title_Section ", "Section
           Details", TYPE_TITLE, None, True, 'No Validator'))
        # Add section profile selection like in flexure.py
        options_list.append((KEY_SEC_PROFILE,
           KEY_DISP_SEC_PROFILE, TYPE_COMBOBOX,
           KEY_LACEDCOL_SEC_PROFILE_OPTIONS, True, 'No
           Validator'))
        # Section Designation ComboBox with All/Customized
        options_list.append((KEY_SECSIZE, KEY_DISP_SECSIZE,
           TYPE_COMBOBOX_CUSTOMIZED, ['All','Customized'],
           True, 'No Validator'))

        # Material
        options_list.append(("title_Material", "Material
           Properties", TYPE_TITLE, None, True, 'No Validator
           '))
        options_list.append((KEY_MATERIAL, KEY_DISP_MATERIAL,
            TYPE_COMBOBOX, VALUES_MATERIAL, True, 'No
           Validator'))

        # Geometry
```

```
        options_list.append(("title_Geometry", "Geometry",
            TYPE_TITLE, None, True, 'No Validator'))
        options_list.append((KEY_UNSUPPORTED_LEN_YY,
            KEY_DISP_UNSUPPORTED_LEN_YY, TYPE_TEXTBOX, None,
            True, 'Float Validator'))
        options_list.append((KEY_UNSUPPORTED_LEN_ZZ,
            KEY_DISP_UNSUPPORTED_LEN_ZZ, TYPE_TEXTBOX, None,
            True, 'Float Validator'))
        options_list.append((KEY_END1, KEY_DISP_END1,
            TYPE_COMBOBOX_CUSTOMIZED, VALUES_END1, True, 'No
            Validator'))
        options_list.append((KEY_END2,KEY_DISP_END2,
            TYPE_COMBOBOX_CUSTOMIZED, VALUES_END2, True, 'No
            Validator'))
        # Lacing
        options_list.append((KEY_LACING_PATTERN, "Lacing
            Pattern", TYPE_COMBOBOX, VALUES_LACING_PATTERN,
            True, 'No Validator'))
        # Connection
        options_list.append((KEY_CONN_TYPE, "Type of
            Connection", TYPE_COMBOBOX, VALUES_CONNECTION_TYPE
            , True, 'No Validator'))
        # Load
        options_list.append(("title_Load", "Load Details",
            TYPE_TITLE, None, True, 'No Validator'))
        options_list.append((KEY_AXIAL, "Axial Load (kN)",
            TYPE_TEXTBOX, None, True, 'Float Validator'))
        return options_list

    def fn_profile_section(self, *args):
        # Accepts either a list or *args
        if len(args) == 1 and isinstance(args[0], list):
            args = args[0]
        profile = args[0] if args else None
```

```
        # Handle laced column specific profiles
        if profile == '2-channels Back-to-Back':
            return connectdb("Channels", call_type="popup")
        elif profile == '2-channels Toe-to-Toe':
            return connectdb("Channels", call_type="popup")
        elif profile == '2-Girders':
            # For girders, we can use either Beams or Columns
                depending on the application
            res1 = connectdb("Beams", call_type="popup")
            res2 = connectdb("Columns", call_type="popup")
            return list(set(res1 + res2))
        # Handle standard profiles for backward compatibility
        elif profile == 'Beams':
            return connectdb("Beams", call_type="popup")
        elif profile == 'Columns':
            return connectdb("Columns", call_type="popup")
        elif profile == 'Beams and Columns':
            res1 = connectdb("Beams", call_type="popup")
            res2 = connectdb("Columns", call_type="popup")
            return list(set(res1 + res2))
        elif profile == 'RHS and SHS':
            res1 = connectdb("RHS", call_type="popup")
            res2 = connectdb("SHS", call_type="popup")
            return list(set(res1 + res2))
        elif profile == 'CHS':
            return connectdb("CHS", call_type="popup")
        elif profile in ['Angles', 'Back to Back Angles', '
           Star Angles']:
            return connectdb('Angles', call_type="popup")
        elif profile in ['Channels', 'Back to Back Channels'
           ]:
            return connectdb("Channels", call_type="popup")
        else:
```

```
            # Default fallback - return empty list
            return []

    def fn_end1_end2(self, *args):
        if len(args) == 1 and isinstance(args[0], list):
            args = args[0]
        end1 = args[0] if args else None

        if end1 == 'Fixed':
            return VALUES_END2
        elif end1 == 'Free':
            return ['Fixed']
        elif end1 == 'Hinged':
            return ['Fixed', 'Hinged', 'Roller']
        elif end1 == 'Roller':
            return ['Fixed', 'Hinged']

    def fn_end1_image(self, *args):
        if len(args) == 1 and isinstance(args[0], list):
            args = args[0]
        val = args[0] if args else None
        if val == 'Fixed':
            return str(files("osdag.data.ResourceFiles.images
                ").joinpath("6.RRRR.PNG"))
        elif val == 'Free':
            return str(files("osdag.data.ResourceFiles.images
                ").joinpath("1.RRFF.PNG"))
        elif val == 'Hinged':
            return str(files("osdag.data.ResourceFiles.images
                ").joinpath("5.RRRF.PNG"))
        elif val == 'Roller':
            return str(files("osdag.data.ResourceFiles.images
                ").joinpath("4.RRFR.PNG"))
```

```
    def fn_end2_image(self, *args):
        if len(args) == 1 and isinstance(args[0], list):
            args = args[0]
        end1 = args[0] if args else None
        end2 = args[1] if len(args) > 1 else None
        print("end 1 and end 2 are {}".format((end1, end2)))
        if end1 == 'Fixed':
            if end2 == 'Fixed':
                return str(files("osdag.data.ResourceFiles.
                    images").joinpath("6.RRRR.PNG"))
            elif end2 == 'Free':
                return str(files("osdag.data.ResourceFiles.
                    images").joinpath("1.RRFF_rotated.PNG"))
            elif end2 == 'Hinged':
                return str(files("osdag.data.ResourceFiles.
                    images").joinpath("5.RRRF_rotated.PNG"))
            elif end2 == 'Roller':
                return str(files("osdag.data.ResourceFiles.
                    images").joinpath("4.RRFR_rotated.PNG"))
        elif end1 == 'Free':
            return str(files("osdag.data.ResourceFiles.images
                ").joinpath("1.RRFF.PNG"))
        elif end1 == 'Hinged':
            if end2 == 'Fixed':
                return str(files("osdag.data.ResourceFiles.
                    images").joinpath("5.RRRF.PNG"))
            elif end2 == 'Hinged':
                return str(files("osdag.data.ResourceFiles.
                    images").joinpath("3.RFRF.PNG"))
            elif end2 == 'Roller':
                return str(files("osdag.data.ResourceFiles.
                    images").joinpath("2.FRFR_rotated.PNG"))
        elif end1 == 'Roller':
```

```
            if end2 == 'Fixed':
                return str(files("osdag.data.ResourceFiles.
                    images").joinpath("4.RRFR.PNG"))
            elif end2 == 'Hinged':
                return str(files("osdag.data.ResourceFiles.
                    images").joinpath("2.FRFR.PNG"))

    def input_value_changed(self, *args, **kwargs):
        lst = []
        # Section profile changes should update section
            designation (like in flexure.py)
        t1 = ([KEY_SEC_PROFILE], KEY_SECSIZE,
            TYPE_COMBOBOX_CUSTOMIZED, self.
            fn_section_designation)
        lst.append(t1)
        t2 = ([KEY_LYY], KEY_END_COND_YY,
            TYPE_COMBOBOX_CUSTOMIZED, self.get_end_conditions)
        lst.append(t2)
        t3 = ([KEY_LZZ], KEY_END_COND_ZZ,
            TYPE_COMBOBOX_CUSTOMIZED, self.get_end_conditions)
        lst.append(t3)
        t4 = ([KEY_MATERIAL], KEY_MATERIAL,
            TYPE_CUSTOM_MATERIAL, self.new_material)
        lst.append(t4)
        t5 = ([KEY_END1, KEY_END2], KEY_IMAGE, TYPE_IMAGE,
            self.fn_end2_image)
        lst.append(t5)
        t6 = ([KEY_END1_Y], KEY_END2_Y, TYPE_COMBOBOX, self.
            fn_end1_end2)
        lst.append(t6)
        t7 = ([KEY_END1_Y, KEY_END2_Y], KEY_IMAGE_Y,
            TYPE_IMAGE, self.fn_end2_image)
        lst.append(t7)
```

```
        # t8 = (KEY_END2, KEY_IMAGE, TYPE_IMAGE, self.
            fn_end2_image)
        # lst.append(t8)
        return lst

    def output_values(self, flag):
        # --- DEBUG: Print effective_length_yy and key
            results at the start of output_values ---
        print("[DEBUG][output_values] self.
            effective_length_yy:", getattr(self, '
            effective_length_yy', None))
        print("[DEBUG][output_values] self.result_IF_yy:",
            getattr(self, 'result_IF_yy', None))
        print("[DEBUG][output_values] self.result_ebs_yy:",
            getattr(self, 'result_ebs_yy', None))
        if not hasattr(self, 'effective_length_yy') or self.
            effective_length_yy is None:
            print("[WARNING][output_values] self.
                effective_length_yy is missing or None at
                output dock refresh!")
        def get_numeric(val):
            try:
                if val is None or val == '' or val in ['a', '
                    A']:
                    return None
                return float(val)
            except Exception:
                return None
        """
        Output the actual calculated values for effective
            length (YY) and slenderness ratio (YY), not
            classification text.
        """
        out_list = []
```

```
        def safe_display(val):
            try:
                if val is None:
                    return ''
                if isinstance(val, float):
                    return f"{val:.2f}"
                return str(val)
            except Exception:
                return str(val)

        # --- Always show Effective Lengths (YY/ZZ) at the
           top of output dock ---
        eff_len_yy = ''
        eff_len_zz = ''
        out_list.append((None, "Effective Length", TYPE_TITLE
           , None, True))
        # Forcefully show debug/calculation info for
           effective_length_yy, even if not used in output
        debug_yy_val = None
        if hasattr(self, 'effective_length_yy'):
            try:
                debug_yy_val = float(self.effective_length_yy
                   )
            except (TypeError, ValueError):
                debug_yy_val = None
        if debug_yy_val is not None:
            debug_yy = f"[DEBUG] Calculated
               effective_length_yy: {debug_yy_val}"
            out_list.append((None, debug_yy, TYPE_TITLE, None
               , True))
        if hasattr(self, 'effective_length_zz') and self.
           effective_length_zz is not None:
            try:
                vnum = float(self.effective_length_zz)
```

```
                debug_zz = f"[DEBUG] Calculated
                    effective_length_zz: {vnum}"
                out_list.append((None, debug_zz, TYPE_TITLE,
                    None, True))
            except (TypeError, ValueError):
                pass
        if hasattr(self, 'result') and isinstance(self.result
            , dict):
            # Only use numeric values for effective length,
                skip any string like 'mpc400', 'mpc', etc.
            for k in ['effective_length_yy', '
                Effective_length_yy', 'Effective Length YY']:
                v = self.result.get(k, None)
                try:
                    vnum = float(v)
                    eff_len_yy = safe_display(vnum)
                    break
                except (TypeError, ValueError):
                    continue
            for k in ['effective_length_zz', '
                Effective_length_zz', 'Effective Length ZZ']:
                v = self.result.get(k, None)
                try:
                    vnum = float(v)
                    eff_len_zz = safe_display(vnum)
                    break
                except (TypeError, ValueError):
                    continue
        # Fallback to attribute if not found in result, only
            if numeric
        if not eff_len_yy and hasattr(self, '
            effective_length_yy'):
            try:
                vnum = float(self.effective_length_yy)
```

```
                eff_len_yy = safe_display(vnum)
            except (TypeError, ValueError):
                pass
        if not eff_len_zz and hasattr(self, '
            effective_length_zz'):
            try:
                vnum = float(self.effective_length_zz)
                eff_len_zz = safe_display(vnum)
            except (TypeError, ValueError):
                pass
        # Fallback to optimum_section_ur_results if still not
            found, only if numeric
        if (not eff_len_yy or not eff_len_zz) and hasattr(
            self, 'optimum_section_ur_results') and self.
            optimum_section_ur_results:
            best_ur = min(self.optimum_section_ur_results.
                keys())
            best_result = self.optimum_section_ur_results[
                best_ur]
            if not eff_len_yy:
                for k in ['Effective_length_yy', 'Effective
                    Length YY', 'effective_length_yy']:
                    v = best_result.get(k, None)
                    try:
                        vnum = float(v)
                        eff_len_yy = safe_display(vnum)
                        break
                    except (TypeError, ValueError):
                        continue
            if not eff_len_zz:
                for k in ['Effective_length_zz', 'Effective
                    Length ZZ', 'effective_length_zz']:
                    v = best_result.get(k, None)
                    try:
```

```
                    vnum = float(v)
                    eff_len_zz = safe_display(vnum)
                    break
                except (TypeError, ValueError):
                    continue
        # --- Final fallback: if eff_len_yy is still blank,
           but self.effective_length_yy is set and numeric,
           use it ---
        if (not eff_len_yy or eff_len_yy == '') and hasattr(
           self, 'effective_length_yy'):
            try:
                vnum = float(self.effective_length_yy)
                eff_len_yy = safe_display(vnum)
            except (TypeError, ValueError):
                pass
        if (not eff_len_zz or eff_len_zz == '') and hasattr(
           self, 'effective_length_zz'):
            try:
                vnum = float(self.effective_length_zz)
                eff_len_zz = safe_display(vnum)
            except (TypeError, ValueError):
                pass
        # --- FORCE: Always show self.effective_length_yy as
           string in output dock for testing ---
        eff_len_yy_forced = ''
        if hasattr(self, 'effective_length_yy') and self.
           effective_length_yy is not None:
            try:
                eff_len_yy_forced = str(float(self.
                   effective_length_yy))
            except Exception:
                eff_len_yy_forced = str(self.
                   effective_length_yy)
```

```
        out_list.append((KEY_EFF_LEN_YY, "Effective Length (
            YY)", TYPE_TEXTBOX, eff_len_yy_forced, True))
        out_list.append((KEY_EFF_LEN_ZZ, "Effective Length (
            ZZ)", TYPE_TEXTBOX, eff_len_zz, True))

        # Slenderness Ratios
        out_list.append((None, "Slenderness Ratios",
            TYPE_TITLE, None, True))
        slender_yy = ''
        slender_zz = ''
        if flag:
            slender_yy_val = None
            slender_zz_val = None
            # 1. Try optimum_section_ur_results best result
                FIRST (most reliable)
            if hasattr(self, 'optimum_section_ur_results')
                and self.optimum_section_ur_results:
                best_ur = min(self.optimum_section_ur_results
                    .keys())
                best_result = self.optimum_section_ur_results
                    [best_ur]
                for k in ['Effective_sr_yy', 'Slenderness YY'
                    , 'effective_sr_yy', 'Slenderness_yy']:
                    if k in best_result:
                        try:
                            v = get_numeric(best_result[k])
                        except Exception:
                            v = None
                        if v is not None:
                            slender_yy_val = v
                            break
                for k in ['Effective_sr_zz', 'Slenderness ZZ'
                    , 'effective_sr_zz', 'Slenderness_zz']:
                    if k in best_result:
```

```
                    try:
                        v = get_numeric(best_result[k])
                    except Exception:
                        v = None
                    if v is not None:
                        slender_zz_val = v
                        break
        # 2. Fallback to direct attribute if not found above, with robust error handling
        if slender_yy_val is None and hasattr(self, 'effective_sr_yy'):
            try:
                slender_yy_val = get_numeric(self.effective_sr_yy)
            except Exception:
                slender_yy_val = None
        if slender_zz_val is None and hasattr(self, 'effective_sr_zz'):
            try:
                slender_zz_val = get_numeric(self.effective_sr_zz)
            except Exception:
                slender_zz_val = None
        slender_yy = safe_display(slender_yy_val) if slender_yy_val is not None else ''
        slender_zz = safe_display(slender_zz_val) if slender_zz_val is not None else ''
    out_list.append((KEY_SLENDER_YY, "Slenderness Ratio (YY)", TYPE_TEXTBOX, slender_yy, True))
    out_list.append((KEY_SLENDER_ZZ, "Slenderness Ratio (ZZ)", TYPE_TEXTBOX, slender_zz, True))

    # Design Values
```

```
        out_list.append((None, "Design Values", TYPE_TITLE,
            None, True))
        fcd = ''
        design_compressive = ''
        if flag:
            if hasattr(self, 'result_fcd') and self.
                result_fcd is not None:
                fcd = safe_display(self.result_fcd)
            elif hasattr(self, 'optimum_section_ur_results')
                and self.optimum_section_ur_results:
                best_ur = min(self.optimum_section_ur_results
                    .keys()) if self.
                    optimum_section_ur_results else None
                if best_ur:
                    fcd = safe_display(self.
                        optimum_section_ur_results[best_ur].
                        get('FCD', ''))
            if hasattr(self, 'result_capacity') and self.
                result_capacity is not None:
                design_compressive = safe_display(self.
                    result_capacity)
            elif hasattr(self, 'optimum_section_ur_results')
                and self.optimum_section_ur_results:
                best_ur = min(self.optimum_section_ur_results
                    .keys()) if self.
                    optimum_section_ur_results else None
                if best_ur:
                    design_compressive = safe_display(self.
                        optimum_section_ur_results[best_ur].
                        get('Capacity', ''))
        out_list.append((KEY_FCD, "Design Compressive Stress
            (fcd)", TYPE_TEXTBOX, fcd, True))
        out_list.append((KEY_DESIGN_COMPRESSIVE, "Design
            Compressive Strength", TYPE_TEXTBOX,
```

```
        design_compressive, True))

    # Utilization Ratio
    out_list.append((None, "Utilization Ratio",
        TYPE_TITLE, None, True))
    ur_value = ''
    if flag:
        if hasattr(self, 'result_UR') and self.result_UR
            is not None:
            ur_value = safe_display(self.result_UR)
        elif hasattr(self, 'optimum_section_ur_results')
            and self.optimum_section_ur_results:
            best_ur = min(self.optimum_section_ur_results
                .keys()) if self.
                optimum_section_ur_results else None
            if best_ur:
                ur_value = safe_display(best_ur)
    out_list.append(("utilization_ratio", "Utilization
        Ratio", TYPE_TEXTBOX, ur_value, True))

    # Section Classification (show as Plastic, Semi-
        Compact, etc.)
    out_list.append((None, "Section Classification",
        TYPE_TITLE, None, True))
    section_class = ''
    if flag:
        # Try self.result first
        if hasattr(self, 'result') and isinstance(self.
            result, dict):
            for k in ['section_class', 'Section class', '
                Section_class', 'sectionClass']:
                if k in self.result and self.result[k]
                    not in [None, '', 'a', 'A']:
                    section_class = str(self.result[k])
```

```
                    break
        # Fallback to attribute
        if not section_class and hasattr(self, '
            result_section_class') and self.
            result_section_class not in [None, '', 'a', 'A
            ']:
                section_class = str(self.result_section_class
                    )
        # Fallback to optimum_section_ur_results if still
            not found
        if not section_class and hasattr(self, '
            optimum_section_ur_results') and self.
            optimum_section_ur_results:
                best_ur = min(self.optimum_section_ur_results
                    .keys())
                best_result = self.optimum_section_ur_results
                    [best_ur]
                for k in ['Section class', 'section_class', '
                    Section_class', 'sectionClass']:
                    if k in best_result and best_result[k]
                        not in [None, '', 'a', 'A']:
                        section_class = str(best_result[k])
                        break
    out_list.append(("section_class", "Section Class",
        TYPE_TEXTBOX, section_class, True))

    # Effective Area
    out_list.append((None, "Effective Area", TYPE_TITLE,
        None, True))
    effective_area = ''
    if flag:
        if hasattr(self, 'result_effective_area') and
            self.result_effective_area is not None:
```

```
                effective_area = safe_display(self.
                    result_effective_area)
            elif hasattr(self, 'optimum_section_ur_results')
                and self.optimum_section_ur_results:
                best_ur = min(self.optimum_section_ur_results
                    .keys()) if self.
                    optimum_section_ur_results else None
                if best_ur:
                    effective_area = safe_display(self.
                        optimum_section_ur_results[best_ur].
                        get('Effective area', ''))
        out_list.append(("effective_area", "Effective Area (
            mm )", TYPE_TEXTBOX, effective_area, True))

        # Buckling Curve Classification
        out_list.append((None, "Buckling Curve Classification
            ", TYPE_TITLE, None, True))
        bc_yy = ''
        bc_zz = ''
        if flag:
            # --- UI Patch: Always show the latest value from
                self.result for output dock fields ---
            if hasattr(self, 'result') and isinstance(self.
                result, dict):
                for k in ['buckling_curve_yy', '
                    Buckling_curve_yy', 'Buckling Curve YY']:
                    if k in self.result and self.result[k]
                        not in [None, '', 'a', 'A']:
                        bc_yy = str(self.result[k])
                        break
                for k in ['buckling_curve_zz', '
                    Buckling_curve_zz', 'Buckling Curve ZZ']:
                    if k in self.result and self.result[k]
                        not in [None, '', 'a', 'A']:
```

```
                    bc_zz = str(self.result[k])
                    break
        # Fallback to attribute if not found in result
        if not bc_yy and hasattr(self, 'result_bc_yy')
           and self.result_bc_yy not in [None, '', 'a', '
           A']:
            bc_yy = str(self.result_bc_yy)
        if not bc_zz and hasattr(self, 'result_bc_zz')
           and self.result_bc_zz not in [None, '', 'a', '
           A']:
            bc_zz = str(self.result_bc_zz)
        # Fallback to optimum_section_ur_results if still
            not found
        if (not bc_yy or not bc_zz) and hasattr(self, '
           optimum_section_ur_results') and self.
           optimum_section_ur_results:
            best_ur = min(self.optimum_section_ur_results
               .keys())
            best_result = self.optimum_section_ur_results
               [best_ur]
            if not bc_yy:
                for k in ['Buckling_curve_yy', 'Buckling
                   Curve YY', 'buckling_curve_yy']:
                    if k in best_result and best_result[k
                       ] not in [None, '', 'a', 'A']:
                        bc_yy = str(best_result[k])
                        break
            if not bc_zz:
                for k in ['Buckling_curve_zz', 'Buckling
                   Curve ZZ', 'buckling_curve_zz']:
                    if k in best_result and best_result[k
                       ] not in [None, '', 'a', 'A']:
                        bc_zz = str(best_result[k])
                        break
```

```
            # If bc_zz is still 'A' (default), set to blank
            if bc_zz == 'A':
                bc_zz = ''
            out_list.append(("buckling_curve_yy", "Buckling
               Curve (YY)", TYPE_TEXTBOX, bc_yy, True))
            out_list.append(("buckling_curve_zz", "Buckling
               Curve (ZZ)", TYPE_TEXTBOX, bc_zz, True))

        # Imperfection Factor
        out_list.append((None, "Imperfection Factor",
           TYPE_TITLE, None, True))
        if_yy = ''
        if_zz = ''
        def is_numeric(val):
            try:
                float(val)
                return True
            except Exception:
                return False
        if flag:
            if hasattr(self, 'result_IF_yy') and self.
               result_IF_yy is not None and is_numeric(self.
               result_IF_yy):
                if_yy = safe_display(self.result_IF_yy)
            elif hasattr(self, 'optimum_section_ur_results')
               and self.optimum_section_ur_results:
                best_ur = min(self.optimum_section_ur_results
                   .keys()) if self.
                   optimum_section_ur_results else None
                if best_ur:
                    val = self.optimum_section_ur_results[
                       best_ur].get('IF_yy', '')
                    if is_numeric(val):
                        if_yy = safe_display(val)
```

```
            # FINAL FALLBACK: use self.result dict if still
                blank
            if (not if_yy or if_yy == '') and hasattr(self, '
                result') and isinstance(self.result, dict):
                val = self.result.get('imperfection_factor_yy
                    ', '')
                if is_numeric(val):
                    if_yy = safe_display(val)
            if hasattr(self, 'result_IF_zz') and self.
                result_IF_zz is not None and is_numeric(self.
                result_IF_zz):
                if_zz = safe_display(self.result_IF_zz)
            elif hasattr(self, 'optimum_section_ur_results')
                and self.optimum_section_ur_results:
                best_ur = min(self.optimum_section_ur_results
                    .keys()) if self.
                    optimum_section_ur_results else None
                if best_ur:
                    val = self.optimum_section_ur_results[
                        best_ur].get('IF_zz', '')
                    if is_numeric(val):
                        if_zz = safe_display(val)
            if (not if_zz or if_zz == '') and hasattr(self, '
                result') and isinstance(self.result, dict):
                val = self.result.get('imperfection_factor_zz
                    ', '')
                if is_numeric(val):
                    if_zz = safe_display(val)
        out_list.append(("imperfection_factor_yy", "
            Imperfection Factor (YY)", TYPE_TEXTBOX, if_yy,
            True))
        out_list.append(("imperfection_factor_zz", "
            Imperfection Factor (ZZ)", TYPE_TEXTBOX, if_zz,
            True))
```

```
        # Euler Buckling Stress
        out_list.append((None, "Euler Buckling Stress",
            TYPE_TITLE, None, True))
        ebs_yy = ''
        ebs_zz = ''
        if flag:
            if hasattr(self, 'result_ebs_yy') and self.
                result_ebs_yy is not None and is_numeric(self.
                result_ebs_yy):
                ebs_yy = safe_display(self.result_ebs_yy)
            elif hasattr(self, 'optimum_section_ur_results')
                and self.optimum_section_ur_results:
                best_ur = min(self.optimum_section_ur_results
                    .keys()) if self.
                    optimum_section_ur_results else None
                if best_ur:
                    val = self.optimum_section_ur_results[
                        best_ur].get('EBS_yy', '')
                    if is_numeric(val):
                        ebs_yy = safe_display(val)
            if (not ebs_yy or ebs_yy == '') and hasattr(self,
                'result') and isinstance(self.result, dict):
                val = self.result.get('
                    euler_buckling_stress_yy', '')
                if is_numeric(val):
                    ebs_yy = safe_display(val)
            if hasattr(self, 'result_ebs_zz') and self.
                result_ebs_zz is not None and is_numeric(self.
                result_ebs_zz):
                ebs_zz = safe_display(self.result_ebs_zz)
            elif hasattr(self, 'optimum_section_ur_results')
                and self.optimum_section_ur_results:
```

```
                best_ur = min(self.optimum_section_ur_results
                    .keys()) if self.
                    optimum_section_ur_results else None
                if best_ur:
                    val = self.optimum_section_ur_results[
                        best_ur].get('EBS_zz', '')
                    if is_numeric(val):
                        ebs_zz = safe_display(val)
            if (not ebs_zz or ebs_zz == '') and hasattr(self,
                'result') and isinstance(self.result, dict):
                val = self.result.get('
                    euler_buckling_stress_zz', '')
                if is_numeric(val):
                    ebs_zz = safe_display(val)
        out_list.append(("euler_buckling_stress_yy", "Euler
            Buckling Stress (YY)", TYPE_TEXTBOX, ebs_yy, True)
            )
        out_list.append(("euler_buckling_stress_zz", "Euler
            Buckling Stress (ZZ)", TYPE_TEXTBOX, ebs_zz, True)
            )

        # Non-dimensional Effective Slenderness Ratio
        out_list.append((None, "Non-dimensional Effective
            Slenderness Ratio", TYPE_TITLE, None, True))
        nd_esr_yy = ''
        nd_esr_zz = ''
        if flag:
            # --- UI Patch: Always show the latest value from
                self.result for output dock fields ---
            if hasattr(self, 'result') and isinstance(self.
                result, dict):
                for k in ['nd_esr_yy', 'ND_ESR_yy', 'ND ESR
                    YY']:
```

```
                if k in self.result and self.result[k]
                    not in [None, '', 'a', 'A']:
                    nd_esr_yy = safe_display(self.result[
                        k])
                    break
            for k in ['nd_esr_zz', 'ND_ESR_zz', 'ND ESR
                ZZ']:
                if k in self.result and self.result[k]
                    not in [None, '', 'a', 'A']:
                    nd_esr_zz = safe_display(self.result[
                        k])
                    break
        # Fallback to attribute if not found in result
        if not nd_esr_yy and hasattr(self, '
            result_nd_esr_yy') and self.result_nd_esr_yy
            not in [None, '', 'a', 'A']:
            nd_esr_yy = safe_display(self.
                result_nd_esr_yy)
        if not nd_esr_zz and hasattr(self, '
            result_nd_esr_zz') and self.result_nd_esr_zz
            not in [None, '', 'a', 'A']:
            nd_esr_zz = safe_display(self.
                result_nd_esr_zz)
        # Fallback to optimum_section_ur_results if still
            not found
        if (not nd_esr_yy or not nd_esr_zz) and hasattr(
            self, 'optimum_section_ur_results') and self.
            optimum_section_ur_results:
            best_ur = min(self.optimum_section_ur_results
                .keys())
            best_result = self.optimum_section_ur_results
                [best_ur]
            if not nd_esr_yy:
```

```
                for k in ['ND_ESR_yy', 'nd_esr_yy', 'ND
                    ESR YY']:
                    if k in best_result and best_result[k
                        ] not in [None, '', 'a', 'A']:
                        nd_esr_yy = safe_display(
                            best_result[k])
                        break
            if not nd_esr_zz:
                for k in ['ND_ESR_zz', 'nd_esr_zz', 'ND
                    ESR ZZ']:
                    if k in best_result and best_result[k
                        ] not in [None, '', 'a', 'A']:
                        nd_esr_zz = safe_display(
                            best_result[k])
                        break
        out_list.append(("nd_esr_yy", "ND ESR (YY)",
            TYPE_TEXTBOX, nd_esr_yy, True))
        out_list.append(("nd_esr_zz", "ND ESR (ZZ)",
            TYPE_TEXTBOX, nd_esr_zz, True))

    # Phi Values
    out_list.append((None, "Phi Values", TYPE_TITLE, None
        , True))
    phi_yy = ''
    phi_zz = ''
    if flag:
        if hasattr(self, 'result_phi_yy') and self.
            result_phi_yy is not None:
            phi_yy = safe_display(self.result_phi_yy)
        elif hasattr(self, 'optimum_section_ur_results')
            and self.optimum_section_ur_results:
            best_ur = min(self.optimum_section_ur_results
                .keys()) if self.
                optimum_section_ur_results else None
```

```
                if best_ur:
                    phi_yy = safe_display(self.
                        optimum_section_ur_results[best_ur].
                        get('phi_yy', ''))
            if hasattr(self, 'result_phi_zz') and self.
                result_phi_zz is not None:
                phi_zz = safe_display(self.result_phi_zz)
            elif hasattr(self, 'optimum_section_ur_results')
                and self.optimum_section_ur_results:
                best_ur = min(self.optimum_section_ur_results
                    .keys()) if self.
                    optimum_section_ur_results else None
                if best_ur:
                    phi_zz = safe_display(self.
                        optimum_section_ur_results[best_ur].
                        get('phi_zz', ''))
        out_list.append(("phi_yy", "Phi (YY)", TYPE_TEXTBOX,
            phi_yy, True))
        out_list.append(("phi_zz", "Phi (ZZ)", TYPE_TEXTBOX,
            phi_zz, True))

        # Stress Reduction Factor
        out_list.append((None, "Stress Reduction Factor",
            TYPE_TITLE, None, True))
        srf_yy = ''
        srf_zz = ''
        if flag:
            if hasattr(self, 'result_srf_yy') and self.
                result_srf_yy is not None:
                srf_yy = safe_display(self.result_srf_yy)
            elif hasattr(self, 'optimum_section_ur_results')
                and self.optimum_section_ur_results:
                best_ur = min(self.optimum_section_ur_results
                    .keys()) if self.
```

```
                optimum_section_ur_results else None
            if best_ur:
                srf_yy = safe_display(self.
                    optimum_section_ur_results[best_ur].
                    get('SRF_yy', ''))
        if hasattr(self, 'result_srf_zz') and self.
            result_srf_zz is not None:
            srf_zz = safe_display(self.result_srf_zz)
        elif hasattr(self, 'optimum_section_ur_results')
            and self.optimum_section_ur_results:
            best_ur = min(self.optimum_section_ur_results
                .keys()) if self.
                optimum_section_ur_results else None
            if best_ur:
                srf_zz = safe_display(self.
                    optimum_section_ur_results[best_ur].
                    get('SRF_zz', ''))
    out_list.append(("stress_reduction_factor_yy", "SRF (
        YY)", TYPE_TEXTBOX, srf_yy, True))
    out_list.append(("stress_reduction_factor_zz", "SRF (
        ZZ)", TYPE_TEXTBOX, srf_zz, True))

    # Design Compressive Stress Values
    out_list.append((None, "Design Compressive Stress
        Values", TYPE_TITLE, None, True))
    fcd_1_yy = ''
    fcd_1_zz = ''
    fcd_2 = ''
    if flag:
        if hasattr(self, 'result_fcd_1_yy') and self.
            result_fcd_1_yy is not None:
            fcd_1_yy = safe_display(self.result_fcd_1_yy)
        elif hasattr(self, 'optimum_section_ur_results')
            and self.optimum_section_ur_results:
```

```
                best_ur = min(self.optimum_section_ur_results
                    .keys()) if self.
                    optimum_section_ur_results else None
                if best_ur:
                    fcd_1_yy = safe_display(self.
                        optimum_section_ur_results[best_ur].
                        get('FCD_1_yy', ''))
            if hasattr(self, 'result_fcd_1_zz') and self.
                result_fcd_1_zz is not None:
                fcd_1_zz = safe_display(self.result_fcd_1_zz)
            elif hasattr(self, 'optimum_section_ur_results')
                and self.optimum_section_ur_results:
                best_ur = min(self.optimum_section_ur_results
                    .keys()) if self.
                    optimum_section_ur_results else None
                if best_ur:
                    fcd_1_zz = safe_display(self.
                        optimum_section_ur_results[best_ur].
                        get('FCD_1_zz', ''))
            if hasattr(self, 'result_fcd_2') and self.
                result_fcd_2 is not None:
                fcd_2 = safe_display(self.result_fcd_2)
            elif hasattr(self, 'optimum_section_ur_results')
                and self.optimum_section_ur_results:
                best_ur = min(self.optimum_section_ur_results
                    .keys()) if self.
                    optimum_section_ur_results else None
                if best_ur:
                    fcd_2 = safe_display(self.
                        optimum_section_ur_results[best_ur].
                        get('FCD_2', ''))
        out_list.append(("fcd_1_yy", "FCD_1 (YY)",
            TYPE_TEXTBOX, fcd_1_yy, True))
```

```
        out_list.append(("fcd_1_zz", "FCD_1 (ZZ)",
            TYPE_TEXTBOX, fcd_1_zz, True))
        out_list.append(("fcd_2", "FCD_2", TYPE_TEXTBOX,
            fcd_2, True))

        # --- Channel and Lacing Details Section ---
        out_list.append((None, "Channel and Lacing Details",
            TYPE_TITLE, None, True))
        spacing_channels = ''
        if flag:
            if hasattr(self, 'result') and self.result.get('
                channel_spacing') is not None:
                spacing_channels = safe_display(self.result.
                    get('channel_spacing'))
            elif hasattr(self, 'optimum_section_ur_results')
                and self.optimum_section_ur_results:
                best_ur = min(self.optimum_section_ur_results
                    .keys()) if self.
                    optimum_section_ur_results else None
                if best_ur:
                    spacing_channels = safe_display(self.
                        optimum_section_ur_results[best_ur].
                        get('channel_spacing', ''))
        out_list.append(("channel_spacing", "Spacing Between
            Channels (mm)", TYPE_TEXTBOX, spacing_channels,
            True))

        # --- Tie Plate Section ---
        out_list.append((None, "Tie Plate", TYPE_TITLE, None,
            True))
        tie_plate_d = ''
        tie_plate_t = ''
        tie_plate_l = ''
        if flag:
```

```
            if hasattr(self, 'result') and self.result.get('
               tie_plate_d') is not None:
                tie_plate_d = safe_display(self.result.get('
                   tie_plate_d'))
            elif hasattr(self, 'optimum_section_ur_results')
               and self.optimum_section_ur_results:
                best_ur = min(self.optimum_section_ur_results
                   .keys()) if self.
                   optimum_section_ur_results else None
                if best_ur:
                    tie_plate_d = safe_display(self.
                       optimum_section_ur_results[best_ur].
                       get('tie_plate_d', ''))
            if hasattr(self, 'result') and self.result.get('
               tie_plate_t') is not None:
                tie_plate_t = safe_display(self.result.get('
                   tie_plate_t'))
            elif hasattr(self, 'optimum_section_ur_results')
               and self.optimum_section_ur_results:
                best_ur = min(self.optimum_section_ur_results
                   .keys()) if self.
                   optimum_section_ur_results else None
                if best_ur:
                    tie_plate_t = safe_display(self.
                       optimum_section_ur_results[best_ur].
                       get('tie_plate_t', ''))
            if hasattr(self, 'result') and self.result.get('
               tie_plate_l') is not None:
                tie_plate_l = safe_display(self.result.get('
                   tie_plate_l'))
            elif hasattr(self, 'optimum_section_ur_results')
               and self.optimum_section_ur_results:
                best_ur = min(self.optimum_section_ur_results
                   .keys()) if self.
```

```
                optimum_section_ur_results else None
            if best_ur:
                tie_plate_l = safe_display(self.
                    optimum_section_ur_results[best_ur].
                    get('tie_plate_l', ''))
        out_list.append(("tie_plate_d", "Tie Plate Depth D (
            mm)", TYPE_TEXTBOX, tie_plate_d, True))
        out_list.append(("tie_plate_t", "Tie Plate Thickness
            t (mm)", TYPE_TEXTBOX, tie_plate_t, True))
        out_list.append(("tie_plate_l", "Tie Plate Length L (
            mm)", TYPE_TEXTBOX, tie_plate_l, True))

        # --- Lacing Spacing Section ---
        out_list.append((None, "Lacing Spacing", TYPE_TITLE,
            None, True))
        lacing_spacing = ''
        if flag:
            if hasattr(self, 'result') and self.result.get('
                lacing_spacing') is not None:
                lacing_spacing = safe_display(self.result.get
                    ('lacing_spacing'))
            elif hasattr(self, 'optimum_section_ur_results')
                and self.optimum_section_ur_results:
                best_ur = min(self.optimum_section_ur_results
                    .keys()) if self.
                    optimum_section_ur_results else None
                if best_ur:
                    lacing_spacing = safe_display(self.
                        optimum_section_ur_results[best_ur].
                        get('lacing_spacing', ''))
        out_list.append(("lacing_spacing", "Lacing Spacing (
            L0) (mm)", TYPE_TEXTBOX, lacing_spacing, True))

        return out_list
```

```
    def set_input_values(self, design_dictionary):
        # self.logger.info(f"set_input_values called with: {
            design_dictionary}")
        super(Member, self).set_input_values(
            design_dictionary)
        # section properties
        self.module = design_dictionary.get(KEY_DISP_LACEDCOL
            , "")
        self.mainmodule = 'Columns with known support
            conditions'
        self.sec_profile = design_dictionary.get(
            KEY_LACEDCOL_SEC_PROFILE, "")
        self.sec_list = design_dictionary.get(KEY_SECSIZE,
            [])
        # Coerce sec_list to a list if it's a string
        if isinstance(self.sec_list, str):
            if self.sec_list and self.sec_list != 'Select
                Section':
                self.sec_list = [self.sec_list]
            else:
                self.sec_list = []
        elif not isinstance(self.sec_list, list):
            self.sec_list = list(self.sec_list) if self.
                sec_list else []
        self.material = design_dictionary.get(
            KEY_SEC_MATERIAL, "")
        # Defensive checks for required fields
        def is_valid_material(mat):
            if isinstance(mat, list):
                return any(m and m != 'Select Material' for m
                    in mat)
            return mat and mat != 'Select Material'
        if not is_valid_material(self.material):
```

```
            self.logger.error("Material is missing or invalid
                .")
            self.design_status = False
            return
        def is_valid_section(sec):
            if isinstance(sec, list):
                return any(s and s != 'Select Section' for s
                    in sec)
            return sec and sec != 'Select Section'
        if not is_valid_section(self.sec_list):
            self.logger.error(f"Section list is missing or
                invalid: {self.sec_list}")
            self.design_status = False
            return
        # section user data
        try:
            self.length_zz = float(design_dictionary.get(
                KEY_UNSUPPORTED_LEN_ZZ, 0))
            if self.length_zz <= 0:
                raise ValueError
        except:
            self.logger.error("Actual Length (z-z), mm is
                missing or invalid.")
            self.design_status = False
            return
        try:
            self.length_yy = float(design_dictionary.get(
                KEY_UNSUPPORTED_LEN_YY, 0))
            if self.length_yy <= 0:
                raise ValueError
        except:
            self.logger.error("Actual Length (y-y), mm is
                missing or invalid.")
            self.design_status = False
```

```
            return
        # end condition
        self.end_1_z = design_dictionary.get(KEY_END1, "")
        self.end_2_z = design_dictionary.get(KEY_END2, "")
        self.end_1_y = design_dictionary.get(KEY_END1_Y, "")
        self.end_2_y = design_dictionary.get(KEY_END2_Y, "")
        # factored loads
        try:
            axial_force = float(design_dictionary.get(
               KEY_AXIAL, 0))
            if axial_force <= 0:
                raise ValueError
        except:
            self.logger.error("Axial Load (kN) is missing or
               invalid.")
            self.design_status = False
            return
        self.load = Load(axial_force=axial_force, shear_force
           =0.0, moment=0.0, moment_minor=0.0, unit_kNm=True)
        # design preferences
        try:
            self.allowable_utilization_ratio = float(
               design_dictionary.get(KEY_ALLOW_UR, 1.0))
        except:
            self.allowable_utilization_ratio = 1.0
        try:
            self.effective_area_factor = float(
               design_dictionary.get(KEY_EFFECTIVE_AREA_PARA,
                1.0))
        except:
            self.effective_area_factor = 1.0
        try:
            self.optimization_parameter = design_dictionary[
               KEY_OPTIMIZATION_PARA]
```

```
        except:
            self.optimization_parameter = 'Utilization Ratio'
        try:
            self.steel_cost_per_kg = float(design_dictionary[
                KEY_STEEL_COST])
        except:
            self.steel_cost_per_kg = 50
        self.allowed_sections = ['Plastic', 'Compact', 'Semi-
            Compact', 'Slender']

        # Defensive: Only run if section list and material
            are valid
        if self.sec_list and self.material:
            # Clear material cache when material changes to
                ensure fresh properties
            self.material_lookup_cache = {}

            # Initialize material_property BEFORE
                section_classification
            self.material_property = Material(material_grade=
                self.material, thickness=0)
            self.flag = self.section_classification()
            if self.flag:
                self.design_column()
                self.results()

        # safety factors
        self.gamma_m0 = IS800_2007.cl_5_4_1_Table_5["gamma_m0
            "]["yielding"]

        # initialize the design status
        self.design_status_list = []
        self.design_status = False
        self.failed_design_dict = {}
```

```
        # Always perform calculations if required fields are
            present

    # Simulation starts here
    def section_classification(self):
        # Defensive: ensure material_property is set
        if not hasattr(self, 'material_property') or self.
            material_property is None:
            if hasattr(self, 'material') and self.material:
                try:
                    self.material_property = Material(
                        material_grade=self.material,
                        thickness=0)
                except Exception as e:
                    self.logger.error(f"Failed to initialize
                        material_property: {e}")
                    self.design_status = False
                    return False
            else:
                self.logger.error("Material property not
                    initialized and material is missing.")
                self.design_status = False
                return False
        # Deduplicate section list to avoid repeated
            processing
        self.sec_list = list(dict.fromkeys(self.sec_list))
        local_flag = True
        self.input_section_list = []
        self.input_section_classification = {}

        slender_sections = []
        accepted_sections = []
        rejected_sections = []  # Track all rejected sections
            with reasons
```

```
        for section in self.sec_list:
            trial_section = section.strip("'")

            # Always define flange_ratio and web_ratio with
               safe defaults
            flange_ratio = None
            web_ratio = None

            # fetching the section properties
            if self.sec_profile ==
               KEY_LACEDCOL_SEC_PROFILE_OPTIONS[0]:  # Beams
               and columns
                try:
                    result = Beam(designation=trial_section,
                       material_grade=self.material)
                except:
                    result = Column(designation=trial_section
                       , material_grade=self.material)
                self.section_property = result
            elif self.sec_profile ==
               KEY_LACEDCOL_SEC_PROFILE_OPTIONS[1]:  # RHS
               and SHS
                try:
                    result = RHS(designation=trial_section,
                       material_grade=self.material)
                except:
                    result = SHS(designation=trial_section,
                       material_grade=self.material)
                self.section_property = result
            elif self.sec_profile ==
               KEY_LACEDCOL_SEC_PROFILE_OPTIONS[2] and
               isinstance(self.section_property, CHS):  # CHS
                self.section_property = CHS(designation=
                   trial_section, material_grade=self.
```

```
                material)
        else:
            self.section_property = Column(designation=
                trial_section, material_grade=self.
                material)

        # updating the material property based on
            thickness of the thickest element
        # Defensive checks and logging
        if not self.material or self.material in [None, '
            ', 'Select Material']:
            error_msg = f"Material is missing or invalid
                before database lookup: {self.material}"
            self.logger.error(error_msg)
            self.failed_reason = error_msg
            self.design_status = False
            rejected_sections.append((trial_section, '
                Material properties not found'))
            continue

        flange_thk = getattr(self.section_property, '
            flange_thickness', None)
        web_thk = getattr(self.section_property, '
            web_thickness', None)

        if flange_thk is None or web_thk is None:
            error_msg = f"Section property thickness
                missing for {trial_section}:
                flange_thickness={flange_thk},
                web_thickness={web_thk}"
            self.logger.error(error_msg)
            self.failed_reason = error_msg
            self.design_status = False
```

```
                rejected_sections.append((trial_section, '
                    Section property thickness missing'))
                continue

            try:
                max_thk = max(float(flange_thk), float(
                    web_thk))
            except Exception as e:
                error_msg = f"Invalid thickness values for {
                    trial_section}: flange_thickness={
                    flange_thk}, web_thickness={web_thk},
                    error={e}"
                self.logger.error(error_msg)
                self.failed_reason = error_msg
                self.design_status = False
                rejected_sections.append((trial_section, '
                    Invalid thickness values'))
                continue

            cache_key = (self.material, round(max_thk, 1))
            if cache_key not in self.material_lookup_cache:
                self.material_property.
                    connect_to_database_to_get_fy_fu(self.
                    material, max_thk)
                self.material_lookup_cache[cache_key] = (self
                    .material_property.fy, self.
                    material_property.fu)
                # self.logger.info(f"Updated material
                    properties for {self.material}: fy={self.
                    material_property.fy}, fu={self.
                    material_property.fu}")
            else:
                self.material_property.fy, self.
                    material_property.fu = self.
```

```
                material_lookup_cache[cache_key]
            # self.logger.info(f"Using cached material
                properties for {self.material}: fy={self.
                material_property.fy}, fu={self.
                material_property.fu}")

        # Defensive: Check if material properties were
            found
        if not self.material_property.fy or not self.
            material_property.fu:
            from ...Common import PATH_TO_DATABASE
            error_msg = f"Material properties not found
                for grade '{self.material}' and thickness
                '{max_thk}'. Check if the material exists
                in the database at {PATH_TO_DATABASE}."
            self.logger.error(error_msg)
            self.failed_reason = error_msg
            self.design_status = False
            rejected_sections.append((trial_section, '
                Material properties not found'))
            continue

        # section classification
        if self.sec_profile ==
            KEY_LACEDCOL_SEC_PROFILE_OPTIONS[0]:  # Beams
            and Columns
            if self.section_property.type == 'Rolled':
                self.flange_class = IS800_2007.Table2_i((
                    self.section_property.flange_width /
                    2), self.section_property.
                    flange_thickness,
                                                    self
                                                    .
                                                    material_
```

```
                                                    .
                                                    fy
                                                    ,

                                                    self
                                                    .
                                                    section_p
                                                    .
                                                    type
                                                    )
                                                    [0]

            else:
                self.flange_class = IS800_2007.Table2_i
                    (((self.section_property.flange_width
                    / 2) - (self.section_property.
                    web_thickness / 2)),
                                                  self
                                                    .
                                                    section_p
                                                    .
                                                    flange_th
                                                    ,

                                                    self
                                                    .
                                                    section_p
                                                    .
                                                    fy
                                                    ,

                                                  self
                                                    .
                                                    section_p
```

```
                                                    .
                                                    type
                                                    )
                                                    [0]

            # FIX: Use 'Neutral axis at mid-depth' for
               web_class
            self.web_class = IS800_2007.Table2_iii((self.
               section_property.depth - (2 * self.
               section_property.flange_thickness)),
                                                self.
                                                   section_proper
                                                   .
                                                   web_thickness
                                                   ,
                                                   self
                                                   .
                                                   material_prope
                                                   .fy
                                                   ,
                                                classification_ty
                                                   ='
                                                   Axial

                                                   compression
                                                   ')

            # Calculate ratios for I-sections
            web_ratio = (self.section_property.depth - 2
               * (
                        self.section_property.
                           flange_thickness + self.
                           section_property.root_radius))
                            / self.section_property.
```

```
                                web_thickness
                    flange_ratio = self.section_property.
                        flange_width / 2 / self.section_property.
                        flange_thickness

                elif self.sec_profile ==
                    KEY_LACEDCOL_SEC_PROFILE_OPTIONS[1]:  # RHS
                    and SHS
                    self.flange_class = IS800_2007.Table2_iii((
                        self.section_property.depth - (2 * self.
                        section_property.flange_thickness)),
                                                                self
                                                                    .
                                                                    section_pro
                                                                    .
                                                                    flange_thic
                                                                    ,

                                                                    self
                                                                    .
                                                                    material_pr
                                                                    .
                                                                    fy
                                                                    ,

                                                                classification
                                                                    =
                                                                    '
                                                                    Axial

                                                                    compression
                                                                    '
                                                                    )
```

```
                self.web_class = self.flange_class

                # Calculate ratios for RHS/SHS
                web_ratio = (self.section_property.depth - 2
                    * (
                            self.section_property.
                                flange_thickness + self.
                                section_property.root_radius))
                                 / self.section_property.
                                web_thickness
                flange_ratio = self.section_property.
                    flange_width / 2 / self.section_property.
                    flange_thickness

            elif self.sec_profile ==
                KEY_LACEDCOL_SEC_PROFILE_OPTIONS[2] and
                isinstance(self.section_property, CHS):  # CHS
                 self.flange_class = IS800_2007.Table2_x(self.
                    section_property.out_diameter, self.
                    section_property.flange_thickness,
                                                        self.
                                                            material_prop
                                                            .
                                                            fy
                                                            ,
                                                            load_type
                                                            ='
                                                            axial

                                                            compression
                                                            ')
                self.web_class = self.flange_class
                # For CHS, use diameter to thickness ratio
```

```
            web_ratio = self.section_property.
                out_diameter / self.section_property.
                flange_thickness
            flange_ratio = web_ratio  # Same ratio for
                circular sections
        else:
            self.flange_class = self.web_class = None
            web_ratio = flange_ratio = None

        # Smart classification logic
        if self.flange_class == 'Slender' and self.
            web_class == 'Slender':
            self.section_class = 'Slender'
        elif 'Slender' in [self.flange_class, self.
            web_class]:
            self.section_class = 'Semi-Compact'  #
                downgrade if only one is slender
        else:
            if self.flange_class == 'Plastic' and self.
                web_class == 'Plastic':
                self.section_class = 'Plastic'
            elif 'Plastic' in [self.flange_class, self.
                web_class] or 'Compact' in [self.
                flange_class, self.web_class]:
                self.section_class = 'Compact'
            else:
                self.section_class = 'Semi-Compact'

        # Optionally, upgrade borderline slender sections
        if self.section_class == 'Slender':
            if (flange_ratio is not None and web_ratio is
                not None and
                isinstance(flange_ratio, (int, float))
                    and isinstance(web_ratio, (int, float)
```

```
                    ) and
                flange_ratio <= 9.5 and web_ratio <=
                    79.5):
                self.logger.info(f"Reclassifying
                    borderline Slender section {
                    trial_section} to Semi-Compact")
                self.section_class = 'Semi-Compact'
        self.effective_length_zz = IS800_2007.
            cl_7_2_2_effective_length_of_prismatic_compression_member
            (
             self.length_zz,
             end_1=self.end_1_z,
             end_2=self.end_2_z)
        self.effective_length_yy = IS800_2007.
            cl_7_2_2_effective_length_of_prismatic_compression_member
            (
             self.length_yy,
             end_1=self.end_1_y,
             end_2=self.end_2_y)
        print(f"[DEBUG] Calculated effective_length_yy: {
            self.effective_length_yy}")

        # 2.3 - Effective slenderness ratio
        self.effective_sr_zz = self.effective_length_zz /
             self.section_property.rad_of_gy_z
        self.effective_sr_yy = self.effective_length_yy /
             self.section_property.rad_of_gy_y

        limit = IS800_2007.cl_3_8_max_slenderness_ratio
            (1)
        if self.effective_sr_zz > limit and self.
            effective_sr_yy > limit:
             error_msg = f"Length provided is beyond the
                 limit allowed for section {trial_section}.
```

```
                    [Reference: Cl 3.8, IS 800:2007]"
                self.logger.warning(error_msg)
                self.logger.error("Cannot compute. Given
                    Length does not pass.")
                self.failed_reason = error_msg
                slender_sections.append(trial_section)
                rejected_sections.append((trial_section, '
                    Slenderness ratio exceeded'))
                continue

            # Add section to input list if it passes
                classification filter
            if self.section_class in self.allowed_sections:
                self.input_section_list.append(trial_section)
                self.input_section_classification.update({
                    trial_section: [self.section_class, self.
                    flange_class, self.web_class, flange_ratio
                    , web_ratio]})
                accepted_sections.append(trial_section)
            else:
                self.logger.info(f"Section {trial_section}
                    classified as '{self.section_class}' but
                    not in allowed sections: {self.
                    allowed_sections}")
                rejected_sections.append((trial_section, f"
                    Classified as '{self.section_class}', not
                    in allowed_sections"))
        # Check if any sections passed the classification
            filter
        if not self.input_section_list:
            error_msg = f"No sections passed the
                classification filter. Allowed sections: {self
                .allowed_sections}"
            self.logger.error(error_msg)
```

```
            self.failed_reason = error_msg
            self.design_status = False
            # Robust fallback: expand allowed_sections to all
                types and re-check
            self.logger.warning("No section passed
               classification. Expanding allowed_sections to
               include all types.")
            self.allowed_sections = ['Plastic', 'Compact', '
               Semi-Compact', 'Slender']
            for section in self.sec_list:
                trial_section = section.strip("'")
                if trial_section in self.
                   input_section_classification:
                    if self.input_section_classification[
                       trial_section][0] in self.
                       allowed_sections:
                        self.input_section_list.append(
                           trial_section)
                        accepted_sections.append(
                           trial_section)
            if not self.input_section_list:
                self.logger.error("Design Failed. All
                   sections are too slender or do not meet
                   requirements.")
                local_flag = False

        return local_flag


    def design_column(self):
        try:
            summary_lines = []
            # checking DP inputs
```

```
            if (self.allowable_utilization_ratio <= 0.10) or
               (self.allowable_utilization_ratio > 1.0):
                logger.warning("The defined value of
                   Utilization Ratio in the design
                   preferences tab is out of the suggested
                   range.")
                logger.info("Provide an appropriate input and
                    re-design.")
                logger.info("Assuming a default value of 1.0.
                   ")
                self.allowable_utilization_ratio = 1.0
                self.design_status = True
                self.design_status_list.append(self.
                   design_status)

            if (self.effective_area_factor <= 0.10) or (self.
               effective_area_factor > 1.0):
                logger.warning("The defined value of
                   Effective Area Factor in the design
                   preferences tab is out of the suggested
                   range.")
                logger.info("Provide an appropriate input and
                    re-design.")
                logger.info("Assuming a default value of 1.0.
                   ")
                self.effective_area_factor = 1.0
                self.design_status = False
                self.design_status_list.append(self.
                   design_status)

            self.epsilon = math.sqrt(250 / self.
               material_property.fy)
            self.optimum_section_ur_results = {}
            self.optimum_section_ur = []
```

```
            self.flag = self.section_classification()
            # Remove duplicate sections to avoid repeated
               calculations
            self.input_section_list = list(dict.fromkeys(self
               .input_section_list))
            if self.flag:
                for section in self.input_section_list:
                    ur_class = None
                    if section in self.
                       input_section_classification:
                        ur_class = self.
                           input_section_classification[
                           section]
                    ur_value = None
                    for ur in self.optimum_section_ur_results
                       :
                        if self.optimum_section_ur_results[ur
                           ].get('Designation') == section:
                            ur_value = ur
                            break
                for section in self.input_section_list:  #
                   iterating the design over each section to
                   find the most optimum section

                    # fetching the section properties of the
                       selected section
                    if self.sec_profile == VALUES_SEC_PROFILE
                       [0]:  # Beams and columns
                        try:
                            result = Beam(designation=section
                               , material_grade=self.material
                               )
                        except:
```

```
                    result = Column(designation=
                        section, material_grade=self.
                        material)
                self.section_property = result
            elif self.sec_profile ==
                VALUES_SEC_PROFILE[1]:  # RHS and SHS
                try:
                    result = RHS(designation=section,
                        material_grade=self.material)
                except:
                    result = SHS(designation=section,
                        material_grade=self.material)
                self.section_property = result

            elif self.sec_profile ==
                VALUES_SEC_PROFILE[2]:  # CHS
                self.section_property = CHS(
                    designation=section,
                    material_grade=self.material)
                self.section_property.designation =
                    section
            else:   #Why?
                self.section_property = Column(
                    designation=section,
                    material_grade=self.material)

            self.material_property.
                connect_to_database_to_get_fy_fu(self.
                material, max(self.section_property.
                flange_thickness,
```

```
                self.epsilon = math.sqrt(250 / self.
                    material_property.fy)


                # initialize lists for updating the
                    results dictionary
                self.list_zz = []
                self.list_yy = []

                self.list_zz.append(section)
                self.list_yy.append(section)

                # Step 1 - computing the effective
                    sectional area
                self.section_class = self.
                    input_section_classification[section
                    ][0]

                if self.section_class == 'Slender':
                    if (self.sec_profile ==
                        VALUES_SEC_PROFILE[0]):  # Beams
                        and Columns
                        self.effective_area = (2 * ((31.4
                            * self.epsilon * self.
                            section_property.
                            flange_thickness) *
                                                    self.
                                                        section_prope
                                                        .
                                                        flange_thickn
```

```
                                                    ))
                                                     +
                                                     \
                                            (2 * ((21 *
                                               self.
                                               epsilon *
                                               self.
                                               section_property
                                               .
                                               web_thickness
                                               ) * self.
                                               section_property
                                               .
                                               web_thickness
                                               ))
                        elif (self.sec_profile ==
                           VALUES_SEC_PROFILE[1]):
                            self.effective_area = (2 * 21 *
                               self.epsilon * self.
                               section_property.
                               flange_thickness) * 2
                    else:
                        self.effective_area = self.
                           section_property.area  # mm2


                    if self.effective_area_factor < 1.0:
                        self.effective_area = round(self.
                           effective_area * self.
                           effective_area_factor, 2)

                    self.list_zz.append(self.section_class)
                    self.list_yy.append(self.section_class)
```

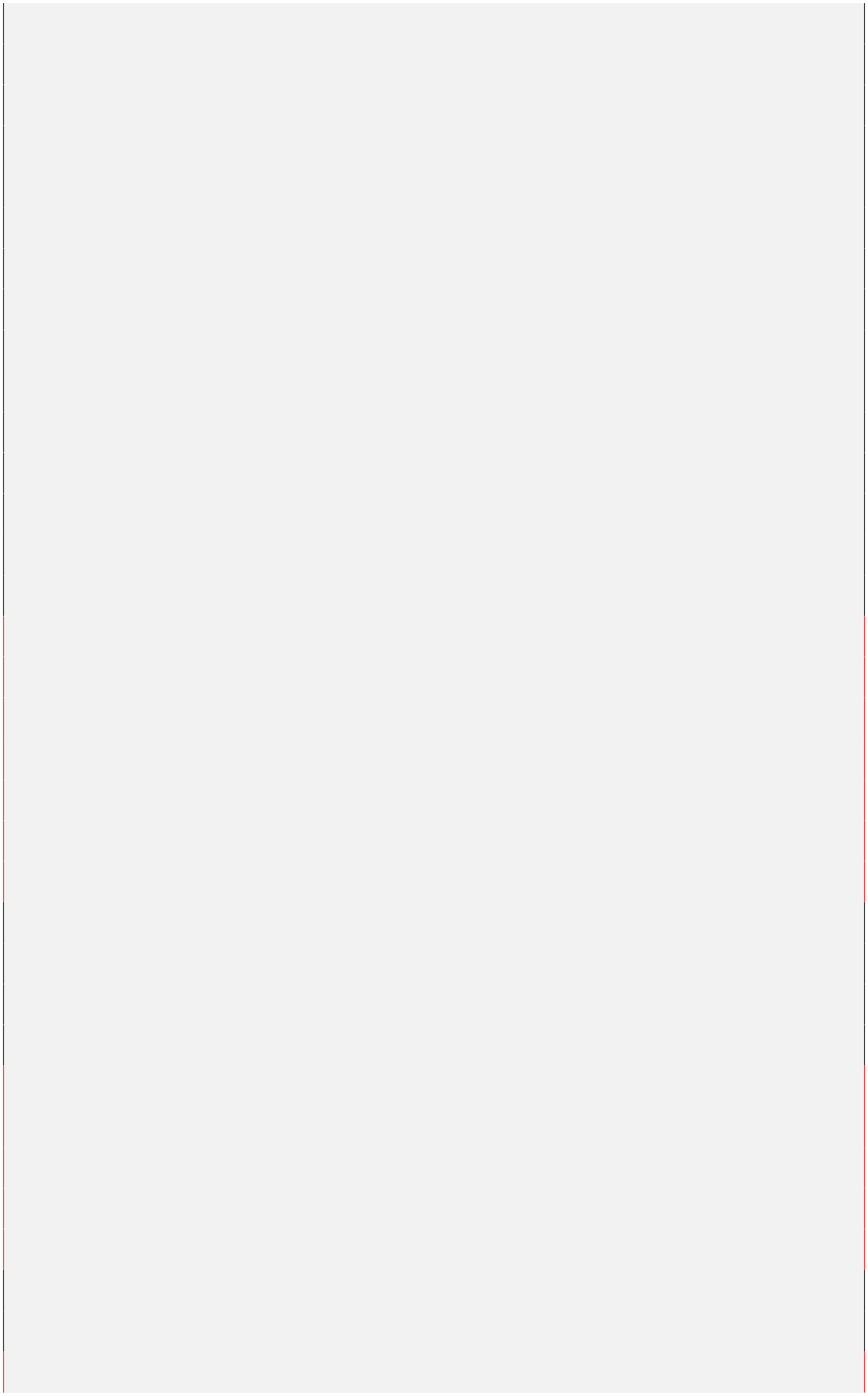
```
                self.list_zz.append(self.effective_area)
                self.list_yy.append(self.effective_area)

                # Step 2 - computing the design
                    compressive stress

                # 2.1 - Buckling curve classification and
                     Imperfection factor
                if (self.sec_profile ==
                    VALUES_SEC_PROFILE[0]):  # Beams and
                    Columns

                     if self.section_property.type == '
                        Rolled':
                         self.buckling_class_zz =
                            IS800_2007.
                            cl_7_1_2_2_buckling_class_of_crosssection
                            (self.section_property.
                            flange_width,
```

```
                    self.buckling_class_yy =
                        IS800_2007.
                        cl_7_1_2_2_buckling_class_of_crosssection
                        (self.section_property.
                        flange_width,
```



```
                else:
                    self.buckling_class_zz =
                        IS800_2007.
                        cl_7_1_2_2_buckling_class_of_crosssection
                        (self.section_property.
                        flange_width,
```

```
self.buckling_class_yy =
    IS800_2007.
    cl_7_1_2_2_buckling_class_of_crosssection
    (self.section_property.
    flange_width,
```

```
                else:
                    self.buckling_class_zz = 'a'
                    self.buckling_class_yy = 'a'
```

```
                self.imperfection_factor_zz = IS800_2007.
                    cl_7_1_2_1_imperfection_factor(
                    buckling_class=self.buckling_class_zz)
                self.imperfection_factor_yy = IS800_2007.
                    cl_7_1_2_1_imperfection_factor(
                    buckling_class=self.buckling_class_yy)

                self.list_zz.append(self.
                    buckling_class_zz)
                self.list_yy.append(self.
                    buckling_class_yy)

                self.list_zz.append(self.
                    imperfection_factor_zz)
                self.list_yy.append(self.
                    imperfection_factor_yy)

                # Store imperfection factors for output
                self.result_IF_zz = float(self.
                    imperfection_factor_zz) if self.
                    imperfection_factor_zz is not None
                    else None
                self.result_IF_yy = float(self.
                    imperfection_factor_yy) if self.
                    imperfection_factor_yy is not None
                    else None
                self.result['imperfection_factor_yy'] =
                    self.result_IF_yy
                self.result['imperfection_factor_zz'] =
                    self.result_IF_zz

                # 2.2 - Effective length
                self.effective_length_zz = IS800_2007.
                    cl_7_2_2_effective_length_of_prismatic_compressio
```

```
                    (self.length_zz ,
                                                                    )
                self.effective_length_yy = IS800_2007.
                    cl_7_2_2_effective_length_of_prismatic_compressio
                    (self.length_yy ,
                                                                    )
```

```
                self.list_zz.append(self.
                    effective_length_zz)
                self.list_yy.append(self.
                    effective_length_yy)

                # 2.3 - Effective slenderness ratio
                self.effective_sr_zz = self.
                    effective_length_zz / self.
                    section_property.rad_of_gy_z
                self.effective_sr_yy = self.
                    effective_length_yy / self.
                    section_property.rad_of_gy_y

                self.list_zz.append(self.effective_sr_zz)
                self.list_yy.append(self.effective_sr_yy)

                # 2.4 - Euler buckling stress
                self.euler_bs_zz = (math.pi ** 2 * self.
                    section_property.modulus_of_elasticity
                    ) / self.effective_sr_zz ** 2
                self.euler_bs_yy = (math.pi ** 2 * self.
                    section_property.modulus_of_elasticity
                    ) / self.effective_sr_yy ** 2

                self.list_zz.append(self.euler_bs_zz)
```

```
                self.list_yy.append(self.euler_bs_yy)

                # Store euler buckling stress for output
                self.result_ebs_zz = float(self.
                    euler_bs_zz) if self.euler_bs_zz is
                    not None else None
                self.result_ebs_yy = float(self.
                    euler_bs_yy) if self.euler_bs_yy is
                    not None else None
                self.result['euler_buckling_stress_yy'] =
                     self.result_ebs_yy
                self.result['euler_buckling_stress_zz'] =
                     self.result_ebs_zz

                # 2.5 - Non-dimensional effective
                    slenderness ratio
                self.non_dim_eff_sr_zz = math.sqrt(self.
                    material_property.fy / self.
                    euler_bs_zz)
                self.non_dim_eff_sr_yy = math.sqrt(self.
                    material_property.fy / self.
                    euler_bs_yy)


                self.list_zz.append(self.
                    non_dim_eff_sr_zz)
                self.list_yy.append(self.
                    non_dim_eff_sr_yy)

                # 2.5 - phi
                self.phi_zz = 0.5 * (1 + (self.
                    imperfection_factor_zz * (self.
                    non_dim_eff_sr_zz - 0.2)) + self.
                    non_dim_eff_sr_zz ** 2)
```

```
                self.phi_yy = 0.5 * (1 + (self.
                    imperfection_factor_yy * (self.
                    non_dim_eff_sr_yy - 0.2)) + self.
                    non_dim_eff_sr_yy ** 2)


                self.list_zz.append(self.phi_zz)
                self.list_yy.append(self.phi_yy)

                # 2.6 - Design compressive stress
                self.stress_reduction_factor_zz = 1 / (
                    self.phi_zz + (self.phi_zz ** 2 - self
                    .non_dim_eff_sr_zz ** 2) ** 0.5)
                self.stress_reduction_factor_yy = 1 / (
                    self.phi_yy + (self.phi_yy ** 2 - self
                    .non_dim_eff_sr_yy ** 2) ** 0.5)

                self.list_zz.append(self.
                    stress_reduction_factor_zz)
                self.list_yy.append(self.
                    stress_reduction_factor_yy)

                self.f_cd_1_zz = (self.
                    stress_reduction_factor_zz * self.
                    material_property.fy) / self.gamma_m0
                self.f_cd_1_yy = (self.
                    stress_reduction_factor_yy * self.
                    material_property.fy) / self.gamma_m0
                self.f_cd_2 = self.material_property.fy /
                     self.gamma_m0

                self.f_cd_zz = min(self.f_cd_1_zz, self.
                    f_cd_2)
```

```
                    self.f_cd_yy = min(self.f_cd_1_yy, self.
                        f_cd_2)

                    self.f_cd = min(self.f_cd_zz, self.
                        f_cd_yy)

                    self.list_zz.append(self.f_cd_1_zz)
                    self.list_yy.append(self.f_cd_1_yy)

                    self.list_zz.append(self.f_cd_2)
                    self.list_yy.append(self.f_cd_2)

                    self.list_zz.append(self.f_cd_zz)
                    self.list_yy.append(self.f_cd_yy)

                    self.list_zz.append(self.f_cd)
                    self.list_yy.append(self.f_cd)

                    # 2.7 - Capacity of the section

                    self.section_capacity = self.f_cd * self.
                        effective_area  # N

                    self.list_zz.append(self.section_capacity
                        )
                    self.list_yy.append(self.section_capacity
                        )

                    # 2.8 - UR
                    self.ur = round(self.load.axial_force /
                        self.section_capacity, 3)

                    self.list_zz.append(self.ur)
                    self.list_yy.append(self.ur)
```

```
                self.optimum_section_ur.append(self.ur)

                # --- Tie Plate, Spacing, and Lacing
                    Angle Calculations ---
                tie_plate_d = round(2 * self.
                    section_property.depth / 3, 2)
                            # mm
                tie_plate_t = round(self.section_property
                    .web_thickness, 2)          # mm
                tie_plate_l = round(self.section_property
                    .depth / 2, 2)              # mm
                spacing_between_channels = round(self.
                    section_property.depth + 2 *
                    tie_plate_t, 2)  # mm
                lacing_angle = round(math.degrees(math.
                    atan(spacing_between_channels / (2 *
                    tie_plate_l))), 2)  # degrees

                # Store in self.result for output dock (
                    for the last/selected section)
                self.result['tie_plate_d'] = tie_plate_d
                self.result['tie_plate_t'] = tie_plate_t
                self.result['tie_plate_l'] = tie_plate_l
                self.result['channel_spacing'] =
                    spacing_between_channels
                self.result['lacing_spacing'] =
                    lacing_angle

                # Store in optimum_section_ur_results for
                     output dock (for each section)
                ur = self.ur
                if ur in self.optimum_section_ur_results:
                     self.optimum_section_ur_results[ur]['
                        tie_plate_d'] = tie_plate_d
```

```
                    self.optimum_section_ur_results[ur]['
                        tie_plate_t'] = tie_plate_t
                    self.optimum_section_ur_results[ur]['
                        tie_plate_l'] = tie_plate_l
                    self.optimum_section_ur_results[ur]['
                        channel_spacing'] =
                        spacing_between_channels
                    self.optimum_section_ur_results[ur]['
                        lacing_spacing'] = lacing_angle

                # Calculate and store cost for this
                    section (needed for cost-based
                    optimization)
                self.cost = (self.section_property.
                    unit_mass * self.section_property.area
                     * 1e-4) * min(self.length_zz, self.
                    length_yy) * self.steel_cost_per_kg
                self.optimum_section_cost.append(self.
                    cost)

                #tieplate
                # 2.X - Tie Plate Dimensions
                self.tie_plate_d = round(2 * self.
                    section_property.depth / 3, 2)
                            # mm
                self.tie_plate_t = round(self.
                    section_property.web_thickness, 2)
                            # mm
                self.tie_plate_l = round(self.
                    section_property.depth / 2, 2)
                                # mm
                self.list_zz.append(self.tie_plate_d)
                self.list_yy.append(self.tie_plate_d)
                self.list_zz.append(self.tie_plate_t)
```

```
                    self.list_yy.append(self.tie_plate_t)
                    self.list_zz.append(self.tie_plate_l)
                    self.list_yy.append(self.tie_plate_l)

# 2.X - Lacing Spacing Between Channels
                    self.spacing_between_channels = round(
                        self.section_property.depth + 2 * self
                        .tie_plate_t, 2)  # mm
                    self.list_zz.append(self.
                        spacing_between_channels)
                    self.list_yy.append(self.
                        spacing_between_channels)

# 2.X - Lacing Angle
                    self.lacing_angle = round(math.degrees(
                        math.atan(self.
                        spacing_between_channels / (2 * self.
                        tie_plate_l))), 2)  # degrees
                    self.list_zz.append(self.lacing_angle)
                    self.list_yy.append(self.lacing_angle)
                    self.store_additional_outputs(
                        d=self.tie_plate_d,
                        t=self.tie_plate_t,
                        l=self.tie_plate_l,
                        spacing=self.lacing_angle,
                        c_spacing=self.
                            spacing_between_channels
                        )


                    # Step 3 - Storing the optimum results to
                        a list in descending order
                    list_1 = [
```

```
                    'Designation', 'Section class', '
                        Effective area', '
                        Buckling_curve_zz', 'IF_zz', '
                        Effective_length_zz', '
                        Effective_SR_zz',
                    'EBS_zz', 'ND_ESR_zz', 'phi_zz', '
                        SRF_zz', 'FCD_1_zz', 'FCD_2', '
                        FCD_zz', 'FCD', 'Capacity', 'UR',
                        'Cost', 'Designation',
                    'Section class', 'Effective area', '
                        Buckling_curve_yy', 'IF_yy', '
                        Effective_length_yy', '
                        Effective_SR_yy', 'EBS_yy',
                    'ND_ESR_yy', 'phi_yy', 'SRF_yy', '
                        FCD_1_yy', 'FCD_2', 'FCD_yy', 'FCD
                        ', 'Capacity', 'UR', 'Cost'
                ]

        try:
            for section in self.input_section_list:
                # perform all calculations for this
                    section (assumed already done)
                section_result = {
                    'Designation': section,
                    'Section class': self.section_class,
                    'Effective area': self.effective_area
                        ,
                    'Buckling_curve_zz': self.
                        buckling_class_zz,
                    'IF_zz': self.result_IF_zz,
                    'Effective_length_zz': self.
                        effective_length_zz,
                    'Effective_SR_zz': self.
                        effective_sr_zz,
```

```
                'EBS_zz': self.result_ebs_zz,
                'ND_ESR_zz': self.non_dim_eff_sr_zz,
                'phi_zz': self.phi_zz,
                'SRF_zz': self.
                    stress_reduction_factor_zz,
                'FCD_1_zz': self.f_cd_1_zz,
                'FCD_2': self.f_cd_2,
                'FCD_zz': self.f_cd_zz,
                'FCD': self.f_cd,
                'Capacity': self.section_capacity,
                'UR': self.ur,
                'Cost': self.cost,
                'Buckling_curve_yy': self.
                    buckling_class_yy,
                'IF_yy': self.result_IF_yy,
                'Effective_length_yy': self.
                    effective_length_yy,
                'Effective_SR_yy': self.
                    effective_sr_yy,
                'EBS_yy': self.result_ebs_yy,
                'ND_ESR_yy': self.non_dim_eff_sr_yy,
                'phi_yy': self.phi_yy,
                'SRF_yy': self.
                    stress_reduction_factor_yy,
                'FCD_1_yy': self.f_cd_1_yy,
                'FCD_yy': self.f_cd_yy,
                'tie_plate_d': self.tie_plate_d,
                'tie_plate_t': self.tie_plate_t,
                'tie_plate_l': self.tie_plate_l,
                'channel_spacing': self.
                    spacing_between_channels,
                'lacing_spacing': self.lacing_angle,
            }
```

```
                    self.optimum_section_ur_results[self.ur]
                        = section_result

                # 2- Based on optimum cost
                self.optimum_section_cost_results[self.cost]
                    = {}
                list_2 = self.list_zz + self.list_yy
                for j in list_1:
                    for k in list_2:
                        self.optimum_section_cost_results[
                            self.cost][j] = k
                        list_2.pop(0)
                        break

            except Exception as e:
                self.logger.error(f"Exception in
                    design_column: {e}")
                import traceback
                self.logger.error(traceback.format_exc())
                self.design_status = False
                self.failed_design_dict = {}
                return

            best_ur = None
            best_section_results = None
            if self.optimum_section_ur:
                best_ur = round(min(self.optimum_section_ur,
                    key=lambda x: abs(x-1.0)), 3)
                best_section_results = self.
                    optimum_section_ur_results.get(best_ur)
                if best_section_results is None:
                    # fallback: find the closest key within a
                        small tolerance
                    if self.optimum_section_ur_results:
```

```
                    closest_key = min(self.
                        optimum_section_ur_results.keys(),
                         key=lambda k: abs(k - best_ur))
                    if abs(closest_key - best_ur) < 1e-3:
                        best_section_results = self.
                            optimum_section_ur_results[
                            closest_key]
                if best_section_results is None:
                    self.logger.error(f"Result UR {
                        best_ur} not found in
                        optimum_section_ur_results. No
                        valid design result to display.")
                    return
                summary_lines.append(f"Best UR: {best_ur}")
                summary_lines.append(f"Best Section Results:
                    {best_section_results}")

            summary_lines.append("======================")
            summary_text = "\n".join(summary_lines)

            self.design_summary = {
                'input_section_list': self.input_section_list
                    ,
                'optimum_section_ur': self.optimum_section_ur
                    ,
                'best_ur': best_ur,
                'best_section_results': best_section_results,
                'summary_text': summary_text
            }
        except Exception as e:
            self.logger.error(f"Exception in design_column: {
                e}")
            import traceback
            self.logger.error(traceback.format_exc())
```

```
            self.design_status = False
            self.failed_design_dict = {}
            return

    def store_additional_outputs(self, d=None, t=None, l=None
        , spacing=None, c_spacing=None, ur=None):
        """
        Store additional calculated outputs for tie plate,
            lacing, and channel spacing in self.result and, if
             ur is provided, in self.
            optimum_section_ur_results[ur].
        """
        if d is not None:
            self.result['tie_plate_d'] = d
        if t is not None:
            self.result['tie_plate_t'] = t
        if l is not None:
            self.result['tie_plate_l'] = l
        if spacing is not None:
            self.result['lacing_spacing'] = spacing
        if c_spacing is not None:
            self.result['channel_spacing'] = c_spacing
        # Also store in optimum_section_ur_results[ur] if ur
            is provided
        if ur is not None and hasattr(self, '
            optimum_section_ur_results') and ur in self.
            optimum_section_ur_results:
            if d is not None:
                self.optimum_section_ur_results[ur]['
                    tie_plate_d'] = d
            if t is not None:
                self.optimum_section_ur_results[ur]['
                    tie_plate_t'] = t
            if l is not None:
```

```
                self.optimum_section_ur_results[ur]['
                    tie_plate_l'] = l
            if spacing is not None:
                self.optimum_section_ur_results[ur]['
                    lacing_spacing'] = spacing
            if c_spacing is not None:
                self.optimum_section_ur_results[ur]['
                    channel_spacing'] = c_spacing

    def calculate(self, design_dictionary):
        self.reset_state_for_new_design()
        # --- PATCH: Ensure end condition values are always
            present and correct in design_dictionary ---
        # Try to extract from possible keys, fallback to UI
            attributes if missing, and update dictionary
        # This ensures end1/end2 are always correct for
            calculation and output
        # 1. Try to get from dictionary as before
        end1 = (
            design_dictionary.get('End Condition 1', None)
            or design_dictionary.get('KEY_END1', None)
            or design_dictionary.get('end1', None)
            or design_dictionary.get('End_1', None)
        )
        end2 = (
            design_dictionary.get('End Condition 2', None)
            or design_dictionary.get('KEY_END2', None)
            or design_dictionary.get('end2', None)
            or design_dictionary.get('End_2', None)
        )
        # 2. If still missing, try to get from UI combo boxes
            if available
        if (not end1 or isinstance(end1, list)) and hasattr(
            self, 'end1_combo'):
```

```
            try:
                end1_val = self.end1_combo.currentText() if
                    hasattr(self.end1_combo, 'currentText')
                    else None
                if end1_val and isinstance(end1_val, str):
                    end1 = end1_val
            except Exception:
                pass
        if (not end2 or isinstance(end2, list)) and hasattr(
            self, 'end2_combo'):
            try:
                end2_val = self.end2_combo.currentText() if
                    hasattr(self.end2_combo, 'currentText')
                    else None
                if end2_val and isinstance(end2_val, str):
                    end2 = end2_val
            except Exception:
                pass
        # 3. If still missing, try to get from UI line edits
            (if used)
        if (not end1 or isinstance(end1, list)) and hasattr(
            self, 'end1_lineedit'):
            try:
                end1_val = self.end1_lineedit.text() if
                    hasattr(self.end1_lineedit, 'text') else
                    None
                if end1_val and isinstance(end1_val, str):
                    end1 = end1_val
            except Exception:
                pass
        if (not end2 or isinstance(end2, list)) and hasattr(
            self, 'end2_lineedit'):
            try:
```

```
            end2_val = self.end2_lineedit.text() if
               hasattr(self.end2_lineedit, 'text') else
               None
            if end2_val and isinstance(end2_val, str):
                end2 = end2_val
        except Exception:
            pass
    # 4. If still missing, try to get from attributes set
       by UI (if any)
    if (not end1 or isinstance(end1, list)) and hasattr(
       self, 'end1'):
        if isinstance(self.end1, str):
            end1 = self.end1
    if (not end2 or isinstance(end2, list)) and hasattr(
       self, 'end2'):
        if isinstance(self.end2, str):
            end2 = self.end2
    # 5. Update the dictionary so the rest of the method
       works as expected
    if end1 and isinstance(end1, str):
        design_dictionary['End Condition 1'] = end1
        design_dictionary['End_1'] = end1
    if end2 and isinstance(end2, str):
        design_dictionary['End Condition 2'] = end2
        design_dictionary['End_2'] = end2
    for k in design_dictionary:
        print(f"    {k!r}: {design_dictionary[k]!r}")

    # --- Patch: Calculate and assign effective length YY
       using IS 800:2007 Table 11 logic ---
    try:
        # --- Patch: Forcefully push calculated values
           for output dock ---
        unsupported_length_yy = (
```

```
            design_dictionary.get('Unsupported Length YY'
                , None)
            or design_dictionary.get('
                KEY_UNSUPPORTED_LEN_YY', None)
            or design_dictionary.get('
                unsupported_length_yy', None)
            or design_dictionary.get('Unsupported.
                Length_yy', None)
        )
        # Warn if end1 or end2 is a list, not a string
        if isinstance(end1, list):
            end1 = None
        if isinstance(end2, list):
            end2 = None
        if not end1:
            print("[DEBUG][calculate] MISSING: end1 is
                not set or empty!")
            pass
        if not end2:
            print("[DEBUG][calculate] MISSING: end2 is
                not set or empty!")
            pass
        if not unsupported_length_yy:
            pass
            try:
                unsupported_length_yy = float(
                    unsupported_length_yy)
                eff_len_yy_calc = self.
                    calculate_effective_length_yy(end1,
                    end2, unsupported_length_yy)
                self.effective_length_yy =
                    eff_len_yy_calc
                if not hasattr(self, 'result') or not
                    isinstance(self.result, dict):
```

```
                    self.result = {}
                self.result['effective_length_yy'] =
                    eff_len_yy_calc
                self.result['Effective_length_yy'] =
                    eff_len_yy_calc
                self.result['Effective Length YY'] =
                    eff_len_yy_calc
                print(f"[DEBUG] Calculated
                    effective_length_yy: {eff_len_yy_calc}
                    ")
            except Exception as e:
                print('[DEBUG] Could not calculate
                    effective_length_yy:', e)
            pass
        # --- Buckling Curve ZZ ---
        bc_zz_val = None
        if hasattr(self, 'result_bc_zz') and self.
            result_bc_zz:
            bc_zz_val = self.result_bc_zz
        elif hasattr(self, 'optimum_section_ur_results')
            and self.optimum_section_ur_results:
            best_ur = min(self.optimum_section_ur_results
                .keys())
            best_result = self.optimum_section_ur_results
                [best_ur]
            for k in ['Buckling_curve_zz', 'Buckling
                Curve ZZ', 'buckling_curve_zz']:
                if k in best_result:
                    bc_zz_val = best_result[k]
                    break
        if bc_zz_val is not None:
            self.result['buckling_curve_zz'] = bc_zz_val
            self.result['Buckling_curve_zz'] = bc_zz_val
            self.result['Buckling Curve ZZ'] = bc_zz_val
```

```
                self.result_bc_zz = bc_zz_val
            # --- ND ESR YY ---
            nd_esr_yy_val = None
            if hasattr(self, 'result_nd_esr_yy') and self.
               result_nd_esr_yy is not None:
                nd_esr_yy_val = self.result_nd_esr_yy
            elif hasattr(self, 'optimum_section_ur_results')
               and self.optimum_section_ur_results:
                best_ur = min(self.optimum_section_ur_results
                   .keys())
                best_result = self.optimum_section_ur_results
                   [best_ur]
                for k in ['ND_ESR_yy', 'nd_esr_yy', 'ND ESR
                   YY']:
                    if k in best_result:
                        nd_esr_yy_val = best_result[k]
                        break
            if nd_esr_yy_val is not None:
                self.result['nd_esr_yy'] = nd_esr_yy_val
                self.result['ND_ESR_yy'] = nd_esr_yy_val
                self.result['ND ESR YY'] = nd_esr_yy_val
                self.result_nd_esr_yy = nd_esr_yy_val
        except Exception as e:
            print('[DEBUG] Error in effective_length_yy/
               buckling_curve_zz/nd_esr_yy patch:', e)
        """
        Perform all calculations for the laced column based
           on user input and assign results to output fields.
        This method is called explicitly from the UI after
           user input is collected.
        """
        try:
            # Set input values and run validation/
               classification
```

```
            self.set_input_values(design_dictionary)
            self.section_classification()
            # Print all section results for debug/
               verification
            self.print_all_section_results()
            # For the best section (lowest UR), extract and
               assign all output fields
            if hasattr(self, 'optimum_section_ur_results')
               and self.optimum_section_ur_results:
                # Find the best section (lowest UR) with all
                   required results
                best_ur = None
                for ur in sorted(self.
                   optimum_section_ur_results.keys()):
                    res = self.optimum_section_ur_results[ur]
                    # Check if all required keys are present
                       and not None
                    required_keys = [
                        'Effective Length YY', 'Effective
                           Length ZZ', 'Slenderness YY', '
                           Slenderness ZZ', 'FCD', 'Capacity'
                           ,
                        'Section class', 'Effective area', '
                           Buckling_curve_yy', '
                           Buckling_curve_zz', 'IF_yy', '
                           IF_zz',
                        'EBS_yy', 'EBS_zz', 'ND_ESR_yy', '
                           ND_ESR_zz', 'phi_yy', 'phi_zz', '
                           SRF_yy', 'SRF_zz',
                        'FCD_1_yy', 'FCD_1_zz', 'FCD_2', '
                           Cost', 'channel_spacing', '
                           tie_plate_d', 'tie_plate_t', '
                           tie_plate_l', 'lacing_spacing'
                    ]
```

```
                if all(k in res and res[k] is not None
                    for k in required_keys):
                    best_ur = ur
                    break
            if best_ur is None:
                # Fallback: just pick the lowest UR
                best_ur = min(self.
                    optimum_section_ur_results.keys())
            best_result = self.optimum_section_ur_results
                [best_ur]
            # Assign all output fields for UI/terminal
            # Only assign actual calculated values for
                effective length and slenderness, never
                allow 'mpc' or similar text
            def get_numeric_value(keys):
                for k in keys:
                    v = best_result.get(k)
                    if v is not None:
                        # If value is a string and
                            contains 'mpc' or is not a
                            number, skip
                        if isinstance(v, str):
                            if 'mpc' in v.lower():
                                continue
                            try:
                                return float(v)
                            except Exception:
                                continue
                        elif isinstance(v, (int, float)):
                            return v
                return None

            # Only assign if not already set to a valid
                number
```

```
                if not (hasattr(self, 'effective_length_yy')
                    and isinstance(self.effective_length_yy, (
                    int, float)) and self.effective_length_yy
                    > 0):
                     self.effective_length_yy =
                        get_numeric_value(['
                        Effective_length_yy', 'Effective
                        Length YY', 'effective_length_yy'])
                self.effective_length_zz = get_numeric_value
                    (['Effective_length_zz', 'Effective Length
                     ZZ', 'effective_length_zz'])
                self.effective_sr_yy = get_numeric_value(['
                    Effective_sr_yy', 'Slenderness YY', '
                    effective_sr_yy', 'Slenderness_yy'])
                self.effective_sr_zz = get_numeric_value(['
                    Effective_sr_zz', 'Slenderness ZZ', '
                    effective_sr_zz', 'Slenderness_zz'])
                self.result_fcd = best_result.get('FCD')
                self.result_capacity = best_result.get('
                    Capacity')
                self.result_UR = best_ur
                self.result_section_class = best_result.get('
                    Section class')
                self.result_effective_area = best_result.get(
                    'Effective area')
                self.result_bc_yy = best_result.get('
                    Buckling_curve_yy')
                self.result_bc_zz = best_result.get('
                    Buckling_curve_zz')
                self.result_IF_yy = best_result.get('IF_yy')
                self.result_IF_zz = best_result.get('IF_zz')
                self.result_ebs_yy = best_result.get('EBS_yy'
                    )
```

```
                self.result_ebs_zz = best_result.get('EBS_zz'
                    )
                self.result_nd_esr_yy = best_result.get('
                    ND_ESR_yy')
                self.result_nd_esr_zz = best_result.get('
                    ND_ESR_zz')
                self.result_phi_yy = best_result.get('phi_yy'
                    )
                self.result_phi_zz = best_result.get('phi_zz'
                    )
                self.result_srf_yy = best_result.get('SRF_yy'
                    )
                self.result_srf_zz = best_result.get('SRF_zz'
                    )
                self.result_fcd_1_yy = best_result.get('
                    FCD_1_yy')
                self.result_fcd_1_zz = best_result.get('
                    FCD_1_zz')
                self.result_fcd_2 = best_result.get('FCD_2')
                self.result_cost = best_result.get('Cost')
                self.result_channel_spacing = best_result.get
                    ('channel_spacing')
                self.result_tie_plate_d = best_result.get('
                    tie_plate_d')
                self.result_tie_plate_t = best_result.get('
                    tie_plate_t')
                self.result_tie_plate_l = best_result.get('
                    tie_plate_l')
                self.result_lacing_spacing = best_result.get(
                    'lacing_spacing')
                # Print all calculated values for debug/
                    verification
                print("\n[DEBUG] All calculated values for
                    best section:")
```

```
                for k, v in best_result.items():
                    print(f"  {k}: {v}")
                print("\n[DEBUG] Attribute values set on self
                    :")
                attrs = [
                    'effective_length_yy', '
                        effective_length_zz', 'effective_sr_yy
                        ', 'effective_sr_zz', 'result_fcd', '
                        result_capacity',
                    'result_UR', 'result_section_class', '
                        result_effective_area', 'result_bc_yy'
                        , 'result_bc_zz', 'result_IF_yy', '
                        result_IF_zz',
                    'result_ebs_yy', 'result_ebs_zz', '
                        result_nd_esr_yy', 'result_nd_esr_zz',
                         'result_phi_yy', 'result_phi_zz', '
                        result_srf_yy',
                    'result_srf_zz', 'result_fcd_1_yy', '
                        result_fcd_1_zz', 'result_fcd_2', '
                        result_cost', 'result_channel_spacing'
                        ,
                    'result_tie_plate_d', 'result_tie_plate_t
                        ', 'result_tie_plate_l', '
                        result_lacing_spacing'
                ]
                for attr in attrs:
                    print(f"  {attr}: {getattr(self, attr,
                        None)}")
        except Exception as e:
            print(traceback.format_exc())

    def results(self):
        # Prevent duplicate logs in a single calculation
```

```
        if not hasattr(self, 'design_status_list') or self.
           design_status_list is None or not isinstance(self.
           design_status_list, list):
            self.design_status_list = []
        if hasattr(self, '_already_logged_failure'):
            del self._already_logged_failure

        if not self.optimum_section_ur:
            error_msg = "No sections available for design.
               Please check your input or section list."
            self.logger.error(error_msg)
            self.failed_reason = error_msg
            self.design_status = False
            self.failed_design_dict = {}
            return

        if len(self.optimum_section_ur) == 0:  # no design
           was successful
            if not hasattr(self, '_already_logged_failure'):
                self._already_logged_failure = True
                error_msg = "The sections selected by the
                   solver from the defined list of sections
                   did not satisfy the Utilization Ratio (UR)
                    criteria"
                self.failed_reason = error_msg
            self.design_status = False
            if self.failed_design_dict is None or not
               isinstance(self.failed_design_dict, dict):
                self.failed_design_dict = {}
            if self.failed_design_dict and isinstance(self.
               failed_design_dict, dict) and len(self.
               failed_design_dict) > 0:
                self.logger.info("The details for the best
                   section provided is being shown")
```

```
                self.result_UR = self.failed_design_dict.get(
                    'UR', None)
                self.common_result(
                    list_result=self.failed_design_dict,
                    result_type=None,
                )
                self.logger.warning("Re-define the list of
                    sections or check the Design Preferences
                    option and re-design.")
                return
            self.failed_design_dict = {}  # Always a dict for
                downstream code
            return

        _ = [i for i in self.optimum_section_ur if i > 1.0]

        if len(_)==1:
            temp = _[0]
        elif len(_)==0:
            temp = None
        else:
            temp = sorted(_)[0]
        self.failed_design_dict = self.
            optimum_section_ur_results[temp] if temp is not
            None else None

        # results based on UR
        if self.optimization_parameter == 'Utilization Ratio'
            :

            filter_UR = filter(lambda x: x <= min(self.
                allowable_utilization_ratio, 1.0), self.
                optimum_section_ur)
            self.optimum_section_ur = list(filter_UR)
```

```
            self.optimum_section_ur.sort()

            # selecting the section with most optimum UR
            if len(self.optimum_section_ur) == 0:  # no
                design was successful
                error_msg = f"The sections selected by the
                    solver from the defined list of sections
                    did not satisfy the Utilization Ratio (UR)
                     criteria. Allowable UR: {self.
                    allowable_utilization_ratio}"
                self.failed_reason = error_msg
                self.design_status = False

                # Fallback: If we have results but they were
                    filtered out, show the best one anyway
                if hasattr(self, 'optimum_section_ur_results'
                    ) and self.optimum_section_ur_results:
                    self.logger.info("Showing best available
                        result despite UR filter failure")
                    best_ur = min(self.
                        optimum_section_ur_results.keys())
                    self.result_UR = best_ur
                    self.common_result(
                        list_result=self.
                            optimum_section_ur_results,
                        result_type=best_ur,
                    )
                    return

                if self.failed_design_dict and isinstance(
                    self.failed_design_dict, dict) and len(
                    self.failed_design_dict) > 0:
                    self.logger.info(
```

```
                "The details for the best section
                    provided is being shown"
            )
                self.result_UR = self.failed_design_dict.
                    get('UR', None) #temp
                self.common_result(
                    list_result=self.failed_design_dict,
                    result_type=None,
                )
                self.logger.warning(
                "Re-define the list of sections or check
                    the Design Preferences option and re-
                    design."
            )
                return

        self.failed_design_dict = {}
        self.result_UR = self.optimum_section_ur[-1]  #
            optimum section which passes the UR check

        self.design_status = True
        if self.result_UR in self.
            optimum_section_ur_results:
            self.common_result(
                list_result=self.
                    optimum_section_ur_results,
                result_type=self.result_UR,
            )
        else:
            error_msg = f"Result UR {self.result_UR} not
                found in optimum_section_ur_results. No
                valid design result to display."
            self.logger.error(error_msg)
            self.failed_reason = error_msg
```

```
                self.design_status = False
        else:  # results based on cost
            self.optimum_section_cost.sort()

            # selecting the section with most optimum cost
            self.result_cost = self.optimum_section_cost[0]
            self.design_status = True

        for status in self.design_status_list:
            if status is False:
                self.design_status = False
                break
            else:
                self.design_status = True

    def common_result(self, list_result, result_type):
        # Defensive: handle None or wrong type for
           list_result
        if not isinstance(list_result, dict) or not
           list_result:
            self.logger.error("No valid results to display.
               Calculation did not yield any results.")
            # Set all result attributes to None or a safe
               default
            self.result_designation = None
            self.section_class = None
            self.result_section_class = None
            self.result_effective_area = None
            self.result_bc_zz = None
            self.result_bc_yy = None
            self.result_IF_zz = None
            self.result_IF_yy = None
            self.result_eff_len_zz = None
            self.result_eff_len_yy = None
```

```
            self.result_eff_sr_zz = None
            self.result_eff_sr_yy = None
            self.result_ebs_zz = None
            self.result_ebs_yy = None
            self.result_nd_esr_zz = None
            self.result_nd_esr_yy = None
            self.result_phi_zz = None
            self.result_phi_yy = None
            self.result_srf_zz = None
            self.result_srf_yy = None
            self.result_fcd_1_zz = None
            self.result_fcd_1_yy = None
            self.result_fcd_2 = None
            self.result_fcd_zz = None
            self.result_fcd_yy = None
            self.result_fcd = None
            self.result_capacity = None
            self.result_cost = None
            return

        # Defensive: handle None or missing result_type
        if result_type is None:
            # Try to get the first key if possible
            if list_result:
                result_type = next(iter(list_result.keys()))
            else:
                self.logger.error("No result type found in
                    results.")
                return

        # Defensive: check if result_type exists in
            list_result
        if result_type not in list_result:
```

```
            self.logger.error(f"Result type '{result_type}'
                not found in results.")
            return

        # Now safe to access
        try:
            self.result_designation = list_result[result_type
                ].get('Designation', None)
            self.section_class = self.
                input_section_classification.get(self.
                result_designation, [None])[0]

            if self.section_class == 'Slender':
                self.logger.warning(f"The trial section ({
                    self.result_designation}) is Slender.
                    Computing the Effective Sectional Area as
                    per Sec. 9.7.2, Fig. 2 (B & C) of The
                    National Building Code of India (NBC),
                    2016.")
            if getattr(self, 'effective_area_factor', 1.0) <
                1.0:
                self.effective_area = round(self.
                    effective_area * self.
                    effective_area_factor, 2)
                self.logger.info(f"The actual effective area
                    is {round((self.effective_area / self.
                    effective_area_factor), 2)} mm2 and the
                    reduced effective area is {self.
                    effective_area} mm2 [Reference: Cl. 7.3.2,
                     IS 800:2007]")
            else:
                if self.result_designation in self.
                    input_section_classification:
                    def safe_round(value, decimals=2):
```

```
                if value is None:
                    return None
                try:
                    return round(float(value),
                        decimals)
                except (ValueError, TypeError):
                    return None

            classification = self.
                input_section_classification[self.
                result_designation]
            flange_value = safe_round(classification[3]
                if len(classification) > 3 else None)
            web_value = safe_round(classification[4] if
                len(classification) > 4 else None)

            self.logger.info(
                "The section is {}. The {} section  has
                    {} flange({}) and  {} web({}).  [
                    Reference: Cl 3.7, IS 800:2007].".
                    format(
                     classification[0] if len(
                        classification) > 0 else 'Unknown'
                        ,
                     self.result_designation,
                     classification[1] if len(
                        classification) > 1 else 'Unknown'
                        , flange_value,
                     classification[2] if len(
                        classification) > 2 else 'Unknown'
                        , web_value
                ))
```

```
            self.result_section_class = list_result[
               result_type].get('Section class', None)
            self.result_effective_area = list_result[
               result_type].get('Effective area', None)
            self.result_bc_zz = list_result[result_type].get(
               'Buckling_curve_zz', None)
            self.result_bc_yy = list_result[result_type].get(
               'Buckling_curve_yy', None)
            self.result_IF_zz = list_result[result_type].get(
               'IF_zz', None)
            self.result_IF_yy = list_result[result_type].get(
               'IF_yy', None)
            self.result_eff_len_zz = list_result[result_type
               ].get('Effective_length_zz', None)
            self.result_eff_len_yy = list_result[result_type
               ].get('Effective_length_yy', None)
            self.result_eff_sr_zz = list_result[result_type].
               get('Effective_SR_zz', None)
            self.result_eff_sr_yy = list_result[result_type].
               get('Effective_SR_yy', None)
            self.result_ebs_zz = list_result[result_type].get
               ('EBS_zz', None)
            self.result_ebs_yy = list_result[result_type].get
               ('EBS_yy', None)
            self.result_nd_esr_zz = list_result[result_type].
               get('ND_ESR_zz', None)
            self.result_nd_esr_yy = list_result[result_type].
               get('ND_ESR_yy', None)
            self.result_phi_zz = list_result[result_type].get
               ('phi_zz', None)
            self.result_phi_yy = list_result[result_type].get
               ('phi_yy', None)
            self.result_srf_zz = list_result[result_type].get
               ('SRF_zz', None)
```

```
            self.result_srf_yy = list_result[result_type].get
                ('SRF_yy', None)
            self.result_fcd_1_zz = list_result[result_type].
                get('FCD_1_zz', None)
            self.result_fcd_1_yy = list_result[result_type].
                get('FCD_1_yy', None)
            self.result_fcd_2 = list_result[result_type].get(
                'FCD_2', None)
            self.result_fcd_zz = list_result[result_type].get
                ('FCD_zz', None)
            self.result_fcd_yy = list_result[result_type].get
                ('FCD_yy', None)
            self.result_fcd = list_result[result_type].get('
                FCD', None)
            self.result_capacity = list_result[result_type].
                get('Capacity', None)
            self.result_cost = list_result[result_type].get('
                Cost', None)
        except Exception as e:
            # Set all result attributes to None or a safe
                default
            self.result_designation = None
            self.section_class = None
            self.result_section_class = None
            self.result_effective_area = None
            self.result_bc_zz = None
            self.result_bc_yy = None
            self.result_IF_zz = None
            self.result_IF_yy = None
            self.result_eff_len_zz = None
            self.result_eff_len_yy = None
            self.result_eff_sr_zz = None
            self.result_eff_sr_yy = None
            self.result_ebs_zz = None
```

```
            self.result_ebs_yy = None
            self.result_nd_esr_zz = None
            self.result_nd_esr_yy = None
            self.result_phi_zz = None
            self.result_phi_yy = None
            self.result_srf_zz = None
            self.result_srf_yy = None
            self.result_fcd_1_zz = None
            self.result_fcd_1_yy = None
            self.result_fcd_2 = None
            self.result_fcd_zz = None
            self.result_fcd_yy = None
            self.result_fcd = None

    def save_design(self, popup_summary):
        # Safe rounding function for all round operations
        def safe_round(value, decimals=2):
            if value is None:
                return None
            try:
                return round(float(value), decimals)
            except (ValueError, TypeError):
                return None

        # Safe access to classification values
        def safe_classification_value(designation, index,
           default=None):
            if (designation in self.
               input_section_classification and
                isinstance(self.input_section_classification[
                   designation], (list, tuple)) and
                len(self.input_section_classification[
                   designation]) > index):
```

```
                return self.input_section_classification[
                    designation][index]
            return default

        if self.design_status:
            if (self.design_status and self.
                failed_design_dict is None) or (not self.
                design_status and self.failed_design_dict is
                not None and hasattr(self.failed_design_dict,
                '__len__') and len(self.failed_design_dict) >
                0):
                 if self.sec_profile=='Columns' or self.
                    sec_profile=='Beams' or self.sec_profile
                    == VALUES_SEC_PROFILE[0]:
                     try:
                         result = Beam(designation=self.
                             result_designation, material_grade
                             =self.material)
                     except:
                         result = Column(designation=self.
                             result_designation, material_grade
                             =self.material)
                     self.section_property = result
                     self.report_column = {
                        KEY_DISP_SEC_PROFILE: "ISection",
                                              KEY_DISP_SECSIZE: (
                                                 self.
                                                 section_property.
                                                 designation, self.
                                                 sec_profile),
                                              KEY_DISP_COLSEC_REPORT
                                                 : self.
                                                 section_property.
                                                 designation,
```

```
                                    KEY_DISP_MATERIAL:
                                        self.
                                        section_property.
                                        material,
            #
                KEY_DISP_APPLIED_AXIAL_FORCE: self.
                section_property.,
                                    KEY_REPORT_MASS: self
                                        .section_property.
                                        mass,
                                    KEY_REPORT_AREA:
                                        safe_round(self.
                                        section_property.
                                        area * 1e-2, 2),
                                    KEY_REPORT_DEPTH:
                                        self.
                                        section_property.
                                        depth,
                                    KEY_REPORT_WIDTH:
                                        self.
                                        section_property.
                                        flange_width,
                                    KEY_REPORT_WEB_THK:
                                        self.
                                        section_property.
                                        web_thickness,
                                    KEY_REPORT_FLANGE_THK
                                        : self.
                                        section_property.
                                        flange_thickness,
                                    KEY_DISP_FLANGE_S_REPORT
                                        : self.
                                        section_property.
                                        flange_slope,
```

```
                                    KEY_REPORT_R1: self.
                                       section_property.
                                       root_radius,
                                    KEY_REPORT_R2: self.
                                       section_property.
                                       toe_radius,
                                    KEY_REPORT_IZ: round(
                                       self.
                                       section_property.
                                       mom_inertia_z * 1e
                                       -4, 2),
                                    KEY_REPORT_IY: round(
                                       self.
                                       section_property.
                                       mom_inertia_y * 1e
                                       -4, 2),
                                    KEY_REPORT_RZ: round(
                                       self.
                                       section_property.
                                       rad_of_gy_z * 1e
                                       -1, 2),
                                    KEY_REPORT_RY: round(
                                       self.
                                       section_property.
                                       rad_of_gy_y * 1e
                                       -1, 2),
                                    KEY_REPORT_ZEZ: round
                                       (self.
                                       section_property.
                                       elast_sec_mod_z *
                                       1e-3, 2),
                                    KEY_REPORT_ZEY: round
                                       (self.
                                       section_property.
```

```
                                        elast_sec_mod_y *
                                        1e-3, 2),
                                    KEY_REPORT_ZPZ: round
                                        (self.
                                        section_property.
                                        plast_sec_mod_z *
                                        1e-3, 2),
                                    KEY_REPORT_ZPY: round
                                        (self.
                                        section_property.
                                        plast_sec_mod_y *
                                        1e-3, 2)}
            else:
                #Update for section profiles RHS and SHS,
                    CHS by making suitable elif condition
                    .
                self.report_column = {
                    KEY_DISP_COLSEC_REPORT: getattr(self.
                    section_property, 'designation', None)
                    ,
                                    KEY_DISP_MATERIAL:
                                        getattr(self.
                                        section_property,
                                        'material', ''),
                                    #

                                        KEY_DISP_APPLIED_AXIAL_FORCE
                                        : getattr(self.
                                        section_property,
                                        '
                                        applied_axial_force
                                        ', ''),
                                    KEY_REPORT_MASS:
                                        getattr(self.
```

```
                                    section_property,
                                    'mass', ''),
                                KEY_REPORT_AREA:
                                    safe_round(getattr
                                    (self.
                                    section_property,
                                    'area', 0) * 1e-2,
                                     2),
                                KEY_REPORT_DEPTH:
                                    getattr(self.
                                    section_property,
                                    'depth', ''),
                                KEY_REPORT_WIDTH:
                                    getattr(self.
                                    section_property,
                                    'flange_width', ''
                                    ),
                                KEY_REPORT_WEB_THK:
                                    getattr(self.
                                    section_property,
                                    'web_thickness', '
                                    '),
                                KEY_REPORT_FLANGE_THK
                                    : getattr(self.
                                    section_property,
                                    'flange_thickness'
                                    , ''),
                                KEY_DISP_FLANGE_S_REPORT
                                    : getattr(self.
                                    section_property,
                                    'flange_slope', ''
                                    )}
```

```
                self.report_input = \
                    {#KEY_MAIN_MODULE: self.mainmodule,
                    KEY_MODULE: self.module, #"Axial load on
                        column "
                            KEY_DISP_AXIAL: self.load.axial_force
                                * 10 ** -3,
                            KEY_DISP_ACTUAL_LEN_ZZ: self.
                                length_zz,
                            KEY_DISP_ACTUAL_LEN_YY: self.
                                length_yy,
                            KEY_DISP_SEC_PROFILE: self.
                                sec_profile,
                            KEY_DISP_SECSIZE: self.
                                result_section_class,
                            KEY_DISP_END1: self.end_1_z,
                            KEY_DISP_END2: self.end_2_z,
                            KEY_DISP_END1_Y: self.end_1_y,
                            KEY_DISP_END2_Y: self.end_2_y,
                            "Column Section - Mechanical
                                Properties": "TITLE",
                    KEY_MATERIAL: self.material,
                            KEY_DISP_ULTIMATE_STRENGTH_REPORT:
                                self.material_property.fu,
                            KEY_DISP_YIELD_STRENGTH_REPORT: self.
                                material_property.fy,
                            KEY_DISP_EFFECTIVE_AREA_PARA: self.
                                effective_area_factor, #To Check
                            KEY_DISP_SECSIZE:  str(self.sec_list)
                                ,
                            "Selected Section Details": self.
                                report_column,
                    }

                self.report_check = []
```

```
                t1 = ('Selected', 'Selected Member Data', '|p
                    {5cm}|p{2cm}|p{2cm}|p{2cm}|p{4cm}|')
                self.report_check.append(t1)

                self.h = (self.section_property.depth - 2 * (
                    self.section_property.flange_thickness +
                    self.section_property.root_radius))
                self.h_bf_ratio = self.h / self.
                    section_property.flange_width


                # 2.2 CHECK: Buckling Class - Compatibility
                    Check
                t1 = ('SubSection', 'Buckling Class -
                    Compatibility Check', '|p{4cm}|p{3.5cm}|p
                    {6.5cm}|p{2cm}|')
                self.report_check.append(t1)

                # YY axis row
                t1 = (
                    "h/bf and tf for YY Axis",
                    comp_column_class_section_check_required(
                        self.h, self.section_property.
                        flange_width, self.section_property.
                        flange_thickness, "YY"),
                    comp_column_class_section_check_provided(
                        self.h, self.section_property.
                        flange_width, self.section_property.
                        flange_thickness, round(self.
                        h_bf_ratio, 2), "YY"), 'Compatible'
                )
                self.report_check.append(t1)

                # ZZ axis row
```

```
                t1 = (
                    "h/bf and tf for ZZ Axis",
                    comp_column_class_section_check_required(
                        self.h, self.section_property.
                        flange_width, self.section_property.
                        flange_thickness, "ZZ"),
                    comp_column_class_section_check_provided(
                        self.h, self.section_property.
                        flange_width, self.section_property.
                        flange_thickness, round(self.
                        h_bf_ratio, 2), "ZZ"), 'Compatible'
                )
                self.report_check.append(t1)

                t1 = ('SubSection', 'Section Classification',
                    '|p{3cm}|p{3.5cm}|p{8.5cm}|p{1cm}|')
                self.report_check.append(t1)
                t1 = ('Web Class', 'Axial Compression',
                        cl_3_7_2_section_classification_web(
                            round(self.h, 2), round(self.
                            section_property.web_thickness, 2)
                            ,
                                                        safe_round
                                                            (
                                                            safe_classi
                                                            (
                                                            self
                                                            .
                                                            result_desi
                                                            ,

                                                            4)
                                                            )
                                                            ,
```

```
                                                    self
                                                        .
                                                        epsilon
                                                        ,

                                                        self
                                                        .
                                                        section_pro
                                                        .
                                                        type
                                                        ,

                                                    safe_classific
                                                        (
                                                        self
                                                        .
                                                        result_desi
                                                        ,

                                                        2)
                                                        )
                                                        ,


                        ' ')
            self.report_check.append(t1)
            t1 = ('Flange Class', self.section_property.
                type,
                      cl_3_7_2_section_classification_flange
                         (round(self.section_property.
                         flange_width/2, 2),
                                                      round
                                                        (
                                                        self
```

```
                                                    .
                                                    section_p
                                                    .
                                                    flange_th
                                                    ,

                                                    2)
                                                    ,

                                safe_round(
                                    safe_classification_value
                                    (self.
                                    result_designation, 3)
                                    ),
                                                        self
                                                        .
                                                        epsilon
                                                        ,

                                                        safe_classif
                                                        (
                                                        self
                                                        .
                                                        result_de
                                                        ,

                                                        1)
                                                        )
                                                        ,

                        ' ')
                self.report_check.append(t1)
                t1 = ('Section Class', ' ',
                        cl_3_7_2_section_classification(
```

```
                                                        self
                                                            .
                                                            input_sec
                                                            [
                                                            self
                                                            .
                                                            result_de
                                                            ][0])
                                                            ,

                    ' ')
                self.report_check.append(t1)


                t1 = ('NewTable', 'Imperfection Factor', '|p
                    {3cm}|p{5 cm}|p{5cm}|p{3 cm}|')
                self.report_check.append(t1)

                t1 = (
                    'YY',
                    self.list_yy[3].upper(),
                    self.list_yy[4], ''
                )
                self.report_check.append(t1)

                t1 = (
                    'ZZ',
                    self.list_zz[3].upper(),
                    self.list_zz[4], ''
                )
                self.report_check.append(t1)
```

```
                # Defensive checks for None before division/
                    round
                if self.result_eff_len_yy is not None and
                    self.length_yy:
                    K_yy = self.result_eff_len_yy / self.
                        length_yy
                else:
                    K_yy = None
                if self.result_eff_len_zz is not None and
                    self.length_zz:
                    K_zz = self.result_eff_len_zz / self.
                        length_zz
                else:
                    K_zz = None
                t1 = ('SubSection', 'Slenderness Ratio', '|p
                    {4cm}|p{2 cm}|p{7cm}|p{3 cm}|')
                self.report_check.append(t1)
                val_yy = safe_float(self.result_eff_sr_yy)
                val_zz = safe_float(self.result_eff_sr_zz)
                val_yy_rounded = round(val_yy if val_yy is
                    not None else 0.0, 2)
                val_zz_rounded = round(val_zz if val_zz is
                    not None else 0.0, 2)
                t1 = ("Effective Slenderness Ratio (For YY
                    Axis)", ' ',
                        cl_7_1_2_effective_slenderness_ratio(
                            K_yy, self.length_yy, self.
                            section_property.rad_of_gy_y,
                            val_yy_rounded),
                        ' ')
                self.report_check.append(t1)
                t1 = ("Effective Slenderness Ratio (For ZZ
                    Axis)", ' ',
```

```
                cl_7_1_2_effective_slenderness_ratio(
                    K_zz, self.length_zz, self.
                    section_property.rad_of_gy_z,
                    val_zz_rounded),
                ' ')
            self.report_check.append(t1)



            t1 = ('SubSection', 'Checks', '|p{4cm}|p{2 cm
               }|p{7cm}|p{3 cm}|')
            self.report_check.append(t1)

            t1 = (r'$\phi_{yy}$', ' ',
                cl_8_7_1_5_phi(self.result_IF_yy,
                    safe_round(self.non_dim_eff_sr_yy, 2),
                     safe_round(self.result_phi_yy, 2)),
                ' ')
            self.report_check.append(t1)

            t1 = (r'$\phi_{zz}$', ' ',
                cl_8_7_1_5_phi(self.result_IF_zz,
                    safe_round(self.non_dim_eff_sr_zz, 2),
                     safe_round(self.result_phi_zz, 2)),
                ' ')
            self.report_check.append(t1)

            t1 = (r'$F_{cd,yy} \, \left( \frac{N}{\text{
               mm}^2} \right)$', ' ',
                cl_8_7_1_5_Buckling(
                    str(self.material_property.fy) if
                        self.material_property.fy is not
                        None else '',
```

```
                    str(self.gamma_m0) if self.gamma_m0
                        is not None else '',
                    str(safe_round(self.non_dim_eff_sr_yy
                        , 2)),
                    str(safe_round(self.result_phi_yy, 2)
                        ),
                    str(safe_round(self.result_fcd_2, 2))
                        ,
                    str(safe_round(self.result_fcd_yy, 2)
                        ),
                ),
                ' ')
            self.report_check.append(t1)

            t1 = (r'$F_{cd,zz} \, \left( \frac{N}{\text{
                mm}^2} \right)$', ' ',
                 cl_8_7_1_5_Buckling(
                    str(self.material_property.fy) if
                        self.material_property.fy is not
                        None else '',
                    str(self.gamma_m0) if self.gamma_m0
                        is not None else '',
                    str(safe_round(self.non_dim_eff_sr_zz
                        , 2)),
                    str(safe_round(self.result_phi_zz, 2)
                        ),
                    str(safe_round(self.result_fcd_2, 2))
                        ,
                    str(safe_round(self.result_fcd_zz, 2)
                        ),
                ),
                ' ')
            self.report_check.append(t1)
```

```
                # Defensive: check for None before division/
                    round for result_capacity and result_fcd
                cap = self.result_capacity if self.
                    result_capacity is not None else 0.0
                fcd = self.result_fcd if self.result_fcd is
                    not None else 0.0
                area = self.section_property.area if self.
                    section_property and hasattr(self.
                    section_property, 'area') else 0.0
                t1 = (r'Design Compressive Strength (\( P_d
                    \)) (For the most critical value of \( F_{
                    cd} \))', self.load.axial_force * 10 **
                    -3,
                     cl_7_1_2_design_compressive_strength(
                         safe_round(cap / 1000, 2), area,
                         safe_round(fcd, 2), self.load.
                         axial_force * 10 ** -3),
                     get_pass_fail(self.load.axial_force * 10
                         ** -3, safe_round(cap, 2), relation="
                         leq"))
                self.report_check.append(t1)

            else:
                self.report_input = \
                    {#KEY_MAIN_MODULE: self.mainmodule,
                    KEY_MODULE: self.module, #"Axial load on
                        column "
                         KEY_DISP_AXIAL: self.load.axial_force
                             * 10 ** -3,
                         KEY_DISP_ACTUAL_LEN_ZZ: self.
                             length_zz,
                         KEY_DISP_ACTUAL_LEN_YY: self.
                             length_yy,
```

```
                    KEY_DISP_SEC_PROFILE: self.
                        sec_profile,
                    KEY_DISP_SECSIZE:  str(self.sec_list)
                        ,
                    #KEY_DISP_SECSIZE: self.
                        result_section_class,
                    KEY_DISP_END1: self.end_1_z,
                    KEY_DISP_END2: self.end_2_z,
                    KEY_DISP_END1_Y: self.end_1_y,
                    KEY_DISP_END2_Y: self.end_2_y,
                    "Column Section - Mechanical
                        Properties": "TITLE",
                KEY_MATERIAL: self.material,
                    KEY_DISP_ULTIMATE_STRENGTH_REPORT:
                        self.material_property.fu,
                    KEY_DISP_YIELD_STRENGTH_REPORT: self.
                        material_property.fy,
                    KEY_DISP_EFFECTIVE_AREA_PARA: self.
                        effective_area_factor, #To Check

                    # "Failed Section Details": self.
                        report_column,
                }
            self.report_check = []

            t1 = ('Selected', 'All Members Failed', '|p{5
                cm}|p{2cm}|p{2cm}|p{2cm}|p{4cm}|')
            self.report_check.append(t1)


        Disp_2d_image = []
        Disp_3D_image = "/ResourceFiles/images/3d.png"
```

```
            rel_path = str(sys.path[0])
            rel_path = os.path.abspath(".") # TEMP
            rel_path = rel_path.replace("\\", "/")
            fname_no_ext = popup_summary['filename']
            CreateLatex.save_latex(CreateLatex(), self.
               report_input, self.report_check, popup_summary
               , fname_no_ext,
                                   rel_path, Disp_2d_image,
                                      Disp_3D_image, module=
                                      self.module)

    def get_end_conditions(self, *args):
        """
        Returns the list of standard end conditions for both
           y-y and z-z axes.
        These values are used in dropdowns for End 1 and End
           2.
        """
        return ["Fixed", "Pinned", "Free"]
    def show_dialog(self, dialog):
        """
        Show a dialog modally, but only if not already open.
           Track it for later closing.
        """
        if dialog in self.dialogs:
            if dialog.isVisible():
                dialog.raise_()
                dialog.activateWindow()
                return
        self.dialogs.append(dialog)
        dialog.setModal(True)
        dialog.exec_()
        if dialog in self.dialogs:
            self.dialogs.remove(dialog)
```

```
    def reset_state_for_new_design(self):
        """
        Reset only calculation/output state for a new design,
            without closing dialogs or clearing input widgets
           .
        """
        self.reset_output_state()
        self.design_status = False
        self.failed_reason = None
        self.result = {}
        self.utilization_ratio = 0
        self.area = 0
        self.epsilon = 1.0
        self.fy = 0
        self.section = None
        self.weld_size = ''
        self.weld_type = ''
        self.weld_strength = 0
        self.lacing_incl_angle = 0
        self.lacing_section = ''
        self.lacing_type = ''
        self.allowed_utilization = ''
        self.section_designation = None
        self.output_title_fields = {}
        self.material_lookup_cache = {}
        self.optimum_section_cost_results = {}
        self.optimum_section_cost = []
```

Chapter 4

Debugging and Documentation

4.1 Problem Statement

The main goal was to improve and validate the accuracy and reliability of the structural steel design software, especially for modules related to column-to-column cover plate connections (both welded and bolted):

1. **Verification and Report Generation:** Making sure the software's calculations and checks match standard references (like N. Subramanian's book and IS 800:2007).
2. **Development of Design and detailing checklists (DDCLs):** Creating step-by-step checklists for the design process, ensuring nothing is missed and all code requirements are met.
3. **Generation of OSI Files:** Turning textbook example problems into OSI files for automated software testing.

4.2 Tasks Done

The following tasks were successfully completed as a part of the above task:

1. **Module and GUI Development:**
   - Developed the user interface (GUI) for the Column-to-Column Cover Plate modules (both welded and bolted).
   - Implemented the logic to take user inputs, trigger calculations, and display the results clearly in the output dock.
   - Ensured the GUI is user-friendly and all necessary design parameters can be entered and reviewed.
2. **Integration of Calculation Logic:**
   - Integrated the calculation logic (provided or reviewed by a partner) into the module so that results are automatically shown to the user.
   - Made sure that all intermediate and final calculation steps are visible in the output, helping users understand the design process.
3. **Verification and Collaboration:**
   - Worked closely with a partner to verify that the displayed results match the expected values from standard references (like N. Subramanian's book and IS 800:2007).
   - Updated the module as needed based on feedback and verification results.
4. **Support for Reporting and Testing:**
   - Enabled the export or display of calculation reports directly from the GUI, making it easier to generate verification reports.
   - Helped in preparing or testing OSI files by ensuring the GUI can accept and process structured input data.

Chapter 5

Conclusions

5.1 Tasks Accomplished

- Designed a reimagined version of the Graphical User Interface (GUI) for *Osdag*, focusing on improved usability and modern aesthetics.
- Contributed to the development of core functions for the *Laced Column* module, enabling programmable execution of individual design and verification tasks as per IS 800:2007.
- Reviewed and validated the design calculation reports for the *Column-to-Column Bolted* and *Welded Cover Plate* modules to ensure accuracy in both functional implementation and numerical results.
- Authored the **Design and Detailing Checklists (DDCL)** for both bolted and welded cover plate connection modules, based on standard design practices and codal provisions.

5.2 Skills Developed

- Canva
- LaTeX
- Teamwork
- Communication
- Python
- IS 800 code

Chapter A

Appendix

A.1 Work Reports

Internship Work Report

Name: Sanket Gaikwad
Project: LacedColumn Module Development
Internship: FOSSEE Summer Fellowship 2025

Date	Day	Task	Hours Worked
19/05/2025	Monday	Attended the first meeting with the team and mentor. Roles and responsibilities were explained. Joined the communication channel on Discord.	7
20/05/2025	Tuesday	Participated in the initial meeting with the module development team to discuss the workflow. Installed Osdag, Anaconda, GitBash, and PyCharm.	7
21/05/2025	Wednesday	Set up the development environment and ensured all files were installed. Raised doubts on Discord for clarification.	6
22/05/2025	Thursday	Familiarized with the environment, studied the repository, and began work on Task.	8
23/05/2025	Friday	I discussed the task with Sir, who provided all the necessary details about the LacedColumn module, and I've now started working on its development.	8
24/05/2025	Saturday	Sir provided me the DDCL for the laced column module—I overviewed it and started developing the module accordingly.	9
26/05/2025	Monday	Referred to Column.py.	8
27–02/05-06/2025	Tue–Mon	Continued development of the Laced Column module.	25
03/06/2025	Tuesday	Attended meetings to give updates and continued module development.	16
04-14/05–02/06/2025	Wed–Sat	Development of the Laced Column module.	32
16/06/2025	Monday	Meeting to share progress and clarify doubts.	8
17/06/2025	Tuesday	Continued module development and attended a clarification session.	12
18/06/2025	Wednesday	Modified functions to align with the format used in *IS800:2007.py*.	8
19/06/2025	Thursday	Meeting to review progress and address doubts.	8
20/06/2025	Friday	Understood the procedure for calculation.	7
23-07/06-07/2025	Mon–Mon	started working on the calculations part before pushing it to the output dock. I'm printing intermediate results to the log window and verifying them with my partner.	32
08/07/2025	Tuesday	Participated in a review meeting to update on report verification.	8
10/07/2025	Thursday	Final meeting to review tasks completed and begin submission checks.	8
15/07/2025	Thursday	Final corrections and submission of all assigned tasks.	8

Bibliography

[1] Siddhartha Ghosh, Danish Ansari, Ajmal Babu Mahasrankintakam, Dharma Teja Nuli, Reshma Konjari, M. Swathi, and Subhrajit Dutta. Osdag: A Software for Structural Steel Design Using IS 800:2007. In Sondipon Adhikari, Anjan Dutta, and Satyabrata Choudhury, editors, *Advances in Structural Technologies*, volume 81 of *Lecture Notes in Civil Engineering*, pages 219–231, Singapore, 2021. Springer Singapore.

[2] FOSSEE Project. FOSSEE News - January 2018, vol 1 issue 3. Accessed: 2024-12-05.

[3] FOSSEE Project. Osdag website. Accessed: 2024-12-05.