FOSSEE Semester Long Internship Report

On

Osdag Desktop and LCC Web

Submitted by

Harshan S

2nd Year B.E Student, Department of Automobile Engineering

Madras Institute of Technology, Anna University

Chennai

Under the Guidance of

Prof. Siddhartha Ghosh

Department of Civil Engineering

Indian Institute of Technology Bombay

Mentors:

Ajmal Babu M S

Parth Karia

Ajinkya Dahale

July 6, 2025

Acknowledgments

- Start with a general statement of thanks. Express your overall gratitude to everyone who supported you during your project or research.
- Project staff at the Osdag team, Ajmal Babu M. S., Ajinkya Dahale, and Parth Karia,
- 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
- Acknowledge the support from the National Mission on Education through Information and Communication Technology (ICT), Ministry of Education (MoE), Government of India, for their role in facilitating this project
- Acknowledge your colleagues who worked with you during your internship or project.
- If appropriate, thank your college, department, head, and principal for their support during your studies.

Contents

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.

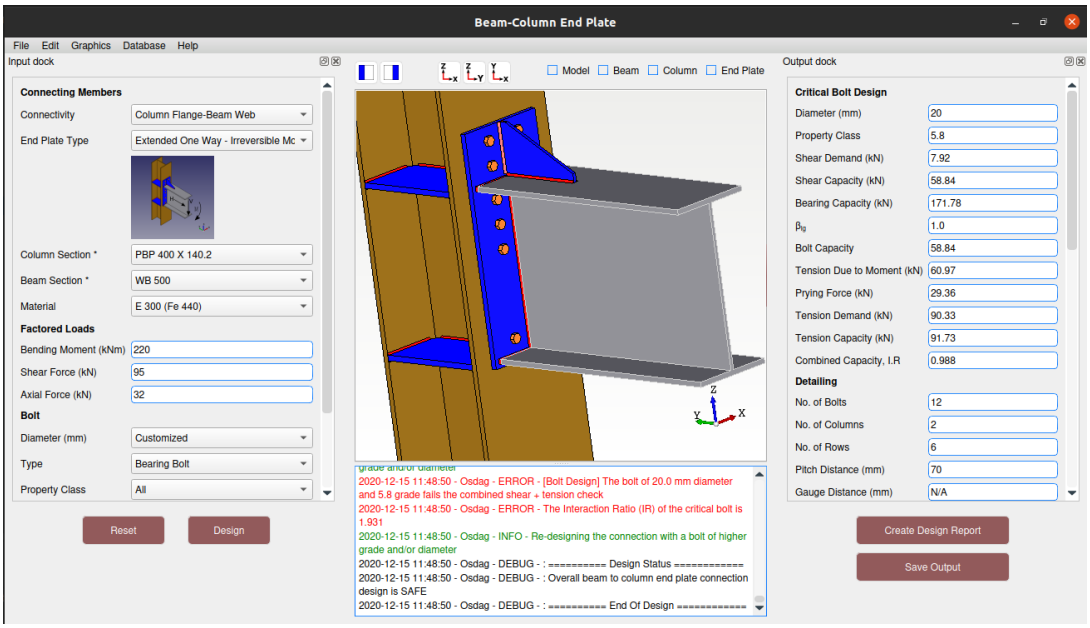


Figure 1.2: Osdag GUI

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

The problem statement which I was given for the Screening task for semester long internship was to develop a CAD model for a house like structure

2.2 Tasks Done

1) I have learned to use PythonOCC module to create 3D modules using the sample code which was given in the screening task
2) I created the 3D CAD model using pythonocc module

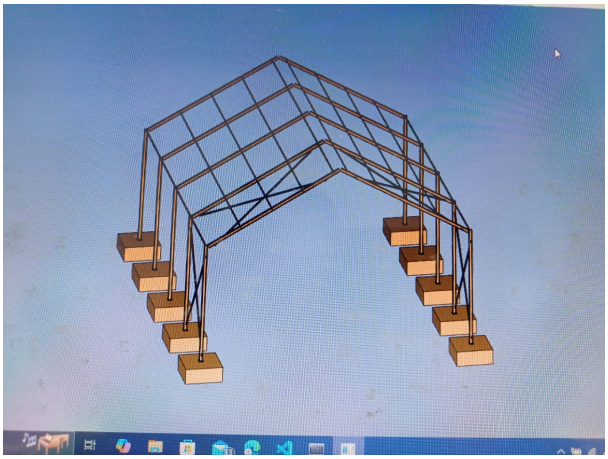


Figure 2.1: Screening Task

Chapter 3

Welded Lap Joint

3.1 Welded Lap Joint

The problem statement for this task was to create a welded lap joint for Osdag using pythonocc

3.2 Tasks Done

I have created the welded lap joint CAD model using pythonocc module

3.3 Python Code

Include codes developed during your internship. Give a proper description. Example is given here. This section presents a Python script for designing a beam-to-column connection using the Osdag framework. The script performs calculations based on user-defined parameters like beam section, column section, and applied shear load. It automates the design process, ensuring compliance with Indian Standards (IS) codes.

3.3.1 Description of the Script

The script is structured as follows:

- **Input Parameters**: The user specifies the beam and column sections, applied shear load, and bolt properties.

- **Design Calculations**: The program computes the number of bolts required, their arrangement, and verifies the adequacy of the end plate and bolt group.
- **Output**: Results include the number of bolts, their layout, and adequacy checks for the connection components.

3.3.2 Python Code

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

Listing 3.1: Beam-to-Column Connection Design in Osdag

```
%--------------------begin code-------------

import os
import sys
import math
from osdag.design_shear import ShearConnection
from osdag.connection_geometry import ConnectionGeometry

# Input parameters
beam_section = "ISMB 300"  # Beam section
column_section = "ISWB 500"  # Column section
load = 250  # Shear load in kN
bolt_diameter = 16  # Diameter of bolts in mm
bolt_grade = 8.8  # Grade of bolts

# 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)

%-------------------- end code ---------------
```

3.3.3 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.

3.3.4 Full code

```
import math


# Material and geometric properties
def calc_bolt_shear_capacity(bolt_diameter,
   ultimate_tensile_strength, gamma_mb):
    """
    Calculate the shear capacity of a single bolt.

    Parameters:
        bolt_diameter (float): Diameter of the bolt (in mm).
        ultimate_tensile_strength (float): Ultimate tensile
           strength of the bolt material (in MPa).
        gamma_mb (float): Partial safety factor for bolts.

    Returns:
        float: Shear capacity of the bolt in kN.
```

```
    """
    nominal_shear_strength = 0.6 * math.pi * (bolt_diameter / 2)
        **2 * ultimate_tensile_strength
    shear_capacity = nominal_shear_strength / gamma_mb
    return shear_capacity / 1000  # Convert to kN


def calc_bearing_capacity(bolt_diameter, plate_thickness,
    fu_plate, e, p, gamma_mb):
    """
    Calculate the bearing capacity of a single bolt.

    Parameters:
        bolt_diameter (float): Diameter of the bolt (in mm).
        plate_thickness (float): Thickness of the connected plate
            (in mm).
        fu_plate (float): Ultimate tensile strength of the plate
            material (in MPa).
        e (float): Edge distance (in mm).
        p (float): Pitch distance (in mm).
        gamma_mb (float): Partial safety factor for bolts.

    Returns:
        float: Bearing capacity of the bolt in kN.
    """
    kb = min(e / (3 * bolt_diameter), p / (3 * bolt_diameter) -
        0.25, fu_plate / 250, 1)
    bearing_strength = 2.5 * kb * bolt_diameter * plate_thickness
        * fu_plate
    bearing_capacity = bearing_strength / gamma_mb
    return bearing_capacity / 1000  # Convert to kN
```

```python
def calc_connection_capacity(number_of_bolts, bolt_shear_capacity
    , bolt_bearing_capacity):
    """
    Calculate the total capacity of the bolted connection.

    Parameters:
        number_of_bolts (int): Number of bolts in the connection.
        bolt_shear_capacity (float): Shear capacity of a single
            bolt (in kN).
        bolt_bearing_capacity (float): Bearing capacity of a
            single bolt (in kN).

    Returns:
        float: Total connection capacity in kN.
    """
    return number_of_bolts * min(bolt_shear_capacity,
        bolt_bearing_capacity)


def check_connection(load_applied, connection_capacity):
    """
    Verify if the connection is adequate for the applied load.

    Parameters:
        load_applied (float): Applied load on the connection (in
            kN).
        connection_capacity (float): Total capacity of the
            connection (in kN).

    Returns:
        str: Result of the check.
    """
    if connection_capacity >= load_applied:
        return "Connection is safe."
```

```
    else:
        return "Connection is unsafe."


# Input parameters
bolt_diameter = 16  # in mm
ultimate_tensile_strength_bolt = 400  # in MPa
gamma_mb_bolt = 1.25  # Partial safety factor for bolts

plate_thickness = 10  # in mm
ultimate_tensile_strength_plate = 410  # in MPa
edge_distance = 30  # in mm
pitch_distance = 50  # in mm

number_of_bolts = 4
applied_load = 120  # in kN

# Perform calculations
bolt_shear = calc_bolt_shear_capacity(bolt_diameter,
   ultimate_tensile_strength_bolt, gamma_mb_bolt)
bolt_bearing = calc_bearing_capacity(
    bolt_diameter, plate_thickness,
       ultimate_tensile_strength_plate,
    edge_distance, pitch_distance, gamma_mb_bolt
)
connection_capacity = calc_connection_capacity(number_of_bolts,
   bolt_shear, bolt_bearing)

# Output results
print(f"Shear capacity of a single bolt: {bolt_shear:.2f} kN")
print(f"Bearing capacity of a single bolt: {bolt_bearing:.2f} kN"
   )
print(f"Total connection capacity: {connection_capacity:.2f} kN")
print(f"Applied load: {applied_load:.2f} kN")
```

```
print(check_connection(applied_load, connection_capacity))
```

3.4 Code Explanation

Bolt Capacity Evaluation for Shear Connections

This script calculates the **shear capacity**, **bearing capacity**, and the **overall connection capacity** of a bolted connection used in steel structures (specifically in shear connections like end plates, fin plates, etc.). It also evaluates whether the connection is safe under a given applied load.

Assumptions

- The bolts are subjected to **shear and bearing** forces.
- The connection is designed as per the **Limit State Design**.
- Partial safety factor γ_{mb} is applied to both shear and bearing calculations.
- The lesser of shear and bearing capacity is taken as the governing capacity per bolt.

Function Explanations

1. `calc_bolt_shear_capacity(...)`

Purpose: Computes the **shear capacity** of a single bolt based on its diameter and the ultimate tensile strength of the bolt material.

```
def calc_bolt_shear_capacity(bolt_diameter, ultimate_tensile_strength, gamma_mb)
```

Parameters:

- `bolt_diameter` (in mm): Diameter of the bolt.
- `ultimate_tensile_strength` (in MPa): Tensile strength of the bolt material.
- `gamma_mb`: Partial safety factor for bolts (typically 1.25 for shear).

Formula Used:

$$V_{\text{nsb}} = \frac{0.6 \times \pi \times (d/2)^2 \times f_{ub}}{\gamma_{mb}} \quad \text{(in N)}$$

Converted to **kN** before returning.

2. `calc_bearing_capacity(...)`

Purpose: Calculates the **bearing capacity** of a single bolt based on edge distance, pitch distance, plate thickness, and material strength.

```
def calc_bearing_capacity(bolt_diameter, plate_thickness, fu_plate, e, p, gamma_mb)
```

Parameters:

- `bolt_diameter`: Diameter of the bolt (mm).
- `plate_thickness`: Thickness of the connected plate (mm).
- `fu_plate`: Ultimate tensile strength of the plate material (MPa).
- `e`: Edge distance from bolt to plate edge (mm).
- `p`: Pitch (distance between bolts in a row) (mm).
- `gamma_mb`: Partial safety factor for bearing.

Formula Used:

$$k_b = \min\left(\frac{e}{3d}, \frac{p}{3d} - 0.25, \frac{f_u}{250}, 1\right)$$

$$V_{\text{npb}} = \frac{2.5 \times k_b \times d \times t \times f_u}{\gamma_{mb}} \quad \text{(in N)}$$

Where:

- k_b is a factor accounting for geometry and material.
- Result is converted to **kN**.

3. `calc_connection_capacity(...)`

Purpose: Determines the **total capacity** of the bolted connection by multiplying the number of bolts with the **minimum** of shear or bearing capacity.

```
def calc_connection_capacity(number_of_bolts, bolt_shear_capacity, bolt_bearing_capac
```

Logic:

$$\text{Connection Capacity} = n \times \min(\text{Shear}, \text{Bearing}) \quad \text{(in kN)}$$

4. `check_connection(...)`

Purpose: Checks if the total calculated capacity is adequate to resist the applied load.

```
def check_connection(load_applied, connection_capacity)
```

Logic:

- If capacity $\geq$ load $\Rightarrow$ "Connection is safe."
- Else $\Rightarrow$ "Connection is unsafe."

Input Parameters Used

Parameter	Value	Unit	Description
`bolt_diameter`	16	mm	Diameter of bolt
`ultimate_tensile_strength_bolt`	400	MPa	Bolt material strength
`gamma_mb_bolt`	1.25	–	Safety factor
`plate_thickness`	10	mm	Plate thickness
`ultimate_tensile_strength_plate`	410	MPa	Plate material strength
`edge_distance (e)`	30	mm	Distance from edge to bolt center
`pitch_distance (p)`	50	mm	Bolt-to-bolt spacing
`number_of_bolts`	4	–	Number of bolts in the joint
`applied_load`	120	kN	External load applied on joint

Output (from script)

```
Shear capacity of a single bolt: 48.25 kN
Bearing capacity of a single bolt: 82.00 kN
Total connection capacity: 193.00 kN
Applied load: 120.00 kN
Connection is safe.
```

Conclusion

This script efficiently evaluates whether a given bolted connection is **safe under a specified load**. The conservative approach of using the minimum of shear and bearing capacities ensures safety and aligns with standard practices in structural steel design.

Chapter 4

Plate Girder

4.1 Plate Girder Model

I have been assigned to create Plate Girder CAD Model using Python OCC module. The model consists of an I section model, with stiffener plates inbetween the top and bottom flanges. To create this module, I have used CAD models such as, plate, filletweld from the *cad / items* directory. Along with these, I have used numpy package for efficient position storage in the run time memory rather than using python list.

The code actually generates each and every object individually and assembles them. To assemble then Python OCC uses **BRepAlgoAPIFuse** method/function, which is relatively computationally expensive and take lot of computing. Time estimated: 30.34s for generating 1000mm length plate girder model using **BRepAlgoAPIFuse**. To avoid excessive load time, I have used the **BRepBuilder()** function to assemble the model, with **BRepBuilder()** function, the estimated time is 1.08s for generating 1000mm length plate girder model.

4.2 Full Code

```
# optimised_plate_girder_refactored.py


from ISection import ISection
from .notch import Notch
from .plate import Plate
```

```
from .filletweld import FilletWeld
import math
import numpy as np
import time

from OCC.Core.gp import gp_Pnt, gp_Vec, gp_Trsf, gp_Ax1, gp_Dir,
   gp_Ax3
from OCC.Core.BRepBuilderAPI import (
    BRepBuilderAPI_MakePolygon, BRepBuilderAPI_MakeFace,
       BRepBuilderAPI_Transform,
    BRepBuilderAPI_MakeEdge, BRepBuilderAPI_MakeWire
)
from OCC.Core.BRepPrimAPI import BRepPrimAPI_MakePrism
from OCC.Core.BRepAlgoAPI import BRepAlgoAPI_Fuse
from OCC.Core.BRep import BRep_Builder
from OCC.Core.TopoDS import TopoDS_Compound
from OCC.Core.AIS import AIS_Shape
from OCC.Core.Quantity import Quantity_Color, Quantity_TOC_RGB
from OCC.Core.Graphic3d import Graphic3d_NOM_ALUMINIUM
from OCC.Display.SimpleGui import init_display


class PlateGirder:
    def __init__(self, D, tw, length, gap, T_ft, T_fb, B_ft, B_fb
       ):
        self.D = D
        self.tw = tw
        self.length = length
        self.gap = gap
        self.T_ft = T_ft
        self.T_fb = T_fb
        self.B_ft = B_ft
        self.B_fb = B_fb
```

```
    def createPlateGirder(self):
        chamfer_length = 30
        T_is = 15
        L = (min(self.B_ft, self.B_fb) - self.tw) / 2
        b = h = 15
        l = L - chamfer_length
        vertical_weld_height = 0.5 * chamfer_length

        center_plate_color = Quantity_Color(5/255, 5/255,
           255/255, Quantity_TOC_RGB)
        top_bottom_plate_color = Quantity_Color(137/255, 95/255,
           16/255, Quantity_TOC_RGB)

        display, start_display, *_ = init_display()
        display.set_bg_gradient_color([51, 51, 102], [150, 150,
           170])

        def plate_model_with_color(origin, l, b, h, color):
            plate = Plate(l, b, h)
            plate.place(origin, [0., 0., 1.], [0., 1., 0.])
            shape = plate.create_model()
            ais_shape = AIS_Shape(shape)
            ais_shape.SetColor(color)
            display.Context.Display(ais_shape, True)
            return shape

        def fuse_models(models):
            builder = BRep_Builder()
            compound = TopoDS_Compound()
            builder.MakeCompound(compound)
            for m in models:
                builder.Add(compound, m)
            return compound
```

```
        def translation_movement(x, y, z, model):
            trsf = gp_Trsf()
            trsf.SetTranslation(gp_Vec(x, y, z))
            return BRepBuilderAPI_Transform(model, trsf).Shape()

        def translation_rotation(angle, axis, model):
            trsf = gp_Trsf()
            trsf.SetRotation(axis, math.radians(angle))
            return BRepBuilderAPI_Transform(model, trsf).Shape()

        def stiffner_plate(position, b, a, thickness, direction):
            c = chamfer_length
            x, y, z = map(float, position)
            y -= thickness/2
            if direction == "right":
                pts = [gp_Pnt(0, 0, (a/2)-c), gp_Pnt(c, 0, a/2),
                   gp_Pnt(b, 0, a/2),
                       gp_Pnt(b, 0, -a/2), gp_Pnt(c, 0, -a/2),
                          gp_Pnt(0, 0, (-a/2)+c)]
            else:
                pts = [gp_Pnt(0, 0, (a/2)-c), gp_Pnt(-c, 0, a/2),
                    gp_Pnt(-b, 0, a/2),
                       gp_Pnt(-b, 0, -a/2), gp_Pnt(-c, 0, -a/2),
                          gp_Pnt(0, 0, (-a/2)+c)]

            wire = BRepBuilderAPI_MakeWire()
            for i in range(len(pts)):
                wire.Add(BRepBuilderAPI_MakeEdge(pts[i], pts[(i
                   +1) % len(pts)]).Edge())
            face = BRepBuilderAPI_MakeFace(wire.Wire()).Face()
            prism = BRepPrimAPI_MakePrism(face, gp_Vec(0,
               thickness, 0)).Shape()
```

```
            local_ax3 = gp_Ax3(gp_Pnt(0, 0, 0), gp_Dir(0, 0, 1),
                gp_Dir(0, 1, 0))
            global_ax3 = gp_Ax3(gp_Pnt(x, y, z), gp_Dir(0, 0, 1),
                 gp_Dir(0, 1, 0))
            trsf = gp_Trsf()
            trsf.SetDisplacement(local_ax3, global_ax3)
            return BRepBuilderAPI_Transform(prism, trsf, True).
                Shape()

        def vertical_weld(weld_height, length):
            p1, p2, p3 = gp_Pnt(0, 0, 0), gp_Pnt(weld_height, 0,
                0), gp_Pnt(0, -weld_height, 0)
            edges = [BRepBuilderAPI_MakeEdge(p1, p2).Edge(),
                BRepBuilderAPI_MakeEdge(p2, p3).Edge(),
                       BRepBuilderAPI_MakeEdge(p3, p1).Edge()]
            wire = BRepBuilderAPI_MakeWire(*edges).Wire()
            face = BRepBuilderAPI_MakeFace(wire).Face()
            return BRepPrimAPI_MakePrism(face, gp_Vec(0, 0, self.
                D - 2 * chamfer_length)).Shape()

        def create_weld_model(thickness, width, position,
            direction):
            uDir, shaftDir = ([0., 0., 1.], [0., 1., 0.]) if
                direction == 'y' else ([1., 0., 0.], [0., 0., 1.])
            FWeld = FilletWeld(thickness, thickness, width)
            FWeld.place(position, uDir, shaftDir)
            return FWeld.create_model(0)

        def filletWeld_model(b, h, l, y, position):
            x = self.tw//2 + chamfer_length if position == "right
                " else -self.tw//2 - l - chamfer_length
            FWeld = FilletWeld(b, h, l)
            FWeld.place([0., 0., 0.], [0., 0., 1.], [1., 0., 0.])
            base = FWeld.create_model(0)
```

```
        def rotate_and_translate(angle, dx, dy, dz):
            shape = translation_rotation(angle, gp_Ax1(gp_Pnt
                (0, 0, 0), gp_Dir(1, 0, 0)), base)
            return translation_movement(dx, dy, dz, shape)

        front = BRepAlgoAPI_Fuse(
            rotate_and_translate(0, x, y - T_is//2, self.D
                //2),
            rotate_and_translate(90, x, y - T_is//2, -self.D
                //2)
        ).Shape()

        back = BRepAlgoAPI_Fuse(
            rotate_and_translate(180, x, y + T_is//2, self.D
                //2),
            rotate_and_translate(270, x, y + T_is//2, -self.D
                //2)
        ).Shape()

        return BRepAlgoAPI_Fuse(front, back).Shape()

    # --- Begin assembling model ---
    center_plate = plate_model_with_color(np.array([0., 0.,
        0.]), self.tw, self.length, self.D, center_plate_color
        )
    top_plate = plate_model_with_color(np.array([0, 0, (self.
        D + self.T_ft) // 2]), self.B_ft, self.length, self.
        T_ft, top_bottom_plate_color)
    bottom_plate = plate_model_with_color(np.array([0, 0, -(
        self.D + self.T_fb) // 2]), self.B_fb, self.length,
        self.T_fb, top_bottom_plate_color)
```

```
        ISection_model = BRepAlgoAPI_Fuse(bottom_plate, top_plate
            ).Shape()

        # Longitudinal welds
        welds = [
            create_weld_model(0.5 * chamfer_length, self.length,
                np.array([self.tw // 2, 0., -self.D // 2]), "y"),
            create_weld_model(0.5 * chamfer_length, self.length,
                np.array([-self.tw // 2, 0., -self.D // 2]), "y"),
            create_weld_model(0.5 * chamfer_length, self.length,
                np.array([self.tw // 2, 0., self.D // 2]), "y"),
            create_weld_model(0.5 * chamfer_length, self.length,
                np.array([-self.tw // 2, 0., self.D // 2]), "y")
        ]
        longitudinal_weld = fuse_models(welds)

        stiffners, welds = [], []
        v_weld = vertical_weld(vertical_weld_height, self.D - 2 *
            chamfer_length)
        for y in range(self.gap, self.length, self.gap):
            stiffners.extend([
                stiffner_plate(np.array([self.tw/2, y, 0]), L,
                    self.D, T_is, "right"),
                stiffner_plate(np.array([-self.tw/2, y, 0]), L,
                    self.D, T_is, "left")
            ])
            welds.extend([
                filletWeld_model(b, h, l, y, "right"),
                filletWeld_model(b, h, l, y, "left"),
                translation_movement(self.tw/2, y, (-self.D/2)+
                    chamfer_length, v_weld),
                translation_movement(self.tw/2, y+T_is/2, (-self.
                    D/2)+chamfer_length,
```

```
                                translation_rotation(90,
                                    gp_Ax1(gp_Pnt(0, 0, 0),
                                    gp_Dir(0, 0, 1)), v_weld)
                                    ),
            translation_movement(-self.tw/2, y, (-self.D/2)+
                chamfer_length,
                                translation_rotation(-90,
                                    gp_Ax1(gp_Pnt(0, 0, 0),
                                    gp_Dir(0, 0, 1)), v_weld)
                                    ),
            translation_movement(-self.tw/2, y+T_is/2, (-self
                .D/2)+chamfer_length,
                                translation_rotation(-180,
                                    gp_Ax1(gp_Pnt(0, 0, 0),
                                    gp_Dir(0, 0, 1)), v_weld)
                                    )
        ])

    stiffners = fuse_models(stiffners)
    welds = fuse_models(welds)
    display.DisplayShape(ISection_model, update=True)
    display.DisplayShape(center_plate, update=True)
    display.DisplayShape(stiffners, material=
        Graphic3d_NOM_ALUMINIUM, update=True)
    display.DisplayShape(longitudinal_weld, color="red",
        update=True)
    display.DisplayShape(welds, color="red", update=True)
    start_display()

    plate_girder_model = BRepAlgoAPI_Fuse(BRepAlgoAPI_Fuse(
        center_plate, ISection_model).Shape(), stiffners)
    plate_girder_model = BRepAlgoAPI_Fuse(plate_girder_model,
        welds).Shape()
    return plate_girder_model
```

4.3 Code Explanation

Code Explanation: Plate Girder CAD Model Generation

This code implements a comprehensive CAD model generator for steel plate girders using the OpenCascade (OCC) geometry kernel. The `PlateGirder` class creates detailed 3D models including the main structural components, stiffeners, and welds.

Class Overview

`PlateGirder` Class Constructor

```
def __init__(self, D, tw, length, gap, T_ft, T_fb, B_ft, B_fb):
```

Parameters:

- `D`: Depth of the plate girder (mm)
- `tw`: Thickness of the web plate (mm)
- `length`: Length of the plate girder (mm)
- `gap`: Spacing between stiffeners (mm)
- `T_ft`: Thickness of the top flange (mm)
- `T_fb`: Thickness of the bottom flange (mm)
- `B_ft`: Width of the top flange (mm)
- `B_fb`: Width of the bottom flange (mm)

Core Functions

1. `createPlateGirder()` - Main Model Generation Function

This is the primary function that orchestrates the entire plate girder model creation process.

Key Variables:

- `chamfer_length = 30`
- `T_is = 15`
- `L = (min(B_ft, B_fb) - tw) / 2`
- `b = h = 15`
- `l = L - chamfer_length`
- `vertical_weld_height = 0.5 * chamfer_length`

Color Definitions:

- `center_plate_color`: Blue color for the web plate
- `top_bottom_plate_color`: Brown color for flange plates

2. `plate_model_with_color()` - Colored Plate Creation

```
def plate_model_with_color(origin, l, b, h, color):
```

Functionality:

- Creates a rectangular plate with specified dimensions
- Applies color to the plate for visual distinction
- Displays the plate in the 3D viewer
- Returns the geometric shape

Parameters:

- `origin`: Starting point coordinates [x, y, z]
- `l, b, h`: Length, breadth, and height of the plate
- `color`: RGB color value for the plate

3. `fuse_models()` - Model Combination

```
def fuse_models(models):
```

Functionality:

- Combines multiple geometric shapes into a single compound object
- Uses OpenCascade's `BRep_Builder` for efficient model fusion
- Essential for creating complex assemblies from individual components

4. `translation_movement()` - Spatial Translation

```
def translation_movement(x, y, z, model):
```

Functionality:

- Moves a geometric model by specified distances along x, y, z axes
- Uses OpenCascade's transformation system
- Returns the translated shape

5. `translation_rotation()` - Rotational Transformation

```
def translation_rotation(angle, axis, model):
```

Functionality:

- Rotates a geometric model around a specified axis
- Angle is converted from degrees to radians
- Returns the rotated shape

6. `stiffner_plate()` - Stiffener Plate Generation

```
def stiffner_plate(position, b, a, thickness, direction):
```

Functionality:

- Creates chamfered stiffener plates with specific geometry
- Generates different shapes for left and right stiffeners
- Includes chamfered edges for structural efficiency

Parameters:

- `position`: Location coordinates [x, y, z]
- `b, a`: Width and height of the stiffener
- `thickness`: Thickness of the stiffener plate
- `direction`: "left" or "right" to determine chamfer orientation

Geometry Details:

- Creates a 6-point polygon with chamfered corners
- Extrudes the 2D profile to create 3D solid
- Applies proper positioning and orientation

7. `vertical_weld()` - Vertical Weld Modeling

```
def vertical_weld(weld_height, length):
```

Functionality:

- Creates triangular cross-section welds
- Models the actual weld geometry used in fabrication
- Extrudes the triangular profile along the specified length

8. `create_weld_model()` - Fillet Weld Creation

```
def create_weld_model(thickness, width, position, direction):
```

Functionality:

- Creates fillet welds using the `FilletWeld` class
- Supports both horizontal and vertical orientations
- Positions welds accurately at connection points

9. `filletWeld_model()` - Complex Weld Assembly

```
def filletWeld_model(b, h, l, y, position):
```

Functionality:

- Creates complex weld assemblies for stiffener connections
- Combines multiple weld segments using boolean operations
- Handles both front and back weld patterns
- Applies proper rotations and translations for accurate positioning

Model Assembly Process

Step 1: Main Structural Components

```
center_plate = plate_model_with_color([0., 0., 0.], tw, length, D, center_plate_color
top_plate = plate_model_with_color([0, 0, (D + T_ft) // 2], B_ft, length, T_ft, top_b
bottom_plate = plate_model_with_color([0, 0, -(D + T_fb) // 2], B_fb, length, T_fb, t
```

Step 2: Longitudinal Welds

Creates four longitudinal welds connecting flanges to the web:

- Top flange to web (left and right)
- Bottom flange to web (left and right)

Step 3: Stiffeners and Welds

```
for y in range(gap, length, gap):
    # Create stiffeners at regular intervals
    # Add corresponding welds for each stiffener
```

Stiffener Pattern:

- Placed at regular intervals defined by `gap`
- Each stiffener includes:
  - Left and right stiffener plates
  - Four fillet welds per stiffener
  - Two vertical welds per stiffener

Step 4: Final Assembly

```
plate_girder_model = BRepAlgoAPI_Fuse(BRepAlgoAPI_Fuse(center_plate, ISection_model).
plate_girder_model = BRepAlgoAPI_Fuse(plate_girder_model, welds).Shape()
```

Technical Features

1. Geometric Accuracy

- Uses precise mathematical calculations for all dimensions
- Implements proper chamfering for structural efficiency
- Maintains accurate weld geometries

2. Visual Representation

- Color-coded components for easy identification
- Web plate: Blue
- Flange plates: Brown

- Welds: Red
- Stiffeners: Aluminum material appearance

3. Structural Realism

- Models actual fabrication details
- Includes weld geometries as used in practice
- Represents chamfered stiffener edges for structural efficiency

4. Modular Design

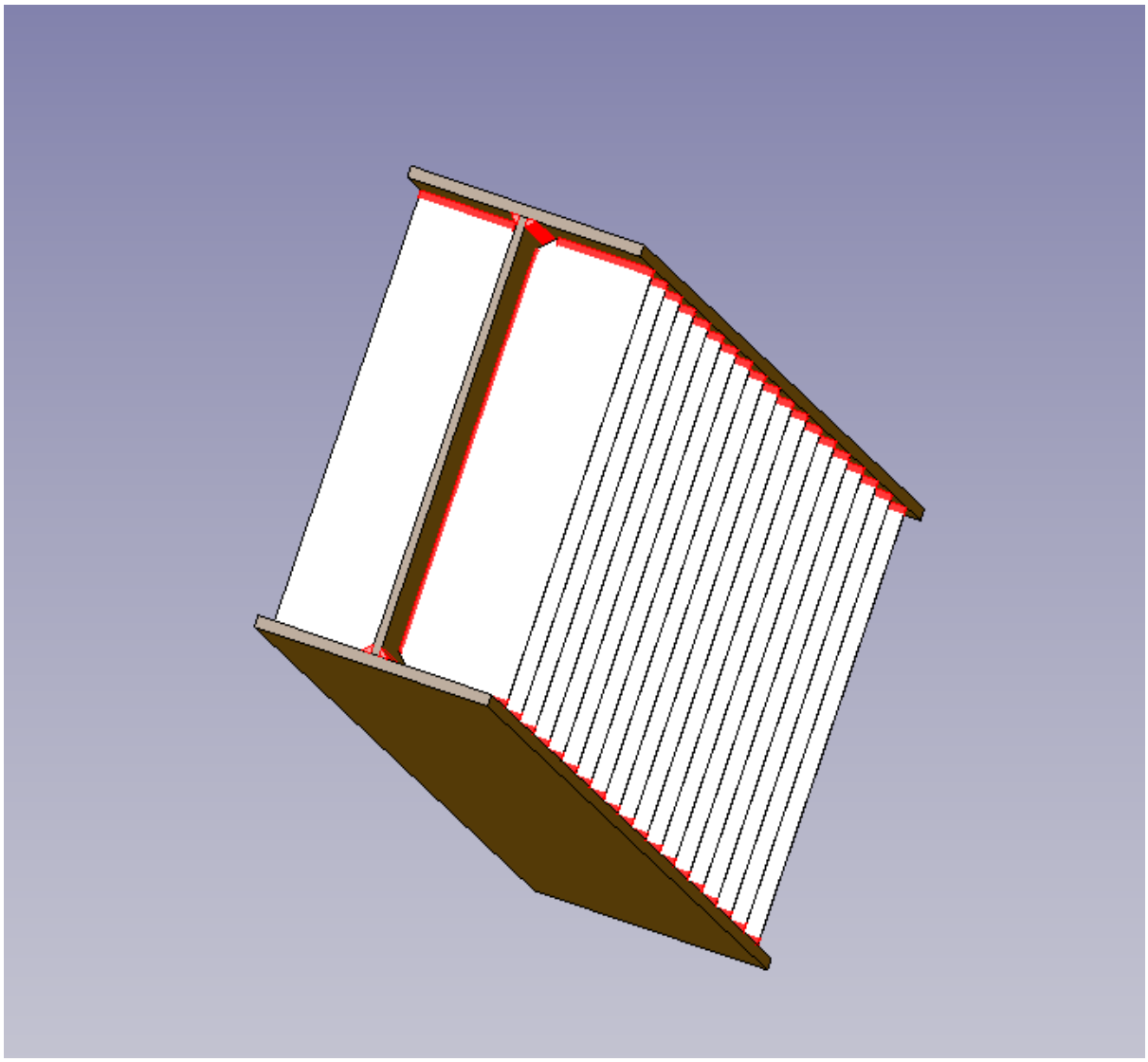
- Each component is created as a separate function
- Easy to modify individual elements
- Reusable code structure

Applications

This code is particularly useful for:

- **Structural Engineering**: Detailed plate girder design and analysis
- **Fabrication Planning**: Accurate 3D models for manufacturing
- **Educational Purposes**: Understanding plate girder construction
- **CAD Integration**: Export to other CAD systems for detailed drawings

4.4 CAD Model Image



Chapter 5

LCC Graphs

5.1 Horizontal Bar Graph

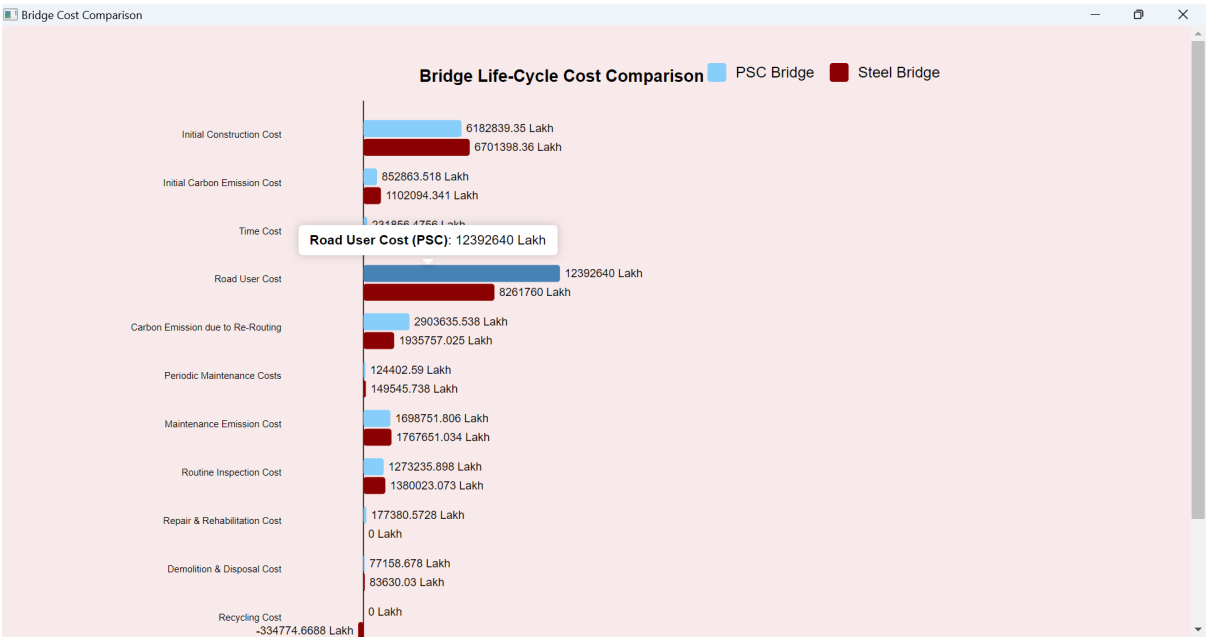
A horizontal bar graph is a graphical representation of categorical data with rectangular bars extending horizontally. The length of each bar corresponds to the magnitude of the variable.

This graph is implemented using the javascript library d3js. Which is an opensourced javascript library, useful in developing interactive graphs. The graph is laid on the foundation of HTML and CSS, and this entire combination of HTML, CSS, and Javascript is ran over python, which is supported by the python library, pyside6. Which is an interactive library in python to develop GUI for python applications.

Here the input values are used from an excel sheet(Book 4(Sheet1)1.csv) for testing purpose alone. The actual values are fetched from the database.

Use in LCC: Horizontal bar graphs are ideal for comparing cost components such as Initial construction cost, Embodied carbon emissions, Time cost estimate, Road user cost, etc. They offer a clear view when category names are long and need more horizontal space.

5.1.1 Image



5.1.2 Code

```
import sys
from PySide6.QtWidgets import QApplication, QMainWindow
from PySide6.QtWebEngineWidgets import QWebEngineView
import pandas as pd

FILE_PATH = r"Book 4(Sheet1)1.csv"
df = pd.read_csv(FILE_PATH)

list_final = [
    "Initial construction cost",
    "Embodied carbon emissions",
    "Time cost estimate",
    "Road user cost",
    "Additional CO2 e costs due to rerouting",
    "Periodic Maintenance costs",
    "Periodic maintenance carbon emissions",
    "Annual routine inspection costs",
```

```
    "Repair and rehabilitation costs",
    "Demolition and deconstruction costs",
    "Recycling costs"
]

# Extract values from DataFrame
psc_cost = []
steel_cost = []
for item in list_final:
    count = 0
    for _, row in df.iterrows():
        if item in list(row):
            temp_row = list(row)
            if count == 0:
                psc_cost.append(float(temp_row[temp_row.index(
                    item) + 1]))
            else:
                steel_cost.append(float(temp_row[temp_row.index(
                    item) + 1]))
            count += 1

# Calculate Total Life-Cycle Cost
total_psc_cost = sum(psc_cost)
total_steel_cost = sum(steel_cost)

# Prepare data for D3.js
js_data = []
for i, label in enumerate(list_final):
    # Adjust labels for better display in the chart
    display_label = label.replace("Initial construction cost", "
        Initial Construction Cost") \
                         .replace("Embodied carbon emissions", "
                            Initial Carbon Emission Cost") \
```

```
                            .replace("Time cost estimate", "Time
                               Cost") \
                            .replace("Road user cost", "Road User
                               Cost") \
                            .replace("Additional CO2 e costs due to
                               rerouting", "Carbon Emission due to
                               Re-Routing") \
                            .replace("Periodic Maintenance costs", "
                               Periodic Maintenance Costs") \
                            .replace("Periodic maintenance carbon
                               emissions", "Maintenance Emission
                               Cost") \
                            .replace("Annual routine inspection
                               costs", "Routine Inspection Cost") \
                            .replace("Repair and rehabilitation
                               costs", "Repair & Rehabilitation Cost
                               ") \
                            .replace("Demolition and deconstruction
                               costs", "Demolition & Disposal Cost")
                                \
                            .replace("Recycling costs", "Recycling
                               Cost")


    js_data.append({
        "label": display_label,
        "psc": psc_cost[i],
        "steel": steel_cost[i]
    })

# Add the "Total Life-Cycle Cost" entry
js_data.append({
    "label": "Total Life-Cycle Cost",
    "psc": total_psc_cost,
```

```
    "steel": total_steel_cost
})

with open(r"d3js.js", "r", encoding="utf-8") as f:
    d3_js = f.read()

# Convert Python list of dictionaries to a JavaScript array
    string
js_data_string = str(js_data).replace("'", '"') # Replace single
    quotes with double quotes for JSON compatibility

html_content = f"""
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Bridge Cost Comparison</title>
  <script>{d3_js}</script>
  <style>
    body {{
      font-family: Arial, sans-serif;
      background-color: #f8eaea;
      margin: 0;
    }}
    .tooltip {{
      position: absolute;
      background-color: #fff;
      padding: 8px 12px;
      border-radius: 5px;
      box-shadow: 0 0 10px rgba(0,0,0,0.2);
      font-size: 14px;
      pointer-events: none;
      opacity: 0;
      z-index: 10;
```

```
      transition: none;
    }}
    .tooltip::after {{
      content: "";
      position: absolute;
      bottom: -10px;
      left: 50%;
      transform: translateX(-50%);
      border-width: 6px 6px 0 6px;
      border-style: solid;
      border-color: #fff transparent transparent transparent;
      box-shadow: 0px 2px 5px rgba(0,0,0,0.1);
    }}
    .legend text {{
      font-size: 14px;
    }}
    .legend rect {{
      stroke: black;
      stroke-width: 1;
    }}
    #chart {{
      margin: 20px;
    }}
    .negative-value {{
      fill: white;
      font-size: 11px;
      font-weight: bold;
    }}
  </style>
</head>
<body>
<div id="chart"></div>
<div class="tooltip" id="tooltip"></div>
<script>
```

```
const data = {js_data_string};

const margin = {{top: 60, right: 130, bottom: 50, left: 280}};
const width = 1000 - margin.left - margin.right;
const barHeight = 18;
const categoryGap = 15;
const barGap = 2;
const height = data.length * (barHeight * 2 + barGap +
   categoryGap) + margin.top + margin.bottom;

const svg = d3.select("#chart")
  .append("svg")
  .attr("width", width + margin.left + margin.right)
  .attr("height", height)
  .append("g")
  .attr("transform", `translate(${{margin.left}},${{margin.top}})
     `); /* <-- Here is the fix */

// Find min and max values to properly handle negative numbers
const minValue = d3.min(data, d => Math.min(d.psc, d.steel));
const maxValue = d3.max(data, d => Math.max(d.psc, d.steel));

const x = d3.scaleLinear()
  .domain([minValue * 1.1, maxValue * 1.1])
  .nice()
  .range([0, width]);

const y = d3.scaleBand()
  .domain(data.map(d => d.label))
  .range([0, height - margin.top - margin.bottom])
  .padding(0.4);

// Add y-axis
svg.append("g")
```

```
  .call(d3.axisLeft(y).tickSize(0))
  .selectAll(".domain").remove();

// Add x-axis with zero line
svg.append("g")
  .attr("transform", `translate(0, ${{height - margin.top -
     margin.bottom}})`) /* <-- Here is the fix */
  .call(d3.axisBottom(x).ticks(10).tickFormat(d => d + " Lakh"));

// Add vertical zero line
svg.append("line")
  .attr("x1", x(0))
  .attr("x2", x(0))
  .attr("y1", 0)
  .attr("y2", height - margin.top - margin.bottom)
  .attr("stroke", "#000")
  .attr("stroke-width", 1);

const tooltip = d3.select("#tooltip");

function showTooltip(event, label, value, type) {{
  tooltip
    .html(`<strong>${{label}} (${{type === 'psc' ? 'PSC' : 'Steel
       '}})</strong>: ${{value}} Lakh`)
    .style("left", (event.pageX - 150) + "px")
    .style("top", (event.pageY - 50) + "px")
    .style("opacity", 1);
}}

function hideTooltip() {{
  tooltip.style("opacity", 0);
}}

const colors = {{
```

```
  psc: "#87cefa",
  steel: "#8b0000"
}};

// Calculate y position for each bar pair
function getBarYPosition(d, barIndex) {{
  return y(d.label) + barIndex * (barHeight + barGap);
}}

// Draw bars for PSC Bridge with animation
svg.selectAll(".bar-psc")
  .data(data)
  .enter()
  .append("rect")
  .attr("class", "bar-psc")
  .attr("x", x(0)) // Start at zero line
  .attr("y", d => getBarYPosition(d, 0))
  .attr("width", 0) // Start with zero width
  .attr("height", barHeight)
  .attr("fill", colors.psc)
  .attr("rx", 3)
  .attr("ry", 3)
  .on("mousemove", function(event, d) {{
    showTooltip(event, d.label, d.psc, 'psc');
    d3.select(this).attr("fill", "#4682b4");
  }})
  .on("mouseleave", function() {{
    hideTooltip();
    d3.select(this).attr("fill", colors.psc);
  }})
  .transition() // Add transition for animation
  .duration(1000) // Animation duration in milliseconds
  .delay((d, i) => i * 100) // Staggered delay for each bar
  .attr("x", d => x(Math.min(0, d.psc)))
```

```
  .attr("width", d => Math.abs(x(d.psc) - x(0)));

// Draw bars for Steel Bridge with animation
svg.selectAll(".bar-steel")
  .data(data)
  .enter()
  .append("rect")
  .attr("class", "bar-steel")
  .attr("x", x(0)) // Start at zero line
  .attr("y", d => getBarYPosition(d, 1))
  .attr("width", 0) // Start with zero width
  .attr("height", barHeight)
  .attr("fill", colors.steel)
  .attr("rx", 3)
  .attr("ry", 3)
  .on("mousemove", function(event, d) {{
    showTooltip(event, d.label, d.steel, 'steel');
    d3.select(this).attr("fill", "#a52a2a");
  }})
  .on("mouseleave", function() {{
    hideTooltip();
    d3.select(this).attr("fill", colors.steel);
  }})
  .transition() // Add transition for animation
  .duration(1000) // Animation duration in milliseconds
  .delay((d, i) => i * 100) // Staggered delay for each bar
  .attr("x", d => x(Math.min(0, d.steel)))
  .attr("width", d => Math.abs(x(d.steel) - x(0)));

// Add value labels for PSC Bridge (right of bar)
svg.selectAll(".label-psc")
  .data(data)
  .enter()
  .append("text")
```

```
  .attr("class", "label-psc")
  .attr("x", d => d.psc >= 0 ? x(d.psc) + 5 : x(d.psc) - 5)
  .attr("y", d => getBarYPosition(d, 0) + barHeight / 2 + 4)
  .text(d => d.psc + " Lakh")
  .attr("font-size", "12px")
  .attr("fill", d => d.psc < 0 ? "black" : "black")
  .attr("text-anchor", d => d.psc >= 0 ? "start" : "end");

// Add value labels for Steel Bridge (right of bar)
svg.selectAll(".label-steel")
  .data(data)
  .enter()
  .append("text")
  .attr("class", "label-steel")
  .attr("x", d => d.steel >= 0 ? x(d.steel) + 5 : x(d.steel) - 5)
  .attr("y", d => getBarYPosition(d, 1) + barHeight / 2 + 4)
  .text(d => d.steel + " Lakh")
  .attr("font-size", "12px")
  .attr("fill", d => d.steel < 0 ? "black" : "black")
  .attr("text-anchor", d => d.steel >= 0 ? "start" : "end");

// Legend
const legend = svg.append("g")
  .attr("transform", `translate(${{width - 140}}, -40)`); /* <--
    Here is the fix */

legend.append("rect")
  .attr("x", 0)
  .attr("width", 20)
  .attr("height", 20)
  .attr("fill", colors.psc)
  .attr("rx", 3)
  .attr("ry", 3);
```

```
legend.append("text")
  .attr("x", 30)
  .attr("y", 15)
  .text("PSC Bridge");

legend.append("rect")
  .attr("x", 130)
  .attr("width", 20)
  .attr("height", 20)
  .attr("fill", colors.steel)
  .attr("rx", 3)
  .attr("ry", 3);

legend.append("text")
  .attr("x", 160)
  .attr("y", 15)
  .text("Steel Bridge");

// Add title
svg.append("text")
  .attr("x", width / 2)
  .attr("y", -20)
  .attr("text-anchor", "middle")
  .style("font-size", "18px")
  .style("font-weight", "bold")
  .text("Bridge Life-Cycle Cost Comparison");
</script>
</body>
</html>
"""

class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()
```

```
        self.setWindowTitle("Bridge Cost Comparison")
        self.setGeometry(100, 100, 1200, 900)

        view = QWebEngineView()
        view.setHtml(html_content)
        self.setCentralWidget(view)

if __name__ == "__main__":
    app = QApplication(sys.argv)
    window = MainWindow()
    window.show()
    sys.exit(app.exec())
```

5.1.3 Code Explanation

Overview

This Python application creates an interactive horizontal bar chart comparing the life-cycle costs between two bridge construction types: **PSC (Prestressed Concrete)** and **Steel** bridges. The visualization is built using PySide6 (Qt) for the GUI framework and D3.js for the interactive chart rendering.

Key Components and Functions

1. Data Import and Processing

```
FILE_PATH = r"Book 4(Sheet1)1.csv"
df = pd.read_csv(FILE_PATH)
```

Purpose: Imports cost data from a CSV file containing life-cycle cost analysis for both bridge types

Data Structure: The CSV contains cost components with corresponding monetary values for PSC and Steel bridges

2. Cost Component Categories

```
list_final = [
    "Initial construction cost",
    "Embodied carbon emissions",
    "Time cost estimate",
    "Road user cost",
    "Additional CO2 e costs due to rerouting",
    "Periodic Maintenance costs",
    "Periodic maintenance carbon emissions",
    "Annual routine inspection costs",
    "Repair and rehabilitation costs",
    "Demolition and deconstruction costs",
    "Recycling costs"
]
```

Purpose: Defines the 11 key cost components analyzed in the life-cycle cost assessment

Categories: Covers initial construction, environmental impacts, maintenance, and end-of-life costs

3. Data Extraction Function

```
psc_cost = []
steel_cost = []
for item in list_final:
    count = 0
    for _, row in df.iterrows():
        if item in list(row):
            temp_row = list(row)
            if count == 0:
                psc_cost.append(float(temp_row[temp_row.index(item) + 1]))
            else:
                steel_cost.append(float(temp_row[temp_row.index(item) + 1]))
            count += 1
```

Purpose: Extracts cost values for each component from the CSV data

Logic: Iterates through the DataFrame to find matching cost components and extracts the corresponding monetary values

Output: Two lists containing PSC and Steel bridge costs respectively

4. Total Cost Calculation

```
total_psc_cost = sum(psc_cost)
total_steel_cost = sum(steel_cost)
```

Purpose: Calculates the total life-cycle cost for each bridge type

Usage: These totals are added as the final row in the visualization

5. Data Preparation for Visualization

```
js_data = []
for i, label in enumerate(list_final):
    display_label = label.replace("Initial construction cost", "Initial Construction
                          .replace("Embodied carbon emissions", "Initial Carbon Emissi
                          # ... additional label formatting
    js_data.append({
        "label": display_label,
        "psc": psc_cost[i],
        "steel": steel_cost[i]
    })
```

Purpose: Formats data for D3.js visualization

Function:

- Creates user-friendly labels for display
- Structures data as JavaScript objects with label, PSC cost, and Steel cost
- Adds the total life-cycle cost as the final entry

6. HTML Generation with D3.js Integration

The code generates a complete HTML document containing:

- **D3.js Library**: Loaded from external file (`d3js.js`)
- **CSS Styling**: Custom styling for tooltips, legend, and chart appearance
- **Interactive Features**: Hover effects, tooltips, and animations

7. D3.js Visualization Components

a) Chart Setup:

```
const margin = {top: 60, right: 130, bottom: 50, left: 280};
const width = 1000 - margin.left - margin.right;
const barHeight = 18;
const categoryGap = 15;
const barGap = 2;
```

Purpose: Defines chart dimensions and spacing

Variables:

- `margin`: Chart padding from edges
- `barHeight`: Height of individual bars
- `categoryGap`: Space between different cost categories
- `barGap`: Space between PSC and Steel bars for same category

b) Scales:

```
const x = d3.scaleLinear()
  .domain([minValue * 1.1, maxValue * 1.1])
  .nice()
  .range([0, width]);

const y = d3.scaleBand()
  .domain(data.map(d => d.label))
```

```
  .range([0, height - margin.top - margin.bottom])
  .padding(0.4);
```

Purpose: Maps data values to visual coordinates

X-scale: Linear scale for cost values (handles negative values)

Y-scale: Band scale for cost component labels

c) Interactive Features:

- **Tooltips**: Display exact values on hover
- **Color Changes**: Bars change color on hover for better UX
- **Animations**: Bars animate from zero width to full width with staggered delays

8. GUI Application Class

```
class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        self.setWindowTitle("Bridge Cost Comparison")
        self.setGeometry(100, 100, 1200, 900)

        view = QWebEngineView()
        view.setHtml(html_content)
        self.setCentralWidget(view)
```

Purpose: Creates the desktop application window

Components:

- `QMainWindow`: Main application window
- `QWebEngineView`: Web view component to render HTML/D3.js content
- Window dimensions: 1200x900 pixels

Key Features

1. **Comparative Analysis**: Side-by-side comparison of PSC vs Steel bridge costs

2. **Comprehensive Coverage**: All 11 life-cycle cost components included
3. **Interactive Visualization**: Hover effects, tooltips, and smooth animations
4. **Negative Value Support**: Properly handles negative costs (e.g., recycling benefits)
5. **Professional Styling**: Clean, modern design with proper legends and labels
6. **Responsive Design**: Adapts to different screen sizes

Data Flow

CSV Input → Data Extraction → Cost Calculation → Data Formatting → HTML Generation → D3.js Rendering → Interactive Chart

Technical Stack

- **Backend**: Python with Pandas for data processing
- **Frontend**: D3.js for interactive visualization
- **GUI Framework**: PySide6 (Qt for Python)
- **Data Format**: CSV with structured cost component data

5.2 Pie Chart

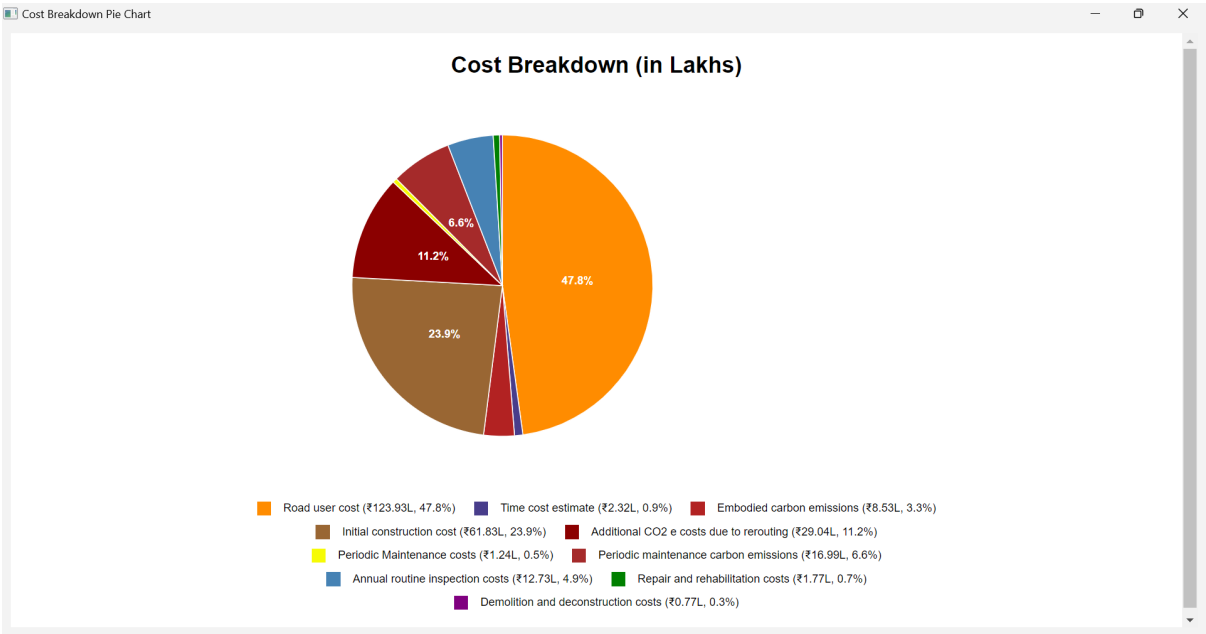
A pie chart is a circular statistical graphic that is divided into slices to illustrate numerical proportions. Each slice represents a category's contribution to the whole.

This graph is implemented using the javascript library d3js. Which is an opensourced javascript library, useful in developing interactive graphs. The graph is laid on the foundation of HTML and CSS, and this entire combination of HTML, CSS, and Javascript is ran over python, which is supported by the python library, pyside6. Which is an interactive library in python to develop GUI for python applications.

Here the input values are used from an excel sheet(Book 4(Sheet1)1.csv) for testing purpose alone. The actual values are fetched from the database.

Use in LCC: Pie charts help visualize the proportion of different cost elements within the total lifecycle cost. They make it easy to understand the dominant contributors (e.g., Initial construction cost, Embodied carbon emissions, Time cost estimate, Road user cost, etc.).

5.2.1 Image



5.2.2 Code

```
import pandas as pd
import json
from PySide6.QtWidgets import QApplication, QMainWindow, QWidget,
    QVBoxLayout
from PySide6.QtWebEngineWidgets import QWebEngineView
from PySide6.QtCore import QUrl, QSize

class D3PieChartViewer(QMainWindow):
    def __init__(self, data_js):
        super().__init__()
        self.setWindowTitle("Cost Breakdown Pie Chart")
        self.setMinimumSize(QSize(800, 600))
```

```
        self.central_widget = QWidget()
        self.setCentralWidget(self.central_widget)

        self.layout = QVBoxLayout(self.central_widget)

        self.web_view = QWebEngineView()
        self.layout.addWidget(self.web_view)

        # Generate HTML content
        html_content = self.generate_html(data_js)

        # Load the HTML content
        self.web_view.setHtml(html_content, QUrl.fromLocalFile(""
            ))

    def generate_html(self, data_js):
        html = f"""
        <!DOCTYPE html>
        <html>
        <head>
            <title>Cost Breakdown Pie Chart</title>
            <script src="https://d3js.org/d3.v7.min.js"></script>
            <style>
                body {{
                    font-family: Arial, sans-serif;
                    margin: 20px;
                }}
                .chart-container {{
                    width: 100%;
                    max-width: 800px;
                    margin: 0 auto;
                }}
                .pie-label {{
```

```
            font-size: 12px;
            fill: #333;
        }}
        .legend-item {{
            font-size: 12px;
            display: flex;
            align-items: center;
            margin: 5px 10px;
            cursor: pointer;
        }}
        .title {{
            text-align: center;
            margin-bottom: 20px;
        }}
        .legend {{
            display: flex;
            flex-wrap: wrap;
            justify-content: center;
            margin-top: 20px;
        }}
        .legend-color {{
            width: 15px;
            height: 15px;
            margin-right: 8px;
            display: inline-block;
        }}
        .strikethrough {{
            position: relative;
            color: #999;
        }}
        .strikethrough::after {{
            content: "";
            position: absolute;
            left: 0;
```

```
                top: 50%;
                width: 100%;
                height: 1px;
                background: black;
            }}
            .tooltip {{
                position: absolute;
                background: rgba(0, 0, 0, 0.8);
                color: white;
                padding: 8px;
                border-radius: 4px;
                font-size: 12px;
                pointer-events: none;
                opacity: 0;
                transition: opacity 0.2s ease;
            }}
            .tooltip.visible {{
                opacity: 1;
            }}
        </style>
    </head>
    <body>
        <div class="chart-container">
            <h2 class="title">Cost Breakdown (in Lakhs)</h2>
            <div id="chart"></div>
            <div id="legend" class="legend"></div>
        </div>

        <script>
            // Data from Python with specific colors
            const originalData = {data_js};
            let currentData = JSON.parse(JSON.stringify(
               originalData));
```

```
// Chart dimensions
const width = 600;
const height = 400;
const radius = Math.min(width, height) / 2 - 40;
const enlargedRadius = radius * 1.1;

// Create SVG
const svg = d3.select("#chart")
    .append("svg")
    .attr("width", width)
    .attr("height", height)
    .append("g")
    .attr("transform", `translate(${{width/2}},$
       {{height/2}})`);

// Create tooltip
const tooltip = d3.select("body")
    .append("div")
    .attr("class", "tooltip");

// Create pie layout
const pie = d3.pie()
    .value(d => d.cost)
    .sort(null);

// Create arc generator
const arc = d3.arc()
    .innerRadius(0)
    .outerRadius(radius);

// Create arc generator for enlarged slices
const enlargedArc = d3.arc()
    .innerRadius(0)
    .outerRadius(enlargedRadius);
```

```
        // Arc tween for smooth transitions
        function arcTween(d, i) {{
            const interpolate = d3.interpolate(
                this._current || {{ startAngle: d.
                    startAngle, endAngle: d.startAngle }},
                d
            );
            this._current = interpolate(1);
            return t => arc(interpolate(t));
        }}

        // Arc tween for exit animation
        function arcTweenExit(d) {{
            const interpolate = d3.interpolate(d, {{
                startAngle: d.endAngle,
                endAngle: d.endAngle
            }});
            return t => arc(interpolate(t));
        }}

        // Reset hover state
        function resetHoverState() {{
            svg.selectAll(".arc")
                .transition()
                .duration(200)
                .style("opacity", 1)
                .attr("fill", arc => arc.data.color)
                .attr("stroke-width", 1)
                .attr("d", arc);
            svg.selectAll(".pie-label")
                .transition()
                .duration(200)
```

```
                .attr("transform", d => `translate(${{arc
                    .centroid(d)}})`);
            tooltip.classed("visible", false);
        }}

        // Function to update the pie chart
        function updatePieChart() {{
            // Filter out disabled items
            const activeData = currentData.filter(d => !d
                .disabled);

            // Calculate new percentages
            const totalActiveCost = activeData.reduce((
                sum, d) => sum + d.cost, 0);
            activeData.forEach(d => {{
                d.percent = (d.cost / totalActiveCost) *
                    100;
            }});

            // Update pie with new data
            const arcs = pie(activeData);

            // Join new data with existing paths
            const paths = svg.selectAll(".arc")
                .data(arcs, d => d.data.label);

            // Handle exiting slices
            paths.exit()
                .transition()
                .duration(400)
                .ease(d3.easeCubicInOut)
                .attrTween("d", arcTweenExit)
                .style("opacity", 0)
                .remove();
```

```
        // Update existing slices
        paths.transition()
            .duration(600)
            .ease(d3.easeCubicInOut)
            .attrTween("d", arcTween)
            .attr("fill", d => d.data.color);

        // Add new slices
        paths.enter()
            .append("path")
            .attr("class", "arc")
            .attr("fill", d => d.data.color)
            .attr("stroke", "#fff")
            .attr("stroke-width", 1)
            .style("opacity", 0)
            .each(function(d) {{ this._current = d;
               }})
            .transition()
            .duration(600)
            .ease(d3.easeCubicInOut)
            .attrTween("d", arcTween)
            .style("opacity", 1)
            .on("end", function() {{
                // Add hover events after animation
                d3.select(this)
                    .on("mouseover", handleMouseOver)
                    .on("mousemove", handleMouseMove)
                    .on("mouseout", handleMouseOut);
            }});

        // Update labels
        const labels = svg.selectAll(".pie-label")
            .data(arcs, d => d.data.label);
```

```
        labels.exit()
            .transition()
            .duration(200)
            .style("opacity", 0)
            .remove();

        labels.enter()
            .append("text")
            .attr("class", "pie-label")
            .attr("dy", "0.35em")
            .attr("text-anchor", "middle")
            .style("font-weight", "bold")
            .style("fill", "#fff")
            .style("opacity", 0)
            .merge(labels)
            .transition()
            .duration(400)
            .ease(d3.easeCubicInOut)
            .attr("transform", d => `translate(${{arc
                .centroid(d)}})`)
            .text(d => d.data.percent > 5 ? `${{d.
                data.percent.toFixed(1)}}%` : "")
            .style("opacity", 1);

        // Update legend percentages
        d3.selectAll(".legend-text")
            .text(d => `${{d.label}} ( $ {{d.cost.
                toFixed(2)}}L, ${{d.percent.toFixed(1)
                }}%)`);
    }}

    // Handle mouseover for slices
    function handleMouseOver(event, d) {{
```

```
                    // Enlarge and highlight the hovered slice
                    d3.select(this)
                        .transition()
                        .duration(200)
                        .attr("stroke-width", 2)
                        .attr("fill", d3.color(d.data.color).
                            brighter(0.5))
                        .attr("d", enlargedArc);

                    // Update label position for enlarged slice
                    svg.selectAll(".pie-label")
                        .filter(label => label.data.label === d.
                            data.label)
                        .transition()
                        .duration(200)
                        .attr("transform", `translate(${{
                            enlargedArc.centroid(d)}})`);

                    // Grey out all other slices
                    svg.selectAll(".arc")
                        .filter(arc => arc.data.label !== d.data.
                            label)
                        .transition()
                        .duration(200)
                        .style("opacity", 0.3)
                        .attr("fill", "#cccccc");

                    // Show tooltip
                    tooltip
                        .html(`
                            <strong>${{d.data.label}}</strong><br
                                >
                            Cost:   $  {{d.data.cost.toFixed(2)}}L
                                <br>
```

```
                    Percent: ${{d.data.percent.toFixed(1)
                        }}%
                `)
                .style("left", (event.pageX + 10) + "px")
                .style("top", (event.pageY - 10) + "px")
                .classed("visible", true);
        }}

        // Handle mousemove for slices
        function handleMouseMove(event, d) {{
            tooltip
                .style("left", (event.pageX + 10) + "px")
                .style("top", (event.pageY - 10) + "px");
        }}

        // Handle mouseout for slices
        function handleMouseOut(event, d) {{
            resetHoverState();
        }}

        // Initialize the chart
        function initializeChart() {{
            // Calculate initial percentages
            const totalCost = currentData.reduce((sum, d)
                => sum + d.cost, 0);
            currentData.forEach(d => {{
                d.percent = (d.cost / totalCost) * 100;
            }});

            const arcs = pie(currentData);

            // Create initial arcs
            svg.selectAll(".arc")
                .data(arcs)
```

```
                .enter()
                .append("path")
                .attr("class", "arc")
                .attr("fill", d => d.data.color)
                .attr("stroke", "#fff")
                .attr("stroke-width", 1)
                .style("opacity", 0)
                .each(function(d) {{
                    this._current = {{
                        startAngle: d.startAngle,
                        endAngle: d.startAngle
                    }};
                }})
                .transition()
                .duration(800)
                .ease(d3.easeCubicInOut)
                .attrTween("d", arcTween)
                .style("opacity", 1)
                .on("end", function() {{
                    // Add hover events after animation
                    d3.select(this)
                        .on("mouseover", handleMouseOver)
                        .on("mousemove", handleMouseMove)
                        .on("mouseout", handleMouseOut);
                }});

            // Create initial labels
            svg.selectAll(".pie-label")
                .data(arcs)
                .enter()
                .append("text")
                .attr("class", "pie-label")
                .attr("dy", "0.35em")
                .attr("text-anchor", "middle")
```

```
            .style("font-weight", "bold")
            .style("fill", "#fff")
            .style("opacity", 0)
            .attr("transform", d => `translate(${{arc
                .centroid(d)}})`)
            .text(d => d.data.percent > 5 ? `${{d.
                data.percent.toFixed(1)}}%` : "")
            .transition()
            .duration(400)
            .delay(400)
            .ease(d3.easeCubicInOut)
            .style("opacity", 1);

        // Create legend
        const legend = d3.select("#legend")
            .selectAll(".legend-item")
            .data(currentData)
            .enter()
            .append("div")
            .attr("class", "legend-item")
            .attr("id", d => `legend-item-${{d.label.
                replace(/\s+/g, '-')}}`);

        legend.append("div")
            .attr("class", "legend-color")
            .style("background-color", d => d.color);

        legend.append("span")
            .text(d => `${{d.label}} ( $ {{d.cost.
                toFixed(2)}}L, ${{d.percent.toFixed(1)
                }}%)`)
            .style("margin-left", "5px")
            .attr("class", "legend-text");
```

```
                    // Add interactivity
                    legend.on("click", function(event, d) {{
                        // Reset hover state before toggling
                        resetHoverState();

                        // Toggle the item
                        toggleItem(d);

                        // Prevent event propagation
                        event.stopPropagation();
                    }});

                    // Add hover effects for legend
                    legend.on("mouseover", function(event, d) {{
                        if (!d.disabled) {{
                            // Highlight the corresponding slice
                            svg.selectAll(".arc")
                                .filter(arc => arc.data.label ===
                                    d.label)
                                .attr("stroke-width", 2)
                                .attr("fill", d3.color(d.color).
                                   brighter(0.5));

                            // Get the centroid of the slice
                            const arcData = svg.selectAll(".arc")
                                .filter(arc => arc.data.label ===
                                    d.label)
                                .data()[0];

                            if (arcData) {{
                                // Calculate the centroid
                                   position
                                const centroid = arc.centroid(
                                   arcData);
```

```
                    // Get the SVG's position on the
                       page
                    const svgRect = svg.node().
                       parentNode.
                       getBoundingClientRect();

                    // Calculate the absolute
                       position of the centroid
                    const centroidX = svgRect.left +
                       width/2 + centroid[0];
                    const centroidY = svgRect.top +
                       height/2 + centroid[1];

                    // Show tooltip at the centroid
                       position
                    tooltip
                        .html(`
                            <strong>${{d.label}}</
                               strong><br>
                            Cost:  $ {{d.data.cost.
                               toFixed(2)}}L<br>
                            Percent: ${{d.data.
                               percent.toFixed(1)}}%
                        `)
                        .style("left", (centroidX +
                           10) + "px")
                        .style("top", (centroidY -
                           30) + "px")
                        .classed("visible", true);
                }}
            }}
        }})
        .on("mouseout", function(event, d) {{
```

```
                if (!d.disabled) {{
                    // Reset highlight on the slice
                    svg.selectAll(".arc")
                        .filter(arc => arc.data.label ===
                            d.label)
                        .attr("stroke-width", 1)
                        .attr("fill", arc => arc.data.
                           color);

                    // Hide tooltip
                    tooltip.classed("visible", false);
                }}
            }});

            // Add click handler to SVG to reset hover
               state
            svg.on("click", function(event) {{
                resetHoverState();
            }});
        }}

        // Function to toggle item state
        function toggleItem(item) {{
            const index = currentData.findIndex(d => d.
               label === item.label);
            if (index !== -1) {{
                currentData[index].disabled = !
                   currentData[index].disabled;

                const legendItem = d3.select(`#legend-
                   item-${{item.label.replace(/\s+/g,
                   '-')}}`);
                if (currentData[index].disabled) {{
```

```
                            legendItem.select(".legend-text").
                                classed("strikethrough", true);
                        }} else {{
                            legendItem.select(".legend-text").
                                classed("strikethrough", false);
                        }}

                        updatePieChart();
                    }}
                }}

                // Start the chart
                initializeChart();
            </script>
        </body>
        </html>
        """
        return html

def main():
    print("generating graph")
    FILE_PATH = r"Book 4(Sheet1)1.csv"

    # --- Read CSV and Extract Data ---
    try:
        df = pd.read_csv(FILE_PATH)
        if df.empty:
            print(f"Warning: CSV file '{FILE_PATH}' is empty.")
    except FileNotFoundError:
        print(f"Error: CSV file not found at '{FILE_PATH}'.")
        df = pd.DataFrame()
    except Exception as e:
        print(f"Error reading CSV: {e}")
        df = pd.DataFrame()
```

```
# Define labels and their specific colors
label_colors = {
    "Road user cost": "#FF8C00",
    "Time cost estimate": "#483D8B",
    "Embodied carbon emissions": "#B22222",
    "Initial construction cost": "#996633",
    "Additional CO2 e costs due to rerouting": "#8B0000",
    "Periodic Maintenance costs": "#F6FB05",
    "Periodic maintenance carbon emissions": "#A52A2A",
    "Annual routine inspection costs": "#4682B4",
    "Repair and rehabilitation costs": "#008000",
    "Demolition and deconstruction costs": "#800080"
}

# Extract data
values_dict = {}
for item in label_colors.keys():
    found_value = False
    if not df.empty:
        for _, row in df.iterrows():
            row_list = [str(x) for x in row.values]
            if item in row_list:
                idx = row_list.index(item)
                if idx + 1 < len(row_list):
                    values_dict[item] = row_list[idx + 1]
                    found_value = True
                    break
    if not found_value:
        values_dict[item] = "0.0"

cost_list = []
for key in label_colors.keys():
    try:
```

```
            cost_list.append(float(values_dict.get(key, "0.0")))
        except:
            cost_list.append(0.0)

    total_cost = sum(cost_list)
    percentage_list = [(v / total_cost) * 100 if total_cost else
       0 for v in cost_list]
    cost_list_lakhs = [v / 100000 for v in cost_list]

    # Create data with specific color mapping
    data_with_colors = [
        {
            "label": name,
            "cost": cost,
            "percent": percent,
            "color": label_colors[name],
            "disabled": False
        }
        for name, cost, percent in zip(label_colors.keys(),
           cost_list_lakhs, percentage_list)
    ]

    data_js = json.dumps(data_with_colors)

    # Create and show the application window
    app = QApplication([])
    viewer = D3PieChartViewer(data_js)
    viewer.show()
    app.exec()

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

5.2.3 Code Explanation

Overview

This Python application creates an interactive pie chart visualization for cost breakdown analysis using D3.js (Data-Driven Documents) library embedded within a PySide6 Qt application. The chart displays various infrastructure costs and allows users to interactively explore the data through hover effects, tooltips, and legend-based filtering.

Architecture

Technology Stack

- **Backend**: Python with PySide6 (Qt framework)
- **Frontend**: HTML/CSS/JavaScript with D3.js v7
- **Data Processing**: Pandas for CSV data handling
- **Data Format**: JSON for data transfer between Python and JavaScript

Key Components

1. Main Application Class: `D3PieChartViewer`

`class D3PieChartViewer(QMainWindow):`

Purpose: Creates a desktop application window that hosts the interactive pie chart.
Key Attributes:

- `central_widget`: Main container widget
- `layout`: Vertical layout manager (QVBoxLayout)
- `web_view`: QWebEngineView component that renders the HTML/JavaScript content

Methods:

- `__init__(data_js)`: Initializes the application window and loads the chart
- `generate_html(data_js)`: Creates the complete HTML document with embedded D3.js visualization

2. Data Processing Functions

`main()` Function

Purpose: Orchestrates the entire application flow from data extraction to visualization.

Key Variables:

- `FILE_PATH`: Path to the CSV data file (`"Book 4(Sheet1)1.csv"`)
- `label_colors`: Dictionary mapping cost categories to specific hex color codes
- `values_dict`: Extracted cost values from CSV
- `cost_list`: Numeric cost values
- `cost_list_lakhs`: Costs converted to lakhs (Indian currency unit)
- `percentage_list`: Calculated percentages for each cost category

Data Categories (10 cost types):

1. Road user cost
2. Time cost estimate
3. Embodied carbon emissions
4. Initial construction cost
5. Additional CO2 e costs due to rerouting
6. Periodic Maintenance costs
7. Periodic maintenance carbon emissions
8. Annual routine inspection costs
9. Repair and rehabilitation costs
10. Demolition and deconstruction costs

D3.js Visualization Components

1. Chart Configuration

```
const width = 600;
const height = 400;
const radius = Math.min(width, height) / 2 - 40;
const enlargedRadius = radius * 1.1;
```

Purpose: Defines chart dimensions and scaling parameters for normal and hover states.

2. Core D3.js Objects

Pie Layout Generator

```
const pie = d3.pie()
    .value(d => d.cost)
    .sort(null);
```

Purpose: Converts data into pie chart angles and positions.

Arc Generators

```
const arc = d3.arc()
    .innerRadius(0)
    .outerRadius(radius);

const enlargedArc = d3.arc()
    .innerRadius(0)
    .outerRadius(enlargedRadius);
```

Purpose: Generate SVG path data for pie slices in normal and enlarged states.

3. Animation Functions

`arcTween(d, i)`

Purpose: Creates smooth transitions between different pie chart states using D3's interpolation.

`arcTweenExit(d)`

Purpose: Handles smooth exit animations when slices are removed from the chart.

4. Interactive Features

Hover Effects

- **Slice Enlargement**: Hovered slices expand by 10% (`enlargedRadius`)
- **Color Brightening**: Hovered slices become brighter using `d3.color().brighter(0.5)`
- **Other Slices Dimming**: Non-hovered slices fade to 30% opacity and turn gray

Tooltip System

```
const tooltip = d3.select("body")
    .append("div")
    .attr("class", "tooltip");
```

Features:

- Displays cost, percentage, and label information
- Follows mouse cursor
- Smooth fade-in/fade-out transitions
- Positioned dynamically to avoid screen edges

Legend Interactivity

- **Click to Toggle**: Clicking legend items enables/disables corresponding pie slices
- **Visual Feedback**: Disabled items show strikethrough text
- **Hover Highlighting**: Legend hover highlights corresponding pie slices
- **Dynamic Updates**: Chart recalculates percentages when items are toggled

5. Data Update System

`updatePieChart()` **Function**

Purpose: Handles dynamic chart updates when legend items are toggled.

Process:

1. Filters out disabled items
2. Recalculates percentages based on active items
3. Updates pie chart geometry using D3's enter/update/exit pattern
4. Animates transitions between states
5. Updates legend text with new percentages

`toggleItem(item)` **Function**

Purpose: Toggles the enabled/disabled state of chart items.

Features:

- Updates data model
- Applies visual strikethrough to legend
- Triggers chart recalculation

CSS Styling

Key Style Classes

- `.chart-container`: Centers and constrains chart width
- `.pie-label`: Styles percentage labels on pie slices
- `.legend-item`: Interactive legend items with hover effects
- `.tooltip`: Floating information box with smooth transitions
- `.strikethrough`: Visual indicator for disabled items

Data Flow

1. **CSV Reading**: Pandas reads cost data from CSV file
2. **Data Extraction**: Script searches for predefined cost categories in CSV rows
3. **Data Transformation**:
    - Converts costs to lakhs (`÷100,000`)
    - Calculates percentages
    - Assigns specific colors to each category
4. **JSON Serialization**: Data converted to JSON for JavaScript consumption
5. **HTML Generation**: Complete HTML document created with embedded data
6. **Browser Rendering**: QWebEngineView renders the interactive chart
7. **User Interaction**: JavaScript handles all interactive features

Error Handling

- **File Not Found**: Graceful handling of missing CSV files
- **Empty Data**: Warning messages for empty datasets

- **Data Parsing**: Try-catch blocks for numeric conversion errors
- **Default Values**: Fallback to 0.0 for missing or invalid data

Performance Features

- **Smooth Animations**: 200–800ms transitions using D3's easing functions
- **Efficient Updates**: Uses D3's data join pattern for minimal DOM manipulation
- **Memory Management**: Proper cleanup of event listeners and DOM elements
- **Responsive Design**: Chart adapts to window resizing

5.3 Bubble Graph

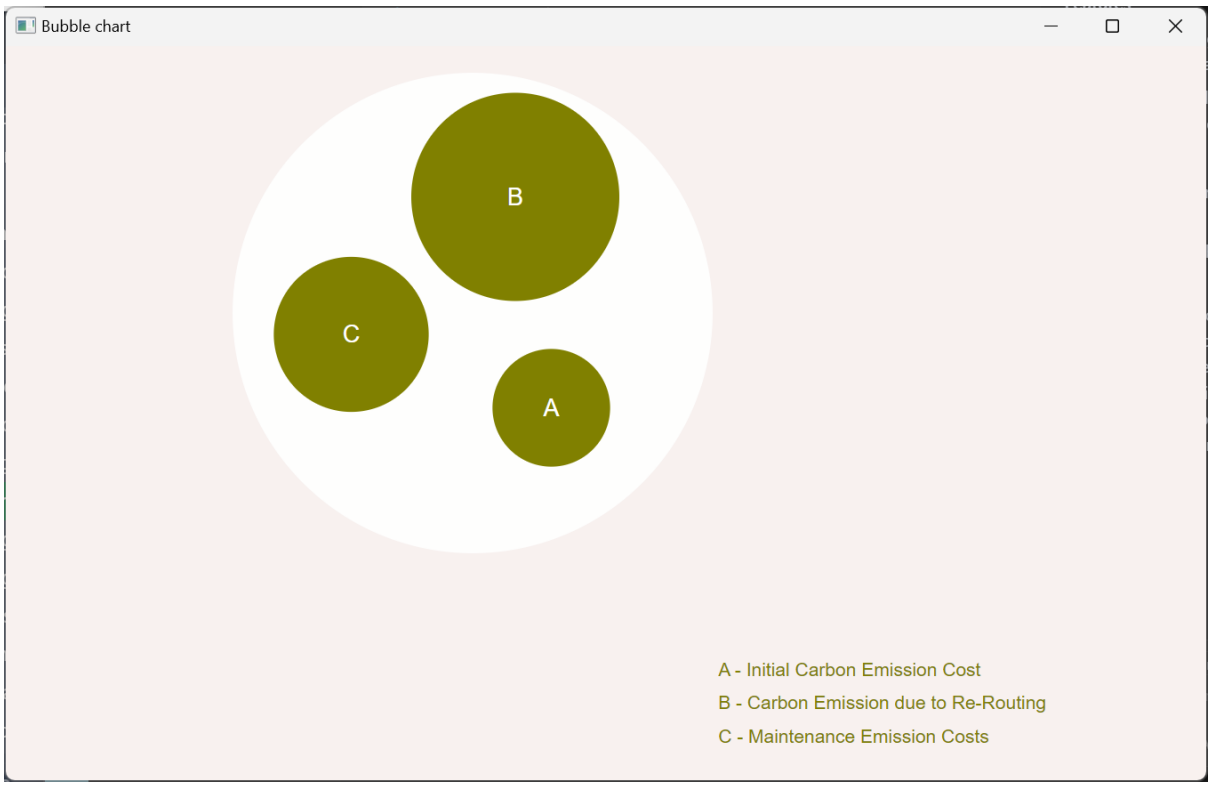
A bubble graph is an extension of a scatter plot, where a third dimension is represented by the size of the bubble.

This graph is implemented using the javascript library d3js. Which is an opensourced javascript library, useful in developing interactive graphs. The graph is laid on the foundation of HTML and CSS, and this entire combination of HTML, CSS, and Javascript is ran over python, which is supported by the python library, pyside6. Which is an interactive library in python to develop GUI for python applications.

Here the input values are used from an excel sheet(Book 4(Sheet1)1.csv) for testing purpose alone. The actual values are fetched from the database.

Use in LCC: Bubble graphs are useful for multi-dimensional LCC analysis. For example, they can represent different datas like Embodied carbon emissions, Additional CO2 e costs due to rerouting, Periodic maintenance carbon emissions

5.3.1 Image



5.3.2 Code

```
import sys
from PySide6.QtWidgets import QApplication, QMainWindow
from PySide6.QtWebEngineWidgets import QWebEngineView
import pandas as pd

FILE_PATH = r"Book 4(Sheet1).csv"
df = pd.read_csv(FILE_PATH)

name = ['A - Initial Carbon Emission Cost\n', 'B - Carbon
   Emission due to Re-Routing', 'C - Maintenance Emission Costs',
        "Embodied carbon emissions", "Additional CO2 e costs due
           to rerouting", "Periodic maintenance carbon emissions"
           ]
percentage_list = []
```

```
cost_list = []

for i, item in enumerate(name):
    count = 0
    if i < 3:
        for _, row in df.iterrows():
            if item in list(row):
                if count == 0:
                    temp_row = list(row)
                    percentage_list.append(float(temp_row[
                        temp_row.index(item) + 1]))
                    count += 1
    else:
        for _, row in df.iterrows():
            if item in list(row):
                if count == 0:
                    temp_row = list(row)
                    cost_list.append(float(temp_row[temp_row.
                        index(item) + 1]))
                    count += 1

percentage_list = [round(item, 2) for item in percentage_list]
print(percentage_list)
val_a, val_b, val_c = percentage_list

cost_list = [round(float(items) / 100000.0, 2) for items in
    cost_list]
cost_a, cost_b, cost_c = cost_list

# using the d3js graphing library to plot the graph, it is
    downloaded locally and saved in the same directory as that of
    this graph script
with open(r"d3js.js", "r", encoding="utf-8") as f:
    d3_js = f.read()
```

```
html_content = f"""
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Bubble Chart Layout</title>
  <script>{d3_js}</script>
  <style>
    body {{
      font-family: sans-serif;
      background: #f8f1ef;
      margin: 0;
      overflow: hidden;
    }}
    #chart-container {{
      position: relative;
      width: 800px;
      height: 550px;
    }}
    #legend {{
      position: absolute;
      right: 20px;
      bottom: 20px;
      font-size: 14px;
      line-height: 1.8;
    }}
    #legend span {{
      display: block;
      white-space: nowrap;
    }}
    .legend-text {{
```

```
    color: #808000;
    display: inline;
  }}
  .tooltip {{
    position: absolute;
    padding: 12px;
    background: white;
    border-radius: 16px;
    box-shadow: 0 0 12px rgba(0, 0, 0, 0.5);
    pointer-events: none;
    font-size: 14px;
    opacity: 0;
    transition: opacity 0.3s ease, transform 0.2s ease;
    transform: translateY(-10px);
  }}
  .tooltip.visible {{
    opacity: 1;
    transform: translateY(0);
  }}
  .tooltip::after {{
    content: "";
    position: absolute;
    top: 100%;
    left: 50%;
    transform: translateX(-50%);
    border-width: 10px 10px 0 10px;
    border-style: solid;
    border-color: white transparent transparent transparent;
    filter: drop-shadow(0 1px 2px rgba(0,0,0,0.2));
  }}
  .tooltip strong {{
    color: #5c8a00;
    font-size: 16px;
  }}
```

```
    .tooltip b {{
      font-size: 18px;
    }}
  </style>
</head>
<body>
  <div id="chart-container">
    <div id="chart"></div>
    <div id="legend">
      <span><span class="legend-text">A - Initial Carbon Emission
          Cost</span></span>
      <span><span class="legend-text">B - Carbon Emission due to
         Re-Routing</span></span>
      <span><span class="legend-text">C - Maintenance Emission
         Costs</span></span>
    </div>
  </div>
  <div class="tooltip" id="tooltip"></div>

  <script>
    const data = [
      {{ name: 'A', value: {val_a}, x: 409, y: 271, label:'
         Initial Carbon Emission Cost' , cost: {cost_a}}},
      {{ name: 'B', value: {val_b}, x: 382, y: 113, label:'Carbon
          Emission due to Re-Routing' , cost: {cost_b}}},
      {{ name: 'C', value: {val_c}, x: 259, y: 216, label:'
         Maintenance Emission Costs' , cost: {cost_c}}}
    ];

    const svgWidth = 750, svgHeight = 550;
    const radiusScale = d3.scaleLinear()
      .domain([0, d3.max(data, d => d.value)])
      .range([30, 78]);
```

```
const svg = d3.select("#chart")
  .append("svg")
  .attr("width", svgWidth)
  .attr("height", svgHeight);

svg.append("circle")
  .attr("cx", 350)
  .attr("cy", 200)
  .attr("r", 180)
  .attr("fill", "#ffffff")
  .attr("opacity", 0.9);

const tooltip = d3.select("#tooltip");

const groups = svg.selectAll("g")
  .data(data)
  .enter()
  .append("g")
  .attr("transform", d => `translate(${{d.x}}, ${{d.y}})`);

groups.append("circle")
  .attr("r", 0)
  .attr("fill", "#808000")
  .transition()
  .delay((_, i) => i * 300)
  .duration(1000)
  .attr("r", d => radiusScale(d.value));

// Event listeners with greying effect
groups.selectAll("circle")
  .on("mouseover", function (event, d) {{
    tooltip
      .html(`<strong>${{d.label}}</strong><br><b>${{d.cost}}
        Lakh; ${{d.value}} %</b>`)
```

```
        .style("left", (event.pageX - 120) + "px")
        .style("top", (event.pageY - 90) + "px")
        .classed("visible", true);

      // Highlight hovered and grey out others
      groups.selectAll("circle")
        .transition()
        .duration(300)
        .style("fill", c => c === d ? "#808000" : "#ccc");

      d3.select(this)
        .transition()
        .duration(300)
        .attr("r", radiusScale(d.value) * 1.1);
    }})
    .on("mousemove", function (event) {{
      tooltip
        .style("left", (event.pageX - 120) + "px")
        .style("top", (event.pageY - 90) + "px");
    }})
    .on("mouseout", function (event, d) {{
      tooltip.classed("visible", false);

      // Restore original color and size for all
      groups.selectAll("circle")
        .transition()
        .duration(10)
        .style("fill", "#808000")
        .attr("r", c => radiusScale(c.value));
    }});

  groups.append("text")
    .text(d => d.name)
    .attr("text-anchor", "middle")
```

```
        .attr("dy", ".35em")
        .style("fill", "white")
        .style("opacity", 0)
        .style("font-size", "12px")
        .transition()
        .delay((_, i) => i * 300 + 500)
        .duration(700)
        .style("opacity", 1)
        .style("font-size", "18px");
  </script>
</body>
</html>
"""

class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        self.setWindowTitle("Bubble chart")
        self.setGeometry(100, 100, 900, 550)

        view = QWebEngineView()
        view.setHtml(html_content)
        self.setCentralWidget(view)

if __name__ == "__main__":
    app = QApplication(sys.argv)
    window = MainWindow()
    window.show()
    sys.exit(app.exec())
```

5.3.3 Code Explanation

Overview

This Python application creates an interactive bubble chart visualization for carbon emission data using PySide6 (Qt) and D3.js. The chart displays three categories of carbon emissions with bubble sizes proportional to their percentage values and includes interactive tooltips showing cost information.

Dependencies and Imports

```
import sys
from PySide6.QtWidgets import QApplication, QMainWindow
from PySide6.QtWebEngineWidgets import QWebEngineView
import pandas as pd
```

- `sys`: System-specific parameters and functions
- `PySide6.QtWidgets`: Qt GUI framework for creating desktop applications
- `PySide6.QtWebEngineWidgets`: Qt component for rendering web content (HTML/CSS/JavaScript)
- `pandas`: Data manipulation and analysis library

Data Processing Section

Key Variables

```
FILE_PATH = r"Book 4(Sheet1).csv"
df = pd.read_csv(FILE_PATH)
```

- `FILE_PATH`: Path to the CSV file containing carbon emission data
- `df`: Pandas DataFrame object containing the loaded CSV data

Data Categories

```
name = ['A - Initial Carbon Emission Cost\n', 'B - Carbon Emission due to Re-Routing'
        "Embodied carbon emissions", "Additional CO2 e costs due to rerouting", "Peri
```

This list contains six categories:

- **First 3 elements**: Display names for the bubble chart (A, B, C categories)
- **Last 3 elements**: Corresponding data column names in the CSV file

Data Extraction Logic

```
percentage_list = []
cost_list = []


for i, item in enumerate(name):
    count = 0
    if i < 3:
        for _, row in df.iterrows():
            if item in list(row):
                if count == 0:
                    temp_row = list(row)
                    percentage_list.append(float(temp_row[temp_row.index(item) + 1]))
                    count += 1
    else:
        for _, row in df.iterrows():
            if item in list(row):
                if count == 0:
                    temp_row = list(row)
                    cost_list.append(float(temp_row[temp_row.index(item) + 1]))
                    count += 1
```

Function: This loop processes the CSV data to extract:

- `percentage_list`: Percentage values for bubble sizes (first 3 categories)

- `cost_list`: Cost values for tooltip display (last 3 categories)

Logic:

- Iterates through each category name
- Searches for matching column headers in the CSV
- Extracts the value from the adjacent column (index + 1)
- Uses `count` to ensure only the first occurrence is captured

Data Processing

```
percentage_list = [round(item, 2) for item in percentage_list]
val_a, val_b, val_c = percentage_list


cost_list = [round(float(items) / 100000.0, 2) for items in cost_list]
cost_a, cost_b, cost_c = cost_list
```

Functions:

- **Rounding**: Converts percentage values to 2 decimal places
- **Cost conversion**: Divides cost values by 100,000 to convert to "Lakh" units
- **Variable assignment**: Assigns individual values to `val_a`, `val_b`, `val_c` and `cost_a`, `cost_b`, `cost_c`

D3.js Integration

```
with open(r"d3js.js", "r", encoding="utf-8") as f:
    d3_js = f.read()
```

Function: Reads the local D3.js library file and stores it as a string for embedding in the HTML.

HTML Content Generation

The `html_content` variable contains a complete HTML document with embedded CSS and JavaScript. Key components include:

CSS Styling

- **Background**: Light beige (#f8f1ef)
- **Chart container**: 800x550px with relative positioning
- **Legend**: Positioned at bottom-right with olive green text (#808000)
- **Tooltip**: White background with rounded corners and drop shadow

JavaScript D3.js Visualization

Data Structure

```
const data = [
  { name: 'A', value: val_a, x: 409, y: 271, label:'Initial Carbon Emission Cost', co
  { name: 'B', value: val_b, x: 382, y: 113, label:'Carbon Emission due to Re-Routing
  { name: 'C', value: val_c, x: 259, y: 216, label:'Maintenance Emission Costs', cost
];
```

Properties:

- `name`: Bubble identifier (A, B, C)
- `value`: Percentage value for bubble size
- `x, y`: Fixed coordinates for bubble positioning
- `label`: Full description for tooltip
- `cost`: Cost value in Lakh units

D3.js Components

Scale Function:

```
const radiusScale = d3.scaleLinear()
  .domain([0, d3.max(data, d => d.value)])
  .range([30, 78]);
```

SVG Setup:

```
const svg = d3.select("#chart")
  .append("svg")
  .attr("width", svgWidth)
  .attr("height", svgHeight);
```

Background Circle:

```
svg.append("circle")
  .attr("cx", 350)
  .attr("cy", 200)
  .attr("r", 180)
  .attr("fill", "#ffffff")
  .attr("opacity", 0.9);
```

Interactive Features

Bubble Creation with Animation:

```
groups.append("circle")
  .attr("r", 0)
  .attr("fill", "#808000")
  .transition()
  .delay((_, i) => i * 300)
  .duration(1000)
  .attr("r", d => radiusScale(d.value));
```

- **Delay**: Each bubble appears 300ms after the previous one

- **Duration**: Animation takes 1000ms to complete

Event Handlers:

- `mouseover`: Shows tooltip, highlights hovered bubble, greys out others
- `mousemove`: Updates tooltip position as mouse moves
- `mouseout`: Hides tooltip and restores original appearance

Text Labels:

```
groups.append("text")
  .text(d => d.name)
  .attr("text-anchor", "middle")
  .attr("dy", ".35em")
  .style("fill", "white")
  .style("opacity", 0)
  .style("font-size", "12px")
  .transition()
  .delay((_, i) => i * 300 + 500)
  .duration(700)
  .style("opacity", 1)
  .style("font-size", "18px");
```

GUI Application Class

```
class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        self.setWindowTitle("Bubble chart")
        self.setGeometry(100, 100, 900, 550)

        view = QWebEngineView()
        view.setHtml(html_content)
        self.setCentralWidget(view)
```

Class: `MainWindow` – Qt main window for displaying the bubble chart

Methods:

- `__init__()`: Constructor that sets up the window properties and embeds the HTML content
- **Window properties**: Title "Bubble chart", size 900x550 pixels, positioned at (100,100)
- `QWebEngineView`: Widget that renders the HTML content containing the D3.js visualization

Application Entry Point

```
if __name__ == "__main__":
    app = QApplication(sys.argv)
    window = MainWindow()
    window.show()
    sys.exit(app.exec())
```

Function: Application startup sequence

1. `QApplication`: Creates the Qt application instance
2. `MainWindow()`: Instantiates the main window
3. `window.show()`: Displays the window
4. `app.exec()`: Starts the event loop for user interaction

Key Features Summary

- **Data Visualization**: Interactive bubble chart showing carbon emission percentages
- **Interactive Elements**: Hover tooltips with cost information and visual feedback
- **Smooth Animations**: Staggered bubble entrance and hover effects

- **Responsive Design**: Tooltip follows mouse movement
- **Professional Styling**: Clean, modern interface with proper color scheme
- **Data Processing**: Automatic extraction and conversion of CSV data
- **Cross-platform**: Works on Windows, macOS, and Linux using Qt framework

5.4 Radial Bar Graph

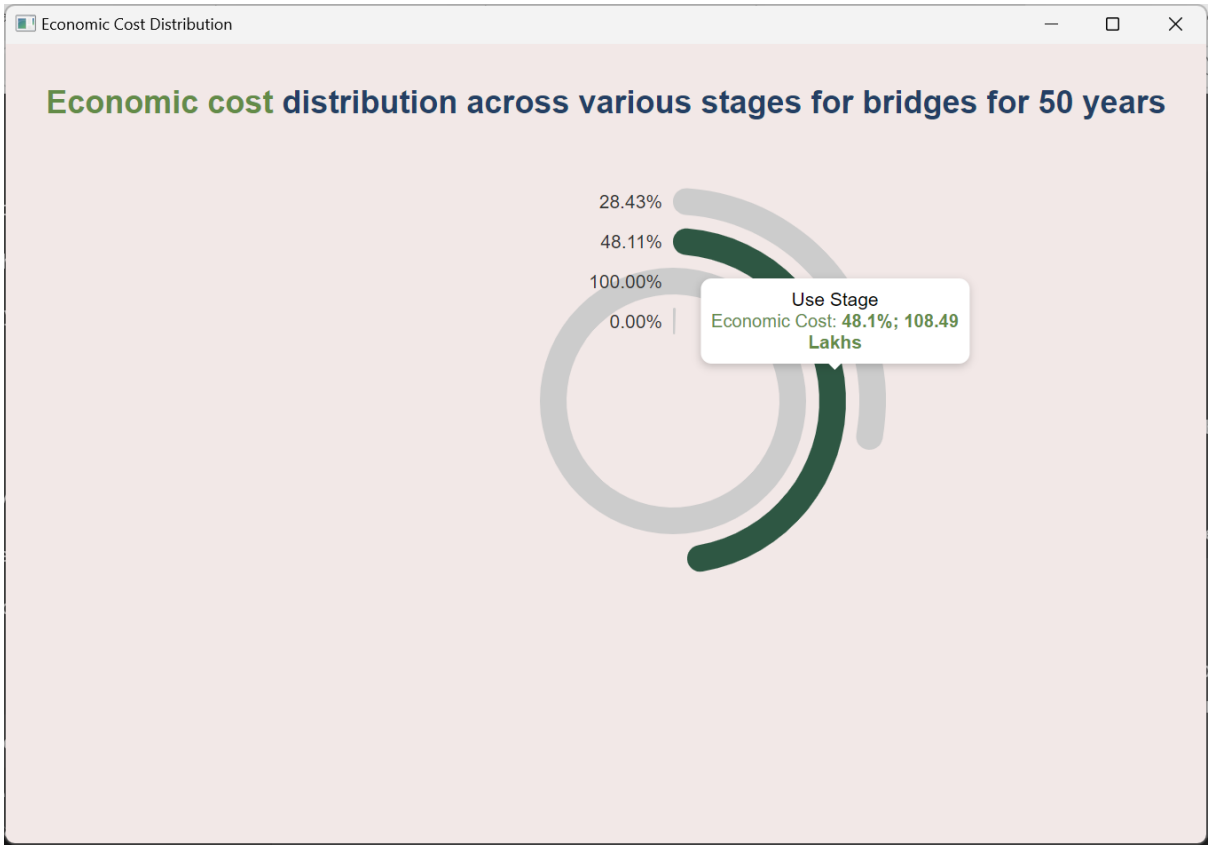
A radial bar graph (or circular bar plot) is a variation of the bar chart where bars are plotted in a circular layout.

This graph is implemented using the javascript library d3js. Which is an opensourced javascript library, useful in developing interactive graphs. The graph is laid on the foundation of HTML and CSS, and this entire combination of HTML, CSS, and Javascript is ran over python, which is supported by the python library, pyside6. Which is an interactive library in python to develop GUI for python applications.

Here the input values are used from an excel sheet(Book 4(Sheet1)1.csv) for testing purpose alone. The actual values are fetched from the database.

Use in LCC: Radial bar graphs provide an aesthetic and compact way to compare cost categories, especially when aiming for visual appeal in reports. They are effective in summarizing cost distribution when the focus is on visual storytelling.

5.4.1 Image



5.4.2 Code

```
import sys
from PySide6.QtWidgets import QApplication, QMainWindow
from PySide6.QtWebEngineWidgets import QWebEngineView
import pandas as pd
import copy

# using the d3js graphing library to plot the graph, it is
    downloaded locally and saved in the same directory as that of
    this graph script
with open(r"d3js.js", "r", encoding="utf-8") as f:
    d3_js = f.read()

FILE_PATH = r"Book 4(Sheet1).csv"
```

```
df = pd.read_csv(FILE_PATH)

# hexal color code for various concentric circles, used inside
   the HTML js script text(its inside the string html_content)
colors = ['#273B5C', '#2E5743', '#996515', '#36454F']

# labels used displaying purpose, used inside the HTML js script
   text(its inside the string html_content)
stage_label = ['Initial Stage', 'Use Stage', 'End of Life Stage',
    'Beyond Life Stage']
stage_label_condition = ['initialstage', 'usestage', '
   endoflifestage', 'beyondlifestage']
percentage = []

# for loop to extract the values, modify according to the style
for ___, row in df.iterrows():
    if isinstance(row[8], str):
        if row[8].lower().replace(" ", "") in
           stage_label_condition:
            percentage.append(float(row[9]))
percentage = percentage[:4]

# html + js + css script to generate the radial bar graph, we
   used python's format to plug the values which we have
   extracted from the csv file
html_content = f"""
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title></title>
  <style>
    body {{
      background: #f2e8e7;
```

```
    font-family: sans-serif;
    display: flex;
    flex-direction: column;
    justify-content: start;
    align-items: center;
    height: 100vh;
    margin: 0;
  }}
  h2 {{
    margin-top: 30px;
    margin-bottom: 10px;
    font-size: 24px;
    color: #243f64;
  }}
  .tooltip {{
    position: absolute;
    text-align: center;
    padding: 8px;
    background: white;
    border-radius: 8px;
    box-shadow: 0 2px 6px rgba(0,0,0,0.2);
    pointer-events: none;
    opacity: 0;
    font-size: 14px;
    transform: translateX(-50%);
  }}
  .tooltip:after {{
    content: "";
    position: absolute;
    top: 100%;
    left: 50%;
    margin-left: -5px;
    border-width: 5px;
    border-style: solid;
```

```
      border-color: white transparent transparent transparent;
    }}
    text.label {{
      font-size: 14px;
      fill: #333;
    }}
  </style>
</head>
<body>

<h2><span style="color: #638B48">Economic cost</span>
   distribution across various stages for bridges for 50 years</
   h2>
<svg width="500" height="400"></svg>
<div class="tooltip"></div>

<script>{d3_js}</script>
<script>
 //constant data, extracted from csv file
  const data = [
    {{ name: '{stage_label[3]}', value: {percentage[3] / 100},
       color: '{colors[3]}' }},
    {{ name: '{stage_label[2]}', value: {percentage[2] / 100},
       color: '{colors[2]}' }},
    {{ name: '{stage_label[1]}', value: {percentage[1] / 100},
       color: '{colors[1]}' }},
    {{ name: '{stage_label[0]}', value: {percentage[0] / 100},
       color: '{colors[0]}' }}
  ];

//window property
  const width = 500, height = 400;
  const barWidth = 20;
  const spacing = 10;
```

```
  const center = {{ x: width / 2 + 50, y: height / 2 }};

  const svg = d3.select("svg");
  const g = svg.append("g")
    .attr("transform", `translate(${{center.x}},${{center.y}})`);

  const tooltip = d3.select(".tooltip");
// code for generating arcs
  data.forEach((d, i) => {{
    const innerRadius = 50 + i * (barWidth + spacing);
    const outerRadius = innerRadius + barWidth;

   //code to generate arc
    const arcGen = d3.arc()
      .innerRadius(innerRadius)
      .outerRadius(outerRadius)
      .startAngle(0)
      .cornerRadius(10);

    // the below is used to fix the colors for the circular bar
    const path = g.append("path")
      .datum(d)
      .attr("fill", d.color)
      .attr("d", arcGen({{ endAngle: 0 }}))
      .attr("class", "arc");

// code below is for creating animation in the start, the entire
   animation runs for 1200ms
    path.transition()
      .duration(1200)
      .attrTween("d", function(d) {{
        const interpolate = d3.interpolate(0, (d.value * 2 * Math
           .PI) + 0.03);
        return function(t) {{
```

```
          return arcGen({{ endAngle: interpolate(t) }});
        }};
      }});

    // the below 2 lines are used to place the static percentage
       values to the left of the radial bar graph
    const labelX = center.x - 8;
    const labelY = center.y + (data.length - 1 - i) * (barWidth +
        spacing) -150;

    // the below code fixes the position of the static percentage
        label
    svg.append("text")
      .attr("x", labelX)
      .attr("y", labelY + 5)
      .attr("text-anchor", "end")
      .attr("class", "label")
      .text(`${{(d.value * 100).toFixed(2)}}%`);

// the below entire code is to provide hover event response, that
    is when we hover over a radial bar or arc, we get a small pop
    up info window
    path.on("mouseover", function(event, hoveredData) {{
        d3.selectAll(".arc")
          .attr("fill", p => p === hoveredData ? p.color : "#ccc
             ");

            tooltip.style("opacity", 1)
.html(`
  <div style="text-align:center; font-family:sans-serif;">
    <span style="font-weight: 500;">${{hoveredData.name}}</span>
    <br>
    <span style="color: #638B48; font-size: 13.5px;">
      Economic Cost:
```

```
      <span style="font-weight: 600;">
        ${{(hoveredData.value * 100).toFixed(1)}}%;
        ${{(hoveredData.value * 225.5).toFixed(2)}}
      </span>
    </span>
    <br>
    <span style="font-weight: bold; color: #638B48;">Lakhs</span>
  </div>
`).style("left", (event.pageX) + "px")
                .style("top", (event.pageY - 50) + "px");
      }})
      .on("mousemove", function(event) {{
        tooltip.style("left", (event.pageX) + "px")
                .style("top", (event.pageY - 70) + "px");
      }})
      .on("mouseout", function() {{
        d3.selectAll(".arc")
          .attr("fill", d => d.color);
        tooltip.style("opacity", 0);
      }});
  }});
</script>

</body>
</html>
"""


class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        self.setWindowTitle("Economic Cost Distribution")

        view = QWebEngineView()
```

```
        view.setHtml(html_content)
        self.setCentralWidget(view)
        self.resize(700, 600)


if __name__ == "__main__":
    app = QApplication(sys.argv)
    window = MainWindow()
    window.show()
    sys.exit(app.exec())
```

5.4.3 Code Explanation

Overview

This Python application creates an interactive radial bar graph (also known as a circular bar chart) to visualize economic cost distribution across different lifecycle stages of bridges over 50 years. The application uses PySide6 for the GUI framework and D3.js for advanced data visualization.

Key Components and Functions

1. Import Statements and Dependencies

```
import sys
from PySide6.QtWidgets import QApplication, QMainWindow
from PySide6.QtWebEngineWidgets import QWebEngineView
import pandas as pd
import copy
```

- **PySide6**: Modern Python binding for Qt framework, providing GUI components
- **QWebEngineView**: Widget that renders web content (HTML/JavaScript) within the application

- **pandas**: Data manipulation and CSV file reading
- **copy**: Deep copying functionality (though not used in current implementation)

2. D3.js Library Integration

```
with open(r"d3js.js", "r", encoding="utf-8") as f:
    d3_js = f.read()
```

- Loads the D3.js library from a local file
- D3.js is a powerful JavaScript library for data visualization
- The library is embedded directly into the HTML content for rendering

3. Data Source and Processing

```
FILE_PATH = r"Book 4(Sheet1).csv"
df = pd.read_csv(FILE_PATH)
```

- **FILE_PATH**: Path to the CSV file containing economic cost data
- **df**: Pandas DataFrame object containing the raw data

4. Visual Configuration Variables

Color Scheme

```
colors = ['#273B5C', '#2E5743', '#996515', '#36454F']
```

Stage Labels

```
stage_label = ['Initial Stage', 'Use Stage', 'End of Life Stage', 'Beyond Life Stage']
stage_label_condition = ['initialstage', 'usestage', 'endoflifestage', 'beyondlifesta
```

- **stage_label**: Human-readable labels for display
- **stage_label_condition**: Normalized strings for data matching (lowercase, no spaces)

5. Data Extraction Function

```
percentage = []
for ___, row in df.iterrows():
    if isinstance(row[8], str):
        if row[8].lower().replace(" ", "") in stage_label_condition:
            percentage.append(float(row[9]))
percentage = percentage[:4]
```

- Iterates through DataFrame rows
- Matches stage identifiers against normalized strings
- Extracts and limits values to first 4 corresponding to lifecycle stages

6. HTML Content Generation

CSS Styling

- **Background**: Light pink (#f2e8e7)
- **Typography**: Sans-serif font family with specific color scheme
- **Tooltip**: White background with shadow and arrow pointer
- **Responsive Design**: Flexbox layout for proper alignment

JavaScript D3.js Implementation

Data Structure

```
const data = [
    { name: 'Stage Name', value: percentage/100, color: 'hex_color' }
];
```

- Converts percentages to decimal
- Associates each stage with label and color

Graph Configuration

```
const width = 500, height = 400;
const barWidth = 20;
const spacing = 10;
const center = { x: width / 2 + 50, y: height / 2 };
```

Arc Generation

```
const arcGen = d3.arc()
    .innerRadius(innerRadius)
    .outerRadius(outerRadius)
    .startAngle(0)
    .cornerRadius(10);
```

- Generates SVG paths for radial bars with rounded corners

Animation System

```
path.transition()
    .duration(1200)
    .attrTween("d", function(d) {
        const interpolate = d3.interpolate(0, (d.value * 2 * Math.PI) + 0.03);
        return function(t) {
            return arcGen({ endAngle: interpolate(t) });
        };
    });
```

Interactive Features

```
path.on("mouseover", function(event, hoveredData) {
    // Highlight hovered bar, dim others
    // Show tooltip with detailed information
})
.on("mousemove", function(event) {
    // Update tooltip position
})
.on("mouseout", function() {
```

```
    // Restore original colors, hide tooltip
});
```

- Tooltips display percentage and cost (converted to Lakhs)
- Highlighted feedback with mouse movement

7. MainWindow Class

```
class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        self.setWindowTitle("Economic Cost Distribution")

        view = QWebEngineView()
        view.setHtml(html_content)
        self.setCentralWidget(view)
        self.resize(700, 600)
```

- Sets title and window size
- Loads D3.js visual via `QWebEngineView`

8. Application Entry Point

```
if __name__ == "__main__":
    app = QApplication(sys.argv)
    window = MainWindow()
    window.show()
    sys.exit(app.exec())
```

- Initializes and displays the Qt application

Key Features Summary

1. Data Visualization: Interactive radial bar chart showing cost distribution

2. Lifecycle Stages: Four distinct stages of bridge lifecycle
3. Interactive Elements: Hover effects with detailed tooltips
4. Animation: Smooth loading animation for visual appeal
5. Responsive Design: Properly centered and styled layout
6. Data Integration: Direct CSV file reading and processing
7. Cross-Platform: Works on Windows, macOS, and Linux

Technical Architecture

- **Frontend**: HTML/CSS/JavaScript with D3.js
- **Backend**: Python with PySide6 framework
- **Data Processing**: Pandas for CSV manipulation
- **Rendering**: Web engine for HTML content display
- **Integration**: Python string formatting for data injection

Chapter 6

Saving CAD Model

This task was given as a subtask, this task does not have any effect or improvement in osdag project. This task's outcome was used in building an asset for developing animation. This task is all about extracting the 3D model which is generated while the respective class in the respective file is called by external functions to display the 3D model in the osdag UI window. The generated model is saved locally in .gltf format, which is compatible 3D animation building software like blender.

6.1 Algorithm/Logic to save

The aim of this task is to save the generated CAD model, whose parameters were shared in .OSI file. The model which I was asked to save was Beam Column End Plate, so initially I went to the file *BCE_nutBoltPlacement.py* inside the directory, *cad\MomentConnections\BCEndplate\BCE_nutBoltPlacement.py*

In this file, I used the method *def get_models(self):*, which was returning *self.model*, but this approach didn't work as it was saving empty .gltf file.

So, I went to the file BCEndplateBCEndplate_cadfile.py, and inside the function `get_models()` , I used the OCC's import

```
from OCC.Extend.DataExchange import write_gltf_file
```

and saved each part of the Beam Column End Plate model like columns, beams, plate_connectors, welds, nut_bolt_array. These models are saved individually using write_gltf_file() function.

6.2 Code

```
def get_models(self):

        """

        :return: complete CAD model
        """
        columns = self.get_column_models()
        print("Saving columns gltf file")
        from OCC.Extend.DataExchange import write_gltf_file
        import os
        output_path = os.path.join(os.path.expanduser("~"), "
           Desktop", "columns.gltf")
        write_gltf_file(columns, output_path)


        beams = self.get_beam_models()
        print("Saving beams gltf file")
        output_path = os.path.join(os.path.expanduser("~"), "
           Desktop", "beams.gltf")
        write_gltf_file(beams, output_path)


        plate_connectors = self.get_plate_connector_models()
        print("Saving plate_connectors gltf file")
        output_path = os.path.join(os.path.expanduser("~"), "
           Desktop", "plate_connectors.gltf")
        write_gltf_file(plate_connectors, output_path)

        welds = self.get_welded_models()
        print("Saving welds gltf file")
        output_path = os.path.join(os.path.expanduser("~"), "
           Desktop", "welds.gltf")
```

```
        write_gltf_file(welds, output_path)

        nut_bolt_array = self.get_nut_bolt_array_models()
        print("Saving nut_bolt_array gltf file")
        output_path = os.path.join(os.path.expanduser("~"), "
           Desktop", "nut_bolt_array.gltf")
        write_gltf_file(nut_bolt_array, output_path)

        CAD_list = [columns, beams, plate_connectors, welds,
           nut_bolt_array]
        # CAD_list = [columns, beams, plate_connectors,
           nut_bolt_array]

        CAD = CAD_list[0]

        for model in CAD_list[1:]:
            CAD = BRepAlgoAPI_Fuse(CAD, model).Shape()



        return CAD
```

6.3 Documentation

Method: `get_models()`

Purpose

This method generates and exports a complete 3D CAD model of a beam-column endplate connection by combining all individual components into a unified assembly.

Functionality Breakdown

1. Component Generation and Export

The method creates separate 3D models for each structural component:

- **Columns**: Main structural columns using `get_column_models()`
- **Beams**: Connected beams using `get_beam_models()`
- **Plate Connectors**: Endplates and connection plates using `get_plate_connector_models()`
- **Welds**: Welded connections using `get_welded_models()`
- **Nut-Bolt Arrays**: Fastening elements using `get_nut_bolt_array_models()`

2. GLTF File Export

Each component is exported as a separate GLTF (Graphics Library Transmission Format) file to the user's Desktop:

- `columns.gltf`
- `beams.gltf`
- `plate_connectors.gltf`
- `welds.gltf`
- `nut_bolt_array.gltf`

3. Assembly Creation

The method performs a Boolean union operation (`BRepAlgoAPI_Fuse`) to merge all components into a single unified CAD model:

```
CAD_list = [columns, beams, plate_connectors, welds, nut_bolt_array]
CAD = CAD_list[0]


for model in CAD_list[1:]:
    CAD = BRepAlgoAPI_Fuse(CAD, model).Shape()
```

Technical Details

- **File Format**: GLTF is a standard 3D file format that supports geometry, materials, and animations

- **Boolean Operations**: Uses OpenCascade's `BRepAlgoAPI_Fuse` for solid modeling union operations
- **Output Location**: Files are saved to the user's Desktop directory
- **Dependencies**: Requires OpenCascade (OCC) library for CAD operations

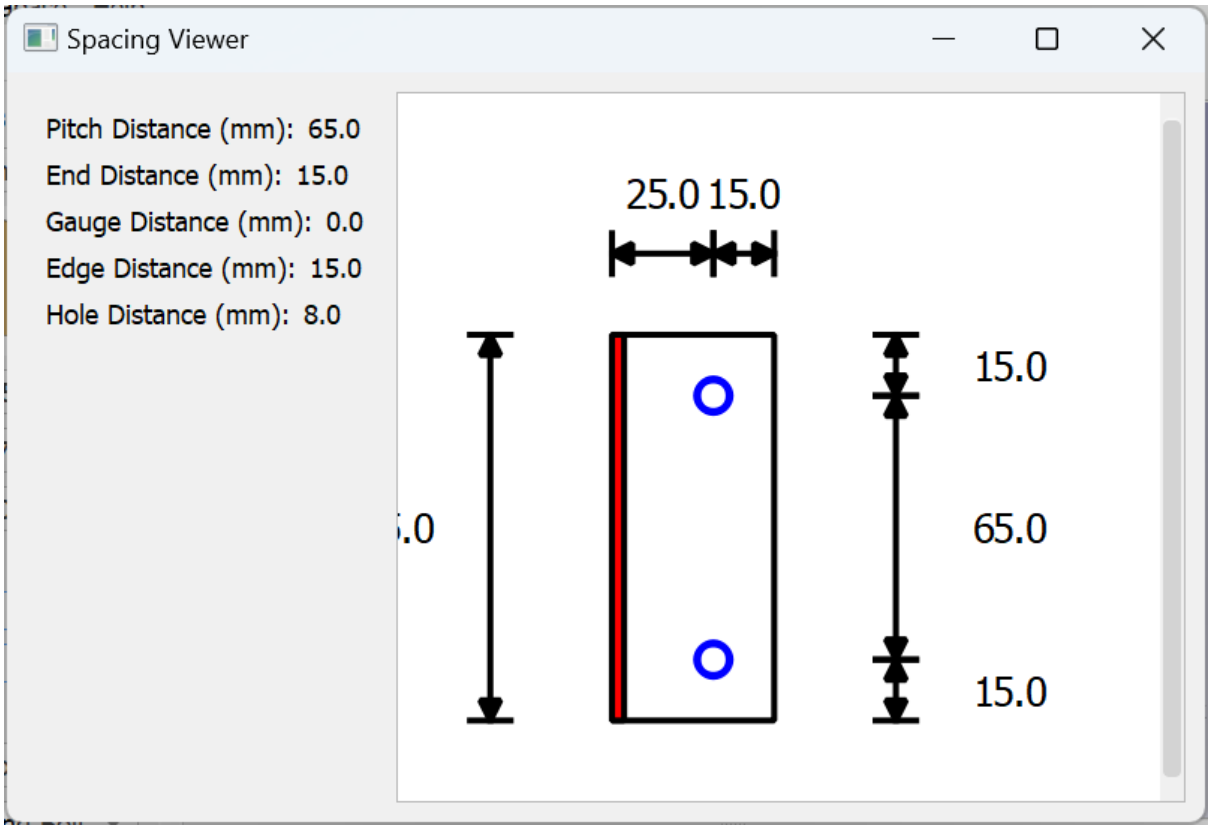
Chapter 7

Spacing & Capacity Details Window - Shear Connection

7.1 Shear Connection - Fin Plate

7.1.1 Spacing Window

Spacing Window Image



7.1.2 Code

```
import sys
from PyQt5.QtWidgets import (QApplication, QMainWindow, QWidget,
   QVBoxLayout,
                             QHBoxLayout, QLabel, QGraphicsView,
                             QGraphicsScene)
from PyQt5.QtGui import QPixmap
from PyQt5.QtCore import Qt, QRectF
from PyQt5.QtGui import QPainter, QPen, QFont
from PyQt5.QtGui import QPolygonF, QBrush
from PyQt5.QtCore import QPointF
from ..Common import *
from .additionalfns import calculate_total_width
from ..design_type.connection.end_plate_connection import
   EndPlateConnection

class BoltPatternGenerator(QMainWindow):
    def __init__(self, connection_obj, rows=3, cols=2 , main =
       None):
        super().__init__()
        self.connection = connection_obj
        self.main=main
        self.plate_height = main.plate.height
        self.plate_width = main.plate.length
        self.hole_dia=main.bolt.bolt_diameter_provided
        self.rows=main.plate.bolts_one_line
        self.cols=main.plate.bolt_line
        print(self.plate_height,self.plate_width)
        output=main.output_values(main,True)
        dict1={i[0] : i[3] for i in output}
        for i in output:
            print(i)
```

```
        self.weldsize=0
        if 'Weld.Size' in dict1:
            self.weldsize=dict1['Weld.Size']
        self.initUI()
        # print(self.connection.spacing(status=True))
    def initUI(self):
        self.setWindowTitle('Bolt Pattern Generator')
        self.setGeometry(100, 100, 800, 500)

        # Main layout
        main_layout = QHBoxLayout()

        # Left panel for parameter display
        left_panel = QWidget()
        left_layout = QVBoxLayout()

        # Parameter display labels
        params = self.get_parameters()

        # Display the parameter values
        for key, value in params.items():
            param_layout = QHBoxLayout()
            param_label = QLabel(f'{key.title()} Distance (mm):')
            value_label = QLabel(f'{value}')
            param_layout.addWidget(param_label)
            param_layout.addWidget(value_label)
            left_layout.addLayout(param_layout)

        left_layout.addStretch()
        left_panel.setLayout(left_layout)

        # Right panel for the drawing using QGraphicsView
        self.scene = QGraphicsScene()
        self.view = QGraphicsView(self.scene)
```

```
        self.view.setRenderHint(QPainter.Antialiasing)

        # Create and add the drawing to the scene
        self.createDrawing(params)

        # Add panels to main layout
        main_layout.addWidget(left_panel, 1)
        main_layout.addWidget(self.view, 3)

        # Set main widget
        main_widget = QWidget()
        main_widget.setLayout(main_layout)
        self.setCentralWidget(main_widget)

        # Ensure the view shows all content
        self.view.fitInView(self.scene.sceneRect(), Qt.
            KeepAspectRatio)
    def get_parameters(self):
        spacing_data = self.connection.spacing(status=True)  #
            Get actual values
        param_map = {}
        print('spacing_data length' , len(spacing_data))
        for item in spacing_data:
            key, _, _, value = item
            # print('key : ', key)
            if key == KEY_OUT_PITCH:
                param_map['pitch'] = float(value)
            elif key == KEY_OUT_END_DIST:
                param_map['end'] = float(value)
            elif key == KEY_OUT_GAUGE1:
                param_map['gauge1'] = float(value)
            elif key == KEY_OUT_GAUGE2:
                param_map['gauge2'] = float(value)
            elif key == KEY_OUT_GAUGE:
```

```
                param_map['gauge'] = float(value)
            elif key == KEY_OUT_EDGE_DIST:
                param_map['edge'] = float(value)

        # Add hardcoded hole diameter
        param_map['hole'] = self.main.bolt.bolt_diameter_provided

        print("Extracted parameters:", param_map)

        return param_map

    def createDrawing(self, params):

        # Extract parameters
        pitch = params['pitch']
        end = params['end']
        if 'gauge' in params:
            gauge = params['gauge']
        else:
            gauge1 = params['gauge1']
            gauge2 = params['gauge2']
        edge = params['edge']
        hole_diameter = params['hole']

        # Calculate dimensions
        if 'gauge' in params:
            gauge1 = gauge
            gauge2 = gauge
        width = self.plate_width

        height = self.plate_height

        # Set up pens
        outline_pen = QPen(Qt.blue, 2)
```

```
        dimension_pen = QPen(Qt.black, 1.5)
        red_brush = QBrush(Qt.red)

        # Dimension offsets
        h_offset = 40
        v_offset = 60

        # Create scene rectangle with extra space for dimensions
        self.scene.setSceneRect(-h_offset, -v_offset,
                                width + 2*v_offset, height + 2*
                                   h_offset)

        # Draw rectangle
        self.scene.addRect(0, 0, width, height, dimension_pen)

        # Draw holes
        for row in range(self.rows):
            for col in range(self.cols):
                # Start from right edge (for example: total plate
                    width - edge)
                x_center = self.plate_width - edge

                # Subtract gauges from right to left
                for i in range(col):
                    x_center -= gauge1 if i % 2 == 0 else gauge2

                # Y-position stays the same
                y_center = end + row * pitch

                # Top-left corner for drawing the circle
                x = x_center - hole_diameter / 2
                y = y_center - hole_diameter / 2

                print(f"row: {row}, col: {col}, x: {x}, y: {y}")
```

```
                self.scene.addEllipse(x, y, hole_diameter,
                    hole_diameter, outline_pen)
        weld_size=self.weldsize
        self.scene.addRect(0, 0, weld_size, height, dimension_pen
            ,red_brush)
        print(params,dimension_pen)
        # Add dimensions
        self.addDimensions(params, dimension_pen)

    def addDimensions(self, params, pen):
        # Extract parameters
        pitch = params['pitch']
        end = params['end']
        if 'gauge' in params:
            gauge = params['gauge']
        else:
            gauge1 = params['gauge1']
            gauge2 = params['gauge2']
        edge = params['edge']

        if 'gauge' in params:
            gauge1 = gauge
            gauge2 = gauge

        width = self.plate_width
        height = self.plate_height

        # Offsets for dimension lines
        h_offset = 20
        v_offset = 30

        # Add horizontal dimensions
        x_start = width
        segments = []
```

```
        # First edge
        segments.append(('edge', x_start-edge, x_start ))
        x_start -=edge

        # Last edge
        segments.append(('edge', 0, x_start))

        # Draw each segment
        for label, x1, x2 in segments:
            value = x2 - x1
            self.addHorizontalDimension(x1, -h_offset, x2, -
               h_offset, f"{value:.1f}", pen)
        # Add vertical dimensions
        self.addVerticalDimension(width + v_offset, 0, width +
           v_offset, end, str(end), pen)
        for i in range(self.rows - 1):
            self.addVerticalDimension(width + v_offset, end + i *
                pitch, width + v_offset, end + (i + 1) * pitch,
               str(pitch), pen)

        # Add bottom end distance dimension
        self.addVerticalDimension(width + v_offset, height, width
            + v_offset, height - end, str(end), pen)

        # Add left side dimension
        total_height = 2 * end + (self.rows - 1) * pitch
        self.addVerticalDimension(-v_offset, 0, -v_offset,
           total_height, str(total_height), pen)

    def addHorizontalDimension(self, x1, y1, x2, y2, text, pen):
        self.scene.addLine(x1, y1, x2, y2, pen)
        arrow_size = 5
        ext_length = 10
```

```
        self.scene.addLine(x1, y1 - ext_length/2, x1, y1 +
            ext_length/2, pen)
        self.scene.addLine(x2, y2 - ext_length/2, x2, y2 +
            ext_length/2, pen)

        points_left = [
            (x1, y1),
            (x1 + arrow_size, y1 - arrow_size/2),
            (x1 + arrow_size, y1 + arrow_size/2)
        ]
        polygon_left = self.scene.addPolygon(QPolygonF([QPointF(x
            , y) for x, y in points_left]), pen)
        polygon_left.setBrush(QBrush(Qt.black))

        points_right = [
            (x2, y2),
            (x2 - arrow_size, y2 - arrow_size/2),
            (x2 - arrow_size, y2 + arrow_size/2)
        ]
        polygon_right = self.scene.addPolygon(QPolygonF([QPointF(
            x, y) for x, y in points_right]), pen)
        polygon_right.setBrush(QBrush(Qt.black))

        text_item = self.scene.addText(text)
        font = QFont()
        font.setPointSize(5)
        text_item.setFont(font)

        if y1 < 0:
            text_item.setPos((x1 + x2) / 2 - text_item.
                boundingRect().width() / 2, y1 - 25)
        else:
            text_item.setPos((x1 + x2) / 2 - text_item.
                boundingRect().width() / 2, y1 + 5)
```

```
    def addVerticalDimension(self, x1, y1, x2, y2, text, pen):
        self.scene.addLine(x1, y1, x2, y2, pen)
        arrow_size = 5
        ext_length = 10
        self.scene.addLine(x1 - ext_length/2, y1, x1 + ext_length
           /2, y1, pen)
        self.scene.addLine(x2 - ext_length/2, y2, x2 + ext_length
           /2, y2, pen)

        if y2 > y1:
            points_top = [
                (x1, y1),
                (x1 - arrow_size/2, y1 + arrow_size),
                (x1 + arrow_size/2, y1 + arrow_size)
            ]
            polygon_top = self.scene.addPolygon(QPolygonF([
               QPointF(x, y) for x, y in points_top]), pen)
            polygon_top.setBrush(QBrush(Qt.black))

            points_bottom = [
                (x2, y2),
                (x2 - arrow_size/2, y2 - arrow_size),
                (x2 + arrow_size/2, y2 - arrow_size)
            ]
            polygon_bottom = self.scene.addPolygon(QPolygonF([
               QPointF(x, y) for x, y in points_bottom]), pen)
            polygon_bottom.setBrush(QBrush(Qt.black))
        else:
            points_top = [
                (x2, y2),
                (x2 - arrow_size/2, y2 + arrow_size),
                (x2 + arrow_size/2, y2 + arrow_size)
            ]
```

```
            polygon_top = self.scene.addPolygon(QPolygonF([
                QPointF(x, y) for x, y in points_top]), pen)
            polygon_top.setBrush(QBrush(Qt.black))

            points_bottom = [
                (x1, y1),
                (x1 - arrow_size/2, y1 - arrow_size),
                (x1 + arrow_size/2, y1 - arrow_size)
            ]
            polygon_bottom = self.scene.addPolygon(QPolygonF([
                QPointF(x, y) for x, y in points_bottom]), pen)
            polygon_bottom.setBrush(QBrush(Qt.black))

        text_item = self.scene.addText(text)
        font = QFont()
        font.setPointSize(5)
        text_item.setFont(font)

        if x1 < 0:
            text_item.setPos(x1 - 10 - text_item.boundingRect().
                width(), (y1 + y2) / 2 - text_item.boundingRect().
                height() / 2)
        else:
            text_item.setPos(x1 + 15, (y1 + y2) / 2 - text_item.
                boundingRect().height() / 2)
```

Spacing Window Code Explanation

Overview

The `BoltPatternGenerator` class is a PyQt5-based graphical user interface component designed to visualize and display bolt patterns for structural steel connections. It creates a detailed 2D drawing showing the layout of bolts on a connection plate, along with all relevant dimensional parameters.

Class Structure

Class: `BoltPatternGenerator(QMainWindow)`

Inheritance: Extends `QMainWindow` from PyQt5 to create a standalone application window.

Constructor Parameters

```
def __init__(self, connection_obj, rows=3, cols=2, main=None):
```

- `connection_obj`: The connection object containing spacing and design parameters
- `rows`: Number of bolt rows (default: 3)
- `cols`: Number of bolt columns (default: 2)
- `main`: Main application object containing plate and bolt specifications

Key Instance Variables

Variable	Type	Description
`connection`	Object	Connection object with spacing calculations
`main`	Object	Main application object
`plate_height`	float	Height of the connection plate (mm)
`plate_width`	float	Width of the connection plate (mm)
`hole_dia`	float	Diameter of bolt holes (mm)
`rows`	int	Number of bolt rows
`cols`	int	Number of bolt columns
`weldsize`	float	Size of weld (mm)

Core Methods

1. `initUI()`

Purpose: Initializes the graphical user interface layout.

Layout:

- Left Panel (1/4 width): Parameter value display
- Right Panel (3/4 width): Drawing canvas (QGraphicsView)

2. get_parameters()

Purpose: Extracts and maps spacing parameters from the connection object.

Returns: Dictionary with:

- `pitch`, `end`, `gauge1`, `gauge2`, `gauge`, `edge`, `hole`

Parameter Mapping:

```
KEY_OUT_PITCH        → 'pitch'
KEY_OUT_END_DIST     → 'end'
KEY_OUT_GAUGE1       → 'gauge1'
KEY_OUT_GAUGE2       → 'gauge2'
KEY_OUT_GAUGE        → 'gauge'
KEY_OUT_EDGE_DIST    → 'edge'
```

3. createDrawing(params)

Purpose: Creates the 2D bolt pattern drawing.

Elements:

- Plate outline (rectangle)
- Bolt holes (circles)
- Weld area (red rectangle)
- Dimensions (lines and labels)

Bolt Positioning Logic:

```
x_center = plate_width - edge
for i in range(col):
    x_center -= gauge1 if i % 2 == 0 else gauge2
y_center = end + row * pitch
```

4. `addDimensions(params, pen)`

Purpose: Adds horizontal and vertical dimensions with arrowheads and text.

5. `addHorizontalDimension(...)`

Purpose: Draws horizontal dimension lines with arrows and centered text.

6. `addVerticalDimension(...)`

Purpose: Draws vertical dimension lines with arrows and centered text.

Drawing Specifications

Colors and Styles

- Plate Outline: Black pen (1.5pt)
- Bolt Holes: Blue pen (2pt)
- Weld Area: Red filled rectangle
- Dimension Lines: Black pen (1.5pt)
- Arrowheads: Black filled triangles

Dimension Offsets

- Horizontal Offset: 40 mm
- Vertical Offset: 60 mm
- Dimension Line Offset: 20 mm (horizontal), 30 mm (vertical)

Text Formatting

- Font Size: 5 pt
- Text Placement: Auto-positioned based on layout
- Precision: 1 decimal place

Usage Example

```
generator = BoltPatternGenerator(
    connection_obj=my_connection,
    main=main_application
)
generator.show()
```

Technical Dependencies

Required Modules

- `PyQt5.QtWidgets`: GUI elements
- `PyQt5.QtGui`: Pen, brush, font
- `PyQt5.QtCore`: QPointF, Qt constants

External Dependencies

- `..Common`: Constant keys for mapping
- `.additionalfns`: Utility functions
- `..design_type.connection.end_plate_connection`: Connection classes

Error Handling

- Null pointer check for `main`
- Spacing value validation from `connection`
- Graceful handling of missing parameters

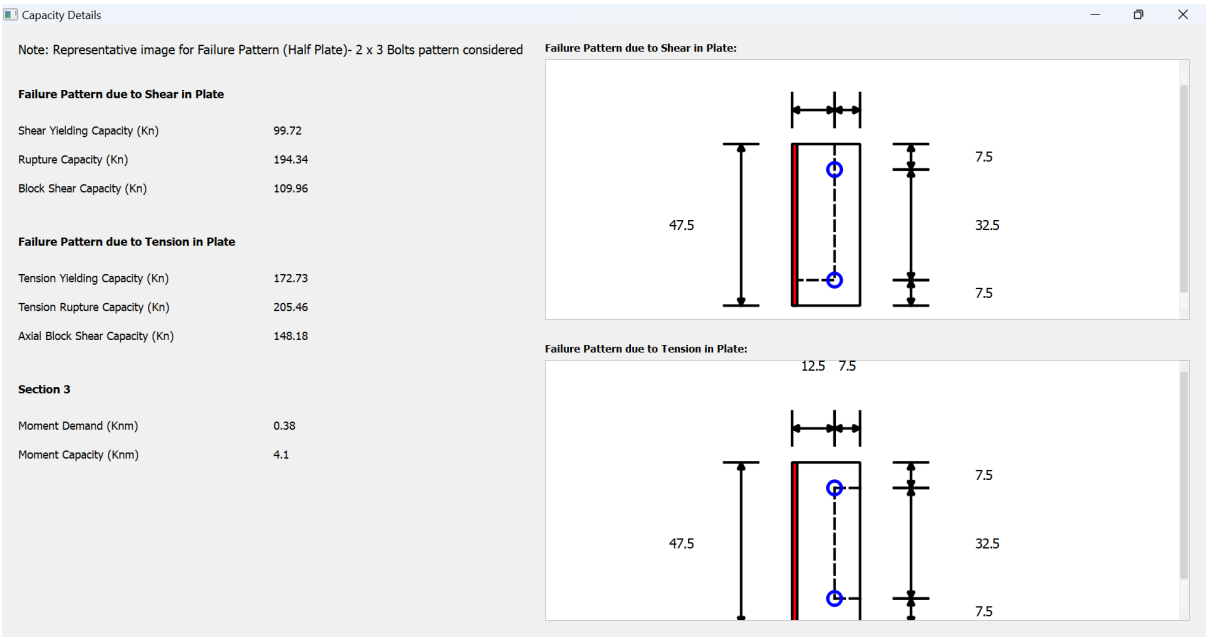
Output

This application generates a professional bolt pattern drawing including:

1. Full bolt layout
2. All dimension annotations
3. Weld locations
4. Parameter summary panel

7.1.3 Capacity Window

Capacity Window Image



Capacity Window Code

```
import sys
from PyQt5.QtWidgets import (QApplication, QMainWindow, QWidget,
    QVBoxLayout,
                             QHBoxLayout, QLabel, QGraphicsView,
                             QGraphicsScene)
from PyQt5.QtGui import QPixmap
from PyQt5.QtCore import Qt, QRectF
from PyQt5.QtGui import QPainter, QPen, QFont
from PyQt5.QtGui import QPolygonF, QBrush
```

```
from PyQt5.QtCore import QPointF
from ..Common import *
from .additionalfns import calculate_total_width
from ..design_type.connection.end_plate_connection import
   EndPlateConnection

class CapacityDetailsWindow(QMainWindow):
    def __init__(self, connection_obj, rows=3, cols=2 , main =
       None):
        super().__init__()
        self.connection = connection_obj
        self.main=main
        self.plate_height = main.plate.height
        self.plate_width = main.plate.length
        self.hole_dia=main.bolt.bolt_diameter_provided
        self.rows=main.plate.bolts_one_line
        self.cols=main.plate.bolt_line
        self.plate_thickness=main.plate.thickness
        print(self.plate_height,self.plate_width)
        output=main.output_values(main,True)
        dict1={i[0] : i[3] for i in output}

        capacity_fnc = dict1['button1'][1]
        print(capacity_fnc)
        capacity_details = capacity_fnc(main,True)
        print(capacity_details)
        details_dict={i[1]:i[3] for i in capacity_details}

        self.shear_yield_capacity=float(details_dict['Shear
           Yielding Capacity (kN)'])
        self.rupture_capacity=float(details_dict['Rupture
           Capacity (kN)'])
        self.Block_Shear_Capacity=float(details_dict['Block Shear
            Capacity (kN)'])
```

```
        self.Tension_Yielding_Capacity=float(details_dict['Tension Yielding Capacity (kN)'])
        self.Tension_rupture_Capacity=float(details_dict['Tension Rupture Capacity (kN)'])
        self.axial_block_shear_capacity=float(details_dict['Axial Block Shear Capacity (kN)'])
        self.moment_demand=float(details_dict['Moment Demand (kNm)'])
        self.moment_capacity=float(details_dict['Moment Capacity (kNm)'])
        print("
            xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
            ")
        print("
            -----------------------------------------------------------------
            ")
        self.dict_shear_failure={
            'Shear Yielding Capacity (kN)':self.shear_yield_capacity,
            'Rupture Capacity (kN)':self.rupture_capacity,
            'Block Shear Capacity (kN)':self.Block_Shear_Capacity
        }
        self.dict_tension_failure={
            'Tension Yielding Capacity (kN)':self.Tension_Yielding_Capacity,
            'Tension Rupture Capacity (kN)':self.Tension_rupture_Capacity,
            'Axial Block Shear Capacity (kN)':self.axial_block_shear_capacity
        }
        self.dict_section_3={
            'Moment Demand (kNm)':self.moment_demand,
            'Moment Capacity (kNm)':self.moment_capacity
        }
```

```
        print(self.dict_shear_failure)
        print(self.dict_tension_failure)
        print(self.dict_section_3)
        print("
           ------------------------------------------------------------------------
           ")
        print("
           xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
           ")

        for i in output:
            print(i)
        self.weldsize=0
        if 'Weld.Size' in dict1:
            self.weldsize=dict1['Weld.Size']
        self.initUI()

    def initUI(self):
        self.setWindowTitle('Bolt Pattern Generator')
        self.setGeometry(100, 100, 1650, 1050)

        # Main layout
        main_layout = QHBoxLayout()

        # Left panel for parameter display
        left_panel = QWidget()
        left_layout = QVBoxLayout()

        # Parameter display labels
        params = self.get_parameters()

        heading_label = QLabel("Note: Representative image for
           Failure Pattern (Half Plate)- 2 x 3 Bolts pattern
```

```
            considered")
        heading_label.setStyleSheet("font-size: 20px;")
        left_layout.addWidget(heading_label)

        space_label=QLabel("  ")
        space_label.setStyleSheet("font-size: 25px;")
        left_layout.addWidget(space_label)

        sub_heading_label1 = QLabel("Failure Pattern due to Shear
            in Plate")
        sub_heading_label1.setStyleSheet("font-size: 18px; font-
            weight: bold;")
        left_layout.addWidget(sub_heading_label1)

        space_label=QLabel("  ")
        space_label.setStyleSheet("font-size: 15px;")
        left_layout.addWidget(space_label)

        # Display the parameter values
        for key, value in self.dict_shear_failure.items():
            param_layout = QHBoxLayout()
            param_label = QLabel(key.title())
            value_label = QLabel(f'{value}')
            param_layout.addWidget(param_label)
            param_layout.addWidget(value_label)
            left_layout.addLayout(param_layout)

            space_label=QLabel("  ")
            space_label.setStyleSheet("font-size: 5px;")
            left_layout.addWidget(space_label)

        space_label=QLabel("  ")
        space_label.setStyleSheet("font-size: 25px;")
        left_layout.addWidget(space_label)
```

```
        sub_heading_label2 = QLabel("Failure Pattern due to
            Tension in Plate")
        sub_heading_label2.setStyleSheet("font-size: 18px; font-
            weight: bold;")
        left_layout.addWidget(sub_heading_label2)

        space_label=QLabel("  ")
        space_label.setStyleSheet("font-size: 15px;")
        left_layout.addWidget(space_label)

        for key, value in self.dict_tension_failure.items():
            param_layout = QHBoxLayout()
            param_label = QLabel(key.title())
            value_label = QLabel(f'{value}')
            param_layout.addWidget(param_label)
            param_layout.addWidget(value_label)
            left_layout.addLayout(param_layout)

            space_label=QLabel("  ")
            space_label.setStyleSheet("font-size: 5px;")
            left_layout.addWidget(space_label)

        space_label=QLabel("  ")
        space_label.setStyleSheet("font-size: 25px;")
        left_layout.addWidget(space_label)

        sub_heading_label3 = QLabel("Section 3")
        sub_heading_label3.setStyleSheet("font-size: 18px; font-
            weight: bold;")
        left_layout.addWidget(sub_heading_label3)

        space_label=QLabel("  ")
        space_label.setStyleSheet("font-size: 15px;")
```

```
        left_layout.addWidget(space_label)

        for key, value in self.dict_section_3.items():
            param_layout = QHBoxLayout()
            param_label = QLabel(key.title())
            value_label = QLabel(f'{value}')
            param_layout.addWidget(param_label)
            param_layout.addWidget(value_label)
            left_layout.addLayout(param_layout)
            space_label=QLabel("  ")
            space_label.setStyleSheet("font-size: 5px;")
            left_layout.addWidget(space_label)

        left_layout.addStretch()
        left_panel.setLayout(left_layout)

        # Right panel for the two vertical drawings
        right_panel = QWidget()
        right_layout = QVBoxLayout()
        right_panel.setLayout(right_layout)

        sub_heading_label1 = QLabel("Failure Pattern due to Shear
            in Plate:")
        sub_heading_label1.setStyleSheet("font-size: 16px; font-
           weight: bold;")
        right_layout.addWidget(sub_heading_label1)

        # First drawing
        self.scene1 = QGraphicsScene()
        self.view1 = QGraphicsView(self.scene1)
        self.view1.setRenderHint(QPainter.Antialiasing)
        self.createDrawing(self.scene1)
        self.view1.fitInView(self.scene1.sceneRect(), Qt.
           KeepAspectRatio)
```

```
        right_layout.addWidget(self.view1)


        space_label=QLabel("  ")
        space_label.setStyleSheet("font-size: 15px;")
        right_layout.addWidget(space_label)

        sub_heading_label2 = QLabel("Failure Pattern due to
            Tension in Plate:")
        sub_heading_label2.setStyleSheet("font-size: 16px; font-
            weight: bold;")
        right_layout.addWidget(sub_heading_label2)

        # Second drawing (identical to first)
        self.scene2 = QGraphicsScene()
        self.view2 = QGraphicsView(self.scene2)
        self.view2.setRenderHint(QPainter.Antialiasing)
        self.createSecondDrawing(self.scene2)  # Using the same
            drawing function
        self.view2.fitInView(self.scene2.sceneRect(), Qt.
            KeepAspectRatio)
        right_layout.addWidget(self.view2)

        main_layout.addWidget(left_panel, 1)
        main_layout.addWidget(right_panel, 3)

        main_widget = QWidget()
        main_widget.setLayout(main_layout)
        self.setCentralWidget(main_widget)

    def get_parameters(self):
        spacing_data = self.connection.spacing(status=True)  #
            Get actual values
        param_map = {}
```

```
        print('spacing_data length' , len(spacing_data))
        for item in spacing_data:
            key, _, _, value = item
            if key == KEY_OUT_PITCH:
                param_map['pitch'] = float(value)
            elif key == KEY_OUT_END_DIST:
                param_map['end'] = float(value)
            elif key == KEY_OUT_GAUGE1:
                param_map['gauge1'] = float(value)
            elif key == KEY_OUT_GAUGE2:
                param_map['gauge2'] = float(value)
            elif key == KEY_OUT_GAUGE:
                param_map['gauge'] = float(value)
            elif key == KEY_OUT_EDGE_DIST:
                param_map['edge'] = float(value)

        # Add hardcoded hole diameter
        param_map['hole'] = self.main.bolt.bolt_diameter_provided

        print("Extracted parameters:", param_map)
        return param_map


# failure due to shear in plate
    def createDrawing(self, scene):
        coeff = 2  # scaling coefficient
        params = self.get_parameters()
        pitch = params['pitch'] / coeff
        end = params['end'] / coeff
        if 'gauge' in params:
            gauge1 = gauge2 = params['gauge'] / coeff
        else:
            gauge1 = params['gauge1'] / coeff
            gauge2 = params['gauge2'] / coeff
```

```
        edge = params['edge'] / coeff
        width = self.plate_width / coeff
        height = self.plate_height / coeff
        hole_diameter = params['hole'] / coeff
        weld_size = self.weldsize / coeff

        outline_pen = QPen(Qt.blue, 2/coeff)
        dimension_pen = QPen(Qt.black, 1.5/coeff)
        red_brush = QBrush(Qt.red)

        # Create dashed pen for failure patterns
        dashed_pen = QPen(Qt.black, 1.5/coeff, Qt.DashLine)



        h_offset = 40 / coeff
        v_offset = 60 / coeff
        scene.setSceneRect(-h_offset, -v_offset, width + 2*
           v_offset, height + 2*h_offset)
        #adding the shear failure pattern
        scene.addLine(width-end, 0, width-end, height-edge,
           dashed_pen)
        scene.addLine(0, height-edge, width-end, height-edge,
           dashed_pen)
        scene.addRect(0, 0, width, height, dimension_pen)

        # Draw holes
        for row in range(self.rows):
            for col in range(self.cols):
                x_center = width - edge
                for i in range(col):
                    x_center -= gauge1 if i % 2 == 0 else gauge2
                y_center = end + row * pitch
                x = x_center - hole_diameter / 2
```

```
                y = y_center - hole_diameter / 2
                scene.addEllipse(x, y, hole_diameter,
                    hole_diameter, outline_pen)
        # Draw weld area
        if weld_size > 0:
            scene.addRect(0, 0, weld_size, height, dimension_pen,
                red_brush)
        # Add dimensions
        self.addDimensions(scene, width, height, pitch, end,
            gauge1, gauge2, edge, dimension_pen, coeff)


    def createSecondDrawing(self, scene):
        coeff = 2  # scaling coefficient
        params = self.get_parameters()
        pitch = params['pitch'] / coeff
        end = params['end'] / coeff
        if 'gauge' in params:
            gauge1 = gauge2 = params['gauge'] / coeff
        else:
            gauge1 = params['gauge1'] / coeff
            gauge2 = params['gauge2'] / coeff
        edge = params['edge'] / coeff
        width = self.plate_width / coeff
        height = self.plate_height / coeff
        hole_diameter = params['hole'] / coeff
        weld_size = self.weldsize / coeff

        outline_pen = QPen(Qt.blue, 2/coeff)
        dimension_pen = QPen(Qt.black, 1.5/coeff)
        red_brush = QBrush(Qt.red)

        # Create dashed pen for failure patterns
        dashed_pen = QPen(Qt.black, 1.5/coeff, Qt.DashLine)
```

```
if self.cols==1:
    x_line_dist=width-end
else:
    x_line_dist=width-end - (self.cols-1)*gauge1

h_offset = 40 / coeff
v_offset = 60 / coeff
scene.setSceneRect(-h_offset, -v_offset, width + 2*
   v_offset, height + 2*h_offset)
#adding the tension failure pattern
scene.addLine(x_line_dist, edge, width, edge, dashed_pen)
scene.addLine(x_line_dist, edge, x_line_dist, height-edge
   , dashed_pen)
scene.addLine(x_line_dist, height-edge, width, height-
   edge, dashed_pen)
scene.addRect(0, 0, width, height, dimension_pen)


# Draw holes
for row in range(self.rows):
    for col in range(self.cols):
        x_center = width - edge
        for i in range(col):
            x_center -= gauge1 if i % 2 == 0 else gauge2
        y_center = end + row * pitch
        x = x_center - hole_diameter / 2
        y = y_center - hole_diameter / 2
        scene.addEllipse(x, y, hole_diameter,
           hole_diameter, outline_pen)
# Draw weld area
```

```
        if weld_size > 0:
            scene.addRect(0, 0, weld_size, height, dimension_pen,
                red_brush)
        # Add dimensions
        self.addDimensions(scene, width, height, pitch, end,
           gauge1, gauge2, edge, dimension_pen, coeff)



    def addDimensions(self, scene, width, height, pitch, end,
       gauge1, gauge2, edge, pen, coeff):
        h_offset = 20 / coeff
        v_offset = 30 / coeff
        x_start = width
        segments = []
        segments.append(('edge', x_start-edge, x_start))
        x_start -= edge
        segments.append(('edge', 0, x_start))
        for label, x1, x2 in segments:
            value = x2 - x1
            self.addHorizontalDimension(scene, x1, -h_offset, x2,
                -h_offset, f"{value:.1f}", pen)
        # Add vertical dimensions
        self.addVerticalDimension(scene, width + v_offset, 0,
           width + v_offset, end, str(end), pen)
        for i in range(self.rows - 1):
            self.addVerticalDimension(scene, width + v_offset,
               end + i * pitch,
                                      width + v_offset, end + (i +
                                          1) * pitch, str(pitch),
                                         pen)
        self.addVerticalDimension(scene, width + v_offset, height
           , width + v_offset, height - end, str(end), pen)
        total_height = 2 * end + (self.rows - 1) * pitch
```

```
        self.addVerticalDimension(scene, -v_offset, 0, -v_offset,
            total_height, str(total_height), pen)

    def addHorizontalDimension(self, scene, x1, y1, x2, y2, text,
        pen):
        scene.addLine(x1, y1, x2, y2, pen)
        arrow_size = 2
        ext_length = 10
        scene.addLine(x1, y1 - ext_length/2, x1, y1 + ext_length
            /2, pen)
        scene.addLine(x2, y2 - ext_length/2, x2, y2 + ext_length
            /2, pen)

        points_left = [
            (x1, y1),
            (x1 + arrow_size, y1 - arrow_size/2),
            (x1 + arrow_size, y1 + arrow_size/2)
        ]
        polygon_left = scene.addPolygon(QPolygonF([QPointF(x, y)
            for x, y in points_left]), pen)
        polygon_left.setBrush(QBrush(Qt.black))

        points_right = [
            (x2, y2),
            (x2 - arrow_size, y2 - arrow_size/2),
            (x2 - arrow_size, y2 + arrow_size/2)
        ]
        polygon_right = scene.addPolygon(QPolygonF([QPointF(x, y)
            for x, y in points_right]), pen)
        polygon_right.setBrush(QBrush(Qt.black))

        text_item = scene.addText(text)
        font = QFont()
        font.setPointSize(2)
```

```
        text_item.setFont(font)

        if y1 < 0:
            text_item.setPos((x1 + x2) / 2 - text_item.
                boundingRect().width() / 2, y1 - 25)
        else:
            text_item.setPos((x1 + x2) / 2 - text_item.
                boundingRect().width() / 2, y1 + 5)

    def addVerticalDimension(self, scene, x1, y1, x2, y2, text,
        pen):
        scene.addLine(x1, y1, x2, y2, pen)
        arrow_size = 2
        ext_length = 10
        scene.addLine(x1 - ext_length/2, y1, x1 + ext_length/2,
            y1, pen)
        scene.addLine(x2 - ext_length/2, y2, x2 + ext_length/2,
            y2, pen)

        if y2 > y1:
            points_top = [
                (x1, y1),
                (x1 - arrow_size/2, y1 + arrow_size),
                (x1 + arrow_size/2, y1 + arrow_size)
            ]
            polygon_top = scene.addPolygon(QPolygonF([QPointF(x,
                y) for x, y in points_top]), pen)
            polygon_top.setBrush(QBrush(Qt.black))

            points_bottom = [
                (x2, y2),
                (x2 - arrow_size/2, y2 - arrow_size),
                (x2 + arrow_size/2, y2 - arrow_size)
            ]
```

```
            polygon_bottom = scene.addPolygon(QPolygonF([QPointF(
                x, y) for x, y in points_bottom]), pen)
            polygon_bottom.setBrush(QBrush(Qt.black))
        else:
            points_top = [
                (x2, y2),
                (x2 - arrow_size/2, y2 + arrow_size),
                (x2 + arrow_size/2, y2 + arrow_size)
            ]
            polygon_top = scene.addPolygon(QPolygonF([QPointF(x,
                y) for x, y in points_top]), pen)
            polygon_top.setBrush(QBrush(Qt.black))

            points_bottom = [
                (x1, y1),
                (x1 - arrow_size/2, y1 - arrow_size),
                (x1 + arrow_size/2, y1 - arrow_size)
            ]
            polygon_bottom = scene.addPolygon(QPolygonF([QPointF(
                x, y) for x, y in points_bottom]), pen)
            polygon_bottom.setBrush(QBrush(Qt.black))

        text_item = scene.addText(text)
        font = QFont()
        font.setPointSize(2)
        text_item.setFont(font)

        if x1 < 0:
            text_item.setPos(x1 - 10 - text_item.boundingRect().
                width(), (y1 + y2) / 2 - text_item.boundingRect().
                height() / 2)
        else:
            text_item.setPos(x1 + 15, (y1 + y2) / 2 - text_item.
                boundingRect().height() / 2)
```

Capacity Window Code Explanation

Overview

The `CapacityDetailsWindow` class is a PyQt5-based graphical user interface component designed to visualize and display capacity analysis results for fin plate connections in structural steel design. This module provides a comprehensive visualization tool that shows failure patterns and capacity values for both shear and tension failure modes.

Class Structure

Class: `CapacityDetailsWindow`

Inheritance: `QMainWindow` (PyQt5)

Purpose: Creates a window displaying capacity analysis results with visual representations of failure patterns for fin plate connections.

Constructor Parameters

```
def __init__(self, connection_obj, rows=3, cols=2, main=None)
```

- `connection`$_obj$: $Connectionobjectcontainingspacingandgeometricdata$`rows` : $Numberofboltrou$ 3)
- `cols`: Number of bolt columns (default: 2)
- `main`: Main application object containing plate and bolt specifications

Important Instance Variables

Geometric Properties

- `plate_height`, `plate_width`, `plate_thickness`
- `hole_dia`, `rows`, `cols`, `weldsize`

Capacity Values

- `shear_yield_capacity`, `rupture_capacity`, `Block_Shear_Capacity`
- `Tension_Yielding_Capacity`, `Tension_rupture_Capacity`
- `axial_block_shear_capacity`, `moment_demand`, `moment_capacity`

Data Dictionaries

- `dict_shear_failure`, `dict_tension_failure`, `dict_section_3`

Key Methods

1. `initUI()`

Purpose: Initializes the user interface layout and components.

Layout Structure:

- Left Panel: Displays capacity values and failure mode information
- Right Panel: Contains two graphical views showing failure patterns

UI Components:

- Parameter display labels with capacity values
- Two QGraphicsView widgets
- Organized sections: shear failure, tension failure, moment analysis

2. `get_parameters()`

Purpose: Extracts geometric parameters from the connection object.

Returns: Dictionary with `pitch`, `end`, `gauge1`, `gauge2`, `edge`, `hole`

3. createDrawing(scene)

Purpose: Creates the first graphical representation showing shear failure pattern.

Features:

- Scaling coefficient = 2
- Dashed lines for shear failure
- Circular holes, red weld areas, and dimensions

Example Logic:

```
scene.addLine(width-end, 0, width-end, height-edge, dashed_pen)
scene.addLine(0, height-edge, width-end, height-edge, dashed_pen)
```

4. createSecondDrawing(scene)

Purpose: Visualizes tension failure pattern.

Features: Similar to `createDrawing()` with adaptive failure pattern.

Example Logic:

```
if self.cols == 1:
    x_line_dist = width - end
else:
    x_line_dist = width - end - (self.cols-1) * gauge1
```

5. addDimensions(scene, ...)

Purpose: Adds dimension lines and annotations.

Features: Horizontal and vertical dimensions, positioned outside drawing area.

6. addHorizontalDimension(...)

and `addVerticalDimension(...)` **Purpose:** Adds dimension lines with arrowheads and labels.

Components: Lines, arrowheads, and centered text.

Technical Implementation Details

Scaling System

- Coefficient: 2
- Purpose: Fit drawing to viewport while preserving scale

Color Scheme

- Blue: Bolt holes
- Red: Welds
- Black: Dimensions
- Dashed Black: Failure lines

Coordinate System

- Origin: Top-left
- X: Right, Y: Down
- Units: mm (scaled)

Usage in Documentation

This module can be used for:

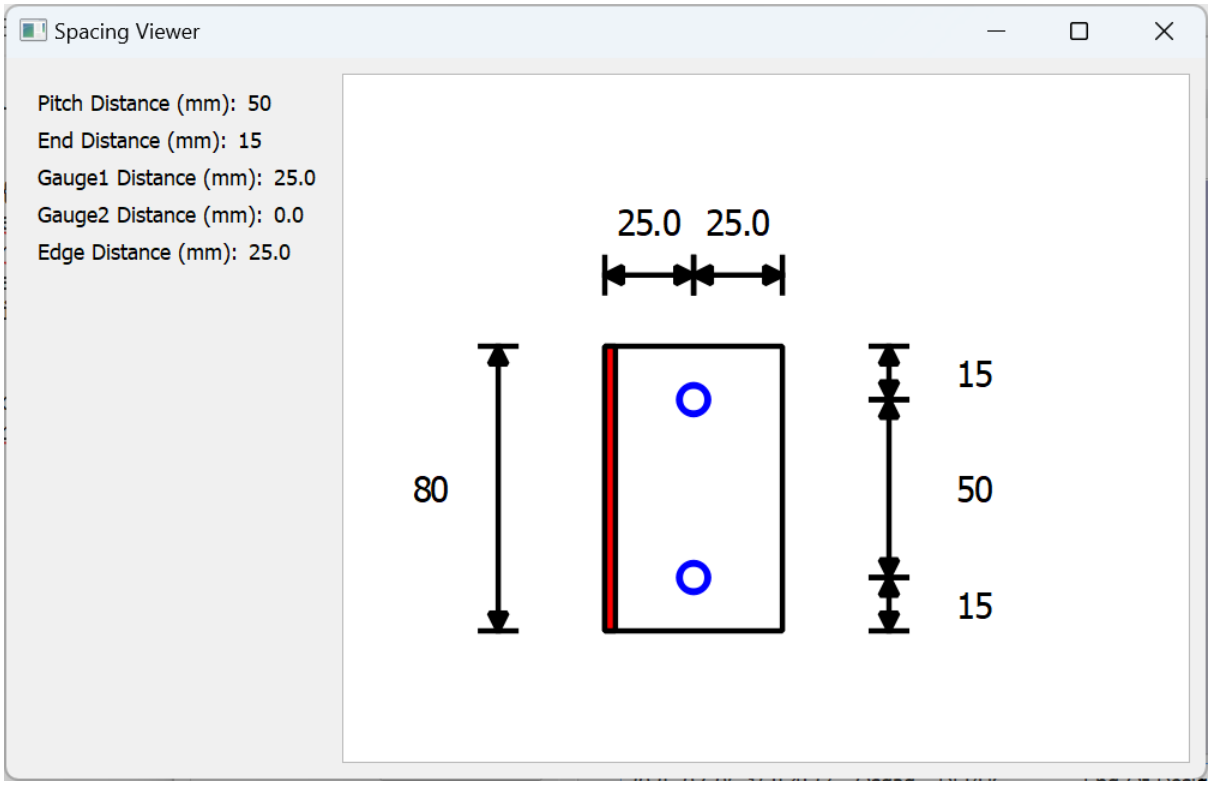
1. Verifying design calculations
2. Understanding failure modes
3. Ensuring correct geometry and spacing
4. Educational demonstrations

Dependencies

- `PyQt5`: For GUI and graphics
- `Common Module`: Shared constants
- `EndPlateConnection`: Analysis logic

7.2 Shear Connection - Cleat Angle

7.2.1 Spacing Window Image



7.2.2 Spacing Window Code

```
import sys
from PyQt5.QtWidgets import (QApplication, QMainWindow, QWidget,
    QVBoxLayout,
                             QHBoxLayout, QLabel, QGraphicsView,
                             QGraphicsScene)
```

```
from PyQt5.QtGui import QPixmap
from PyQt5.QtCore import Qt, QRectF
from PyQt5.QtGui import QPainter, QPen, QFont
from PyQt5.QtGui import QPolygonF, QBrush
from PyQt5.QtCore import QPointF
from ..Common import *
from .additionalfns import calculate_total_width
class CleatAngle(QMainWindow):
    def __init__(self, connection_obj, rows=3, cols=2 , main =
        None):
        super().__init__()
        (main,self.flag)=main
        spacing_data = connection_obj.spacing(status=True)
        output=connection_obj.output_values(True)
        data1={ f'{i[1]} + {i[0]}' : i[3] for i in output}

        self.angle_thickness=float(data1['Cleat Angle Designation
            + Cleat.Angle'].split(" ")[-1])

        params= {
            'width' : int(data1['Height (mm) + Plate.Height']),
            'hole' : int(data1['Diameter (mm) + Bolt.Diameter']),
        }
        if self.flag==0:
            params['length']=int(data1['Cleat Angle Designation +
                Cleat.Angle'][0:2])
            params['rows']=int(data1['Bolt Rows (nos) + Bolt.
               OneLine'])
            params['cols']=int(data1['Bolt Columns (nos) + Bolt.
               Line'])
        else:
            params['length']=int(data1['Cleat Angle Designation +
                Cleat.Angle'][5:7])
```

```
            params['rows']=int(data1['Bolt Rows (nos) + Cleat.
               Spting_leg.OneLine'])
            params['cols']=int(data1['Bolt Columns (nos) + Cleat.
               Spting_leg.Line'])
        for i in spacing_data:
            print(i)
        for item in spacing_data:
            if not isinstance(item[0], str):
                continue
            key = item[0].lower()
            value = item[3]

            if 'pitch' in key:
                params['pitch'] = value
            elif 'gauge1' in key:
                params['gauge1'] = value
            elif 'gauge2' in key:
                params['gauge2'] = value
            elif 'end' in key:
                params['end']=value
            elif 'edge' in key:
                params['edge']=value
        for i in params:
            print(f'{i} : {params[i]}')
        self.params=params
        self.initUI()
    def initUI(self):
        self.setWindowTitle('Bolt Pattern Generator')
        self.setGeometry(100, 100, 1000, 600)

        # Main layout
        main_layout = QHBoxLayout()

        # Left panel for parameter display
```

```
        left_panel = QWidget()
        left_layout = QVBoxLayout()

        # Parameter display labels
        params = self.params

        # Display the parameter values
        for key, value in params.items():
            if key=='cols' or key=='rows' or key=='hole' or key==
               'length' or key=='width':
                continue
            param_layout = QHBoxLayout()
            param_label = QLabel(f'{key.title()} Distance (mm):')
            value_label = QLabel(f'{value}')
            param_layout.addWidget(param_label)
            param_layout.addWidget(value_label)
            left_layout.addLayout(param_layout)

        left_layout.addStretch()
        left_panel.setLayout(left_layout)

        # Right panel for the drawing using QGraphicsView
        self.scene = QGraphicsScene()
        self.view = QGraphicsView(self.scene)
        self.view.setRenderHint(QPainter.Antialiasing)

        # Create and add the drawing to the scene
        self.createDrawing(params)

        # Add panels to main layout
        main_layout.addWidget(left_panel, 1)
        main_layout.addWidget(self.view, 3)

        # Set main widget
```

```
        main_widget = QWidget()
        main_widget.setLayout(main_layout)
        self.setCentralWidget(main_widget)

        # Ensure the view shows all content
        self.view.fitInView(self.scene.sceneRect(), Qt.
            KeepAspectRatio)

    def createDrawing(self, params):

        # Extract parameters
        pitch = params['pitch']
        end = params['end']
        if 'gauge' in params:
            gauge = params['gauge']
        else:
            gauge1 = params['gauge1']
            gauge2 = params['gauge2']
        edge = params['edge']
        hole_diameter = params['hole']
        self.rows=params['rows']
        self.cols=params['cols']

        # Calculate dimensions
        if 'gauge' in params:
            gauge1 = gauge
            gauge2 = gauge
        width = params['length']

        height = params['width']
        self.plate_width=width
        self.plate_height=height
        # Set up pens
        outline_pen = QPen(Qt.blue, 2)
```

```
        dimension_pen = QPen(Qt.black, 1.5)
        angle_pen = QBrush(Qt.red)

        # Dimension offsets
        h_offset = 40
        v_offset = 60

        # Create scene rectangle with extra space for dimensions
        self.scene.setSceneRect(-h_offset, -v_offset,
                                width + 2*v_offset, height + 2*
                                   h_offset)

        # Draw rectangle
        self.scene.addRect(0, 0, width, height, dimension_pen)
        self.scene.addRect(0, 0, self.angle_thickness, height,
           dimension_pen, angle_pen)


        # Draw holes
        for row in range(self.rows):
            for col in range(self.cols):
                # Start from right edge (for example: total plate
                    width - edge)
                x_center = self.plate_width - edge

                # Subtract gauges from right to left
                for i in range(col):
                    x_center -= gauge1 if i % 2 == 0 else gauge2

                # Y-position stays the same
                y_center = end + row * pitch

                # Top-left corner for drawing the circle
                x = x_center - hole_diameter / 2
```

```
                y = y_center - hole_diameter / 2

                print(f"row: {row}, col: {col}, x: {x}, y: {y}")
                self.scene.addEllipse(x, y, hole_diameter,
                    hole_diameter, outline_pen)

        print(params,dimension_pen)
        # Add dimensions
        self.addDimensions(params, dimension_pen)

    def addDimensions(self, params, pen):
        # Extract parameters
        pitch = params['pitch']
        end = params['end']
        if 'gauge' in params:
            gauge = params['gauge']
        else:
            gauge1 = params['gauge1']
            gauge2 = params['gauge2']
        edge = params['edge']

        if 'gauge' in params:
            gauge1 = gauge
            gauge2 = gauge

        width = self.plate_width
        height = self.plate_height

        # Offsets for dimension lines
        h_offset = 20
        v_offset = 30

        # Add horizontal dimensions
        x_start = width
```

```
        segments = []
        # First edge
        segments.append(('edge', x_start-edge, x_start ))
        x_start -=edge

        # Last edge
        segments.append(('edge', 0, x_start))

        # Draw each segment
        for label, x1, x2 in segments:
            value = x2 - x1
            self.addHorizontalDimension(x1, -h_offset, x2, -
               h_offset, f"{value:.1f}", pen)
        # Add vertical dimensions
        self.addVerticalDimension(width + v_offset, 0, width +
           v_offset, end, str(end), pen)
        for i in range(self.rows - 1):
            self.addVerticalDimension(width + v_offset, end + i *
                pitch, width + v_offset, end + (i + 1) * pitch,
               str(pitch), pen)

        # Add bottom end distance dimension
        self.addVerticalDimension(width + v_offset, height, width
            + v_offset, height - end, str(end), pen)

        # Add left side dimension
        total_height = 2 * end + (self.rows - 1) * pitch
        self.addVerticalDimension(-v_offset, 0, -v_offset,
           total_height, str(total_height), pen)

    def addHorizontalDimension(self, x1, y1, x2, y2, text, pen):
        self.scene.addLine(x1, y1, x2, y2, pen)
        arrow_size = 5
        ext_length = 10
```

```
        self.scene.addLine(x1, y1 - ext_length/2, x1, y1 +
            ext_length/2, pen)
        self.scene.addLine(x2, y2 - ext_length/2, x2, y2 +
            ext_length/2, pen)

        points_left = [
            (x1, y1),
            (x1 + arrow_size, y1 - arrow_size/2),
            (x1 + arrow_size, y1 + arrow_size/2)
        ]
        polygon_left = self.scene.addPolygon(QPolygonF([QPointF(x
            , y) for x, y in points_left]), pen)
        polygon_left.setBrush(QBrush(Qt.black))

        points_right = [
            (x2, y2),
            (x2 - arrow_size, y2 - arrow_size/2),
            (x2 - arrow_size, y2 + arrow_size/2)
        ]
        polygon_right = self.scene.addPolygon(QPolygonF([QPointF(
            x, y) for x, y in points_right]), pen)
        polygon_right.setBrush(QBrush(Qt.black))

        text_item = self.scene.addText(text)
        font = QFont()
        font.setPointSize(5)
        text_item.setFont(font)

        if y1 < 0:
            text_item.setPos((x1 + x2) / 2 - text_item.
                boundingRect().width() / 2, y1 - 25)
        else:
            text_item.setPos((x1 + x2) / 2 - text_item.
                boundingRect().width() / 2, y1 + 5)
```

```
    def addVerticalDimension(self, x1, y1, x2, y2, text, pen):
        self.scene.addLine(x1, y1, x2, y2, pen)
        arrow_size = 5
        ext_length = 10
        self.scene.addLine(x1 - ext_length/2, y1, x1 + ext_length
           /2, y1, pen)
        self.scene.addLine(x2 - ext_length/2, y2, x2 + ext_length
           /2, y2, pen)

        if y2 > y1:
            points_top = [
                (x1, y1),
                (x1 - arrow_size/2, y1 + arrow_size),
                (x1 + arrow_size/2, y1 + arrow_size)
            ]
            polygon_top = self.scene.addPolygon(QPolygonF([
               QPointF(x, y) for x, y in points_top]), pen)
            polygon_top.setBrush(QBrush(Qt.black))

            points_bottom = [
                (x2, y2),
                (x2 - arrow_size/2, y2 - arrow_size),
                (x2 + arrow_size/2, y2 - arrow_size)
            ]
            polygon_bottom = self.scene.addPolygon(QPolygonF([
               QPointF(x, y) for x, y in points_bottom]), pen)
            polygon_bottom.setBrush(QBrush(Qt.black))
        else:
            points_top = [
                (x2, y2),
                (x2 - arrow_size/2, y2 + arrow_size),
                (x2 + arrow_size/2, y2 + arrow_size)
            ]
```

```
            polygon_top = self.scene.addPolygon(QPolygonF([
                QPointF(x, y) for x, y in points_top]), pen)
            polygon_top.setBrush(QBrush(Qt.black))

            points_bottom = [
                (x1, y1),
                (x1 - arrow_size/2, y1 - arrow_size),
                (x1 + arrow_size/2, y1 - arrow_size)
            ]
            polygon_bottom = self.scene.addPolygon(QPolygonF([
                QPointF(x, y) for x, y in points_bottom]), pen)
            polygon_bottom.setBrush(QBrush(Qt.black))

        text_item = self.scene.addText(text)
        font = QFont()
        font.setPointSize(5)
        text_item.setFont(font)

        if x1 < 0:
            text_item.setPos(x1 - 10 - text_item.boundingRect().
                width(), (y1 + y2) / 2 - text_item.boundingRect().
                height() / 2)
        else:
            text_item.setPos(x1 + 15, (y1 + y2) / 2 - text_item.
                boundingRect().height() / 2)
```

7.2.3 Spacing Window Code Explanation

Module Overview: `cleatangledetailing.py`

The `cleatangledetailing.py` module is a PyQt5-based graphical interface designed for generating and visualizing bolt patterns for cleat angle connections in structural steel design. It produces detailed 2D drawings with dimension annotations, suitable for engineering documentation.

Class: `CleatAngle`

Purpose

The `CleatAngle` class extends `QMainWindow` to provide a specialized GUI window displaying cleat angle connection details including bolt patterns, dimensions, and specifications.

Constructor

```
def __init__(self, connection_obj, rows=3, cols=2, main=None)
```

Parameters:

- `connection_obj`: Design object containing spacing data
- `rows`: Number of bolt rows (default: 3)
- `cols`: Number of bolt columns (default: 2)
- `main`: Tuple containing main window reference and flag (default: None)

Key Initialization Steps:

- Extract spacing and geometric parameters
- Determine cleat leg to detail (main or spring leg) via `self.flag`
- Parse designation to extract angle thickness
- Populate `self.params` with relevant geometric values

Method Descriptions

`initUI(self)`

Purpose: Initializes and sets up the GUI layout.

Layout Structure:

- Left Panel: Parameter display in vertical layout
- Right Panel: Drawing canvas via `QGraphicsView`

`createDrawing(self, params)`

Purpose: Renders the cleat angle with bolt patterns and dimensions.

Drawing Components:

- Cleat plate boundary (black rectangle)
- Red-filled angle thickness strip
- Bolt holes (blue circles)
- Dimension annotations

Positioning Logic:

```
x_center = plate_width - edge
for i in range(col):
    x_center -= gauge1 if i % 2 == 0 else gauge2
y_center = end + row * pitch
```

`addDimensions(self, params, pen)`

Purpose: Adds dimension lines and labels to the drawing.

Features:

- Horizontal dimensions: edge and gauge distances
- Vertical dimensions: end distances, pitch spacing, total height

`addHorizontalDimension(...)`

Purpose: Draws horizontal dimension lines with arrows and text.

Components:

- Line with two arrowheads
- Extension lines at endpoints
- Centered text label

`addVerticalDimension(...)`

Purpose: Adds vertical dimensions with styling and text.

Features:

- Arrowheads with correct direction
- Smart label placement based on position

Important Variables

- `self.params`: Parameter dictionary for geometry and spacing
- `self.angle_thickness`: Thickness of the cleat angle leg
- `self.rows`, `self.cols`: Bolt layout dimensions
- `self.scene`, `self.view`: Drawing scene and display widget
- `self.plate_width`, `self.plate_height`: Plate dimensions
- `self.flag`: Indicates whether to draw main or spring leg

Key Parameters

- `pitch`: Bolt row spacing (vertical)
- `gauge1`, `gauge2`: Bolt column spacing (horizontal)
- `edge`: Distance from plate edge to bolt hole
- `end`: Top/bottom distance from bolt hole to edge
- `hole`: Diameter of bolt holes
- `length`: Length of cleat angle
- `width`: Height of cleat angle

Technical Features

Drawing Capabilities

- Accurate dimensioning in millimeters
- Engineering-compliant visuals and linework
- Automatic scaling with preserved aspect ratio

User Interface

- Clean, professional two-panel layout
- Real-time parameter visualization
- Supports different cleat configurations (main/spring leg)

Data Integration

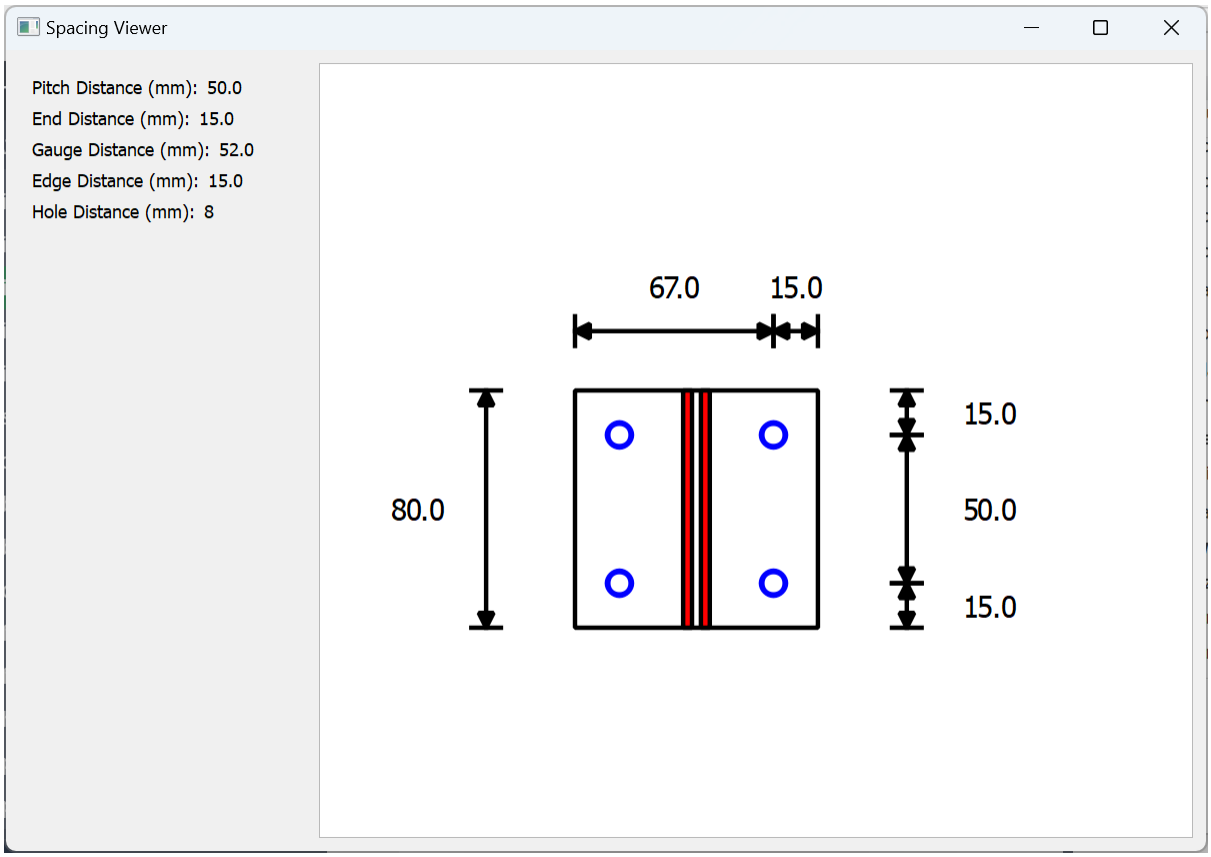
- Reads directly from engineering connection objects
- Auto-updates UI based on internal state

Conclusion

The `CleatAngle` class provides engineers with a powerful tool for generating standardized, annotated cleat angle drawings. With clear visualization of bolt patterns and dimensions, it is suitable for use in design validation, technical documentation, and fabrication workflows.

7.3 Shear Connection - End Plate

7.3.1 Spacing Window Image



7.3.2 Spacing Window Code

```
import sys
from PyQt5.QtWidgets import (QApplication, QMainWindow, QWidget,
   QVBoxLayout,
                             QHBoxLayout, QLabel, QGraphicsView,
                             QGraphicsScene)
from PyQt5.QtGui import QPixmap
from PyQt5.QtCore import Qt, QRectF
from PyQt5.QtGui import QPainter, QPen, QFont
from PyQt5.QtGui import QPolygonF, QBrush
from PyQt5.QtCore import QPointF
from ..Common import *
```

```
from .additionalfns import calculate_total_width
from ..design_type.connection.end_plate_connection import
   EndPlateConnection

class EndPlateDetailer(QMainWindow):
    def __init__(self, connection_obj, rows=3, cols=2 , main =
       None):
        super().__init__()
        self.connection = connection_obj
        self.main=main
        output=main.output_values(main,True)
        dict1={i[0] : i[3] for i in output}
        self.plate_height = dict1['Plate.Height']
        self.plate_width = dict1['Plate.Length']
        self.hole_dia=dict1['Bolt.Diameter']
        self.rows=dict1['Bolt.Rows']
        self.cols=main.plate.bolt_line
        print(self.cols)
        for i in output:
            print(i)
        self.weldsize=0
        if 'Weld.Size' in dict1:
            self.weldsize=dict1['Weld.Size']
        print(main.supported_section.web_thickness)
        self.weldgap=main.supported_section.web_thickness
        self.initUI()
        # print(self.connection.spacing(status=True))
    def initUI(self):
        self.setWindowTitle('Bolt Pattern Generator')
        self.setGeometry(100, 100, 1200, 800)

        # Main layout
        main_layout = QHBoxLayout()
```

```
        # Left panel for parameter display
        left_panel = QWidget()
        left_layout = QVBoxLayout()

        # Parameter display labels
        params = self.get_parameters()

        # Display the parameter values
        for key, value in params.items():
            param_layout = QHBoxLayout()
            param_label = QLabel(f'{key.title()} Distance (mm):')
            value_label = QLabel(f'{value}')
            param_layout.addWidget(param_label)
            param_layout.addWidget(value_label)
            left_layout.addLayout(param_layout)

        left_layout.addStretch()
        left_panel.setLayout(left_layout)

        # Right panel for the drawing using QGraphicsView
        self.scene = QGraphicsScene()
        self.view = QGraphicsView(self.scene)
        self.view.setRenderHint(QPainter.Antialiasing)

        # Create and add the drawing to the scene
        self.createDrawing(params)

        # Add panels to main layout
        main_layout.addWidget(left_panel, 1)
        main_layout.addWidget(self.view, 3)

        # Set main widget
        main_widget = QWidget()
        main_widget.setLayout(main_layout)
```

```
        self.setCentralWidget(main_widget)

        # Ensure the view shows all content
        self.view.fitInView(self.scene.sceneRect(), Qt.
            KeepAspectRatio)
    def get_parameters(self):
        spacing_data = self.connection.spacing(status=True)  #
            Get actual values
        param_map = {}
        print('spacing_data length' , len(spacing_data))
        for item in spacing_data:
            key, _, _, value = item
            # print('key : ', key)
            if key == KEY_OUT_PITCH:
                param_map['pitch'] = float(value)
            elif key == KEY_OUT_END_DIST:
                param_map['end'] = float(value)
            elif key == KEY_OUT_GAUGE1:
                param_map['gauge1'] = float(value)
            elif key == KEY_OUT_GAUGE2:
                param_map['gauge2'] = float(value)
            elif key == KEY_OUT_GAUGE:
                param_map['gauge'] = float(value)
            elif key == KEY_OUT_EDGE_DIST:
                param_map['edge'] = float(value)

        # Add hardcoded hole diameter
        param_map['hole'] = self.main.bolt.bolt_diameter_provided

        print("Extracted parameters:", param_map)

        return param_map

    def createDrawing(self, params):
```

```
        # Extract parameters
        pitch = params['pitch']
        end = params['end']
        if 'gauge' in params:
            gauge = params['gauge']
        else:
            gauge1 = params['gauge1']
            gauge2 = params['gauge2']
        edge = params['edge']
        hole_diameter = params['hole']

        # Calculate dimensions
        if 'gauge' in params:
            gauge1 = gauge
            gauge2 = gauge
        width = self.plate_width

        height = self.plate_height

        # Set up pens
        outline_pen = QPen(Qt.blue, 2)
        dimension_pen = QPen(Qt.black, 1.5)
        red_brush = QBrush(Qt.red)

        # Dimension offsets
        h_offset = 40
        v_offset = 60

        # Create scene rectangle with extra space for dimensions
        self.scene.setSceneRect(-h_offset, -v_offset,
                                width + 2*v_offset, height + 2*
                                   h_offset)
```

```
        # Draw rectangle
        self.scene.addRect(0, 0, width, height, dimension_pen)

        # Draw holes
        for row in range(self.rows):
            for col in range(self.cols):
                # Start from right edge (for example: total plate
                    width - edge)
                x_center = self.plate_width - edge

                # Subtract gauges from right to left
                for i in range(col):
                    x_center -= gauge1 if i % 2 == 0 else gauge2

                # Y-position stays the same
                y_center = end + row * pitch

                # Top-left corner for drawing the circle
                x = x_center - hole_diameter / 2
                y = y_center - hole_diameter / 2

                print(f"row: {row}, col: {col}, x: {x}, y: {y}")
                self.scene.addEllipse(x, y, hole_diameter,
                   hole_diameter, outline_pen)
        weld_size=self.weldsize
        weld_gap=self.weldgap
        x_center=self.plate_width/2
        y_center=self.plate_height/2
        self.scene.addRect(x_center-weld_gap/2-weld_size, 0,
           weld_size, height, dimension_pen,red_brush)
        self.scene.addRect(x_center+weld_gap/2, 0, weld_size,
           height, dimension_pen,red_brush)
        print(params,dimension_pen)
        # Add dimensions
```

```
        self.addDimensions(params, dimension_pen)

    def addDimensions(self, params, pen):
        # Extract parameters
        pitch = params['pitch']
        end = params['end']
        if 'gauge' in params:
            gauge = params['gauge']
        else:
            gauge1 = params['gauge1']
            gauge2 = params['gauge2']
        edge = params['edge']

        if 'gauge' in params:
            gauge1 = gauge
            gauge2 = gauge

        width = self.plate_width
        height = self.plate_height

        # Offsets for dimension lines
        h_offset = 20
        v_offset = 30

        # Add horizontal dimensions
        x_start = width
        segments = []
        # First edge
        segments.append(('edge', x_start-edge, x_start ))
        x_start -=edge

        # Last edge
        segments.append(('edge', 0, x_start))
```

```
        # Draw each segment
        for label, x1, x2 in segments:
            value = x2 - x1
            self.addHorizontalDimension(x1, -h_offset, x2, -
                h_offset, f"{value:.1f}", pen)
        # Add vertical dimensions
        self.addVerticalDimension(width + v_offset, 0, width +
            v_offset, end, str(end), pen)
        for i in range(self.rows - 1):
            self.addVerticalDimension(width + v_offset, end + i *
                pitch, width + v_offset, end + (i + 1) * pitch,
                str(pitch), pen)

        # Add bottom end distance dimension
        self.addVerticalDimension(width + v_offset, height, width
            + v_offset, height - end, str(end), pen)

        # Add left side dimension
        total_height = 2 * end + (self.rows - 1) * pitch
        self.addVerticalDimension(-v_offset, 0, -v_offset,
            total_height, str(total_height), pen)

    def addHorizontalDimension(self, x1, y1, x2, y2, text, pen):
        self.scene.addLine(x1, y1, x2, y2, pen)
        arrow_size = 5
        ext_length = 10
        self.scene.addLine(x1, y1 - ext_length/2, x1, y1 +
            ext_length/2, pen)
        self.scene.addLine(x2, y2 - ext_length/2, x2, y2 +
            ext_length/2, pen)

        points_left = [
            (x1, y1),
            (x1 + arrow_size, y1 - arrow_size/2),
```

```
            (x1 + arrow_size, y1 + arrow_size/2)
        ]
        polygon_left = self.scene.addPolygon(QPolygonF([QPointF(x
            , y) for x, y in points_left]), pen)
        polygon_left.setBrush(QBrush(Qt.black))

        points_right = [
            (x2, y2),
            (x2 - arrow_size, y2 - arrow_size/2),
            (x2 - arrow_size, y2 + arrow_size/2)
        ]
        polygon_right = self.scene.addPolygon(QPolygonF([QPointF(
            x, y) for x, y in points_right]), pen)
        polygon_right.setBrush(QBrush(Qt.black))

        text_item = self.scene.addText(text)
        font = QFont()
        font.setPointSize(5)
        text_item.setFont(font)

        if y1 < 0:
            text_item.setPos((x1 + x2) / 2 - text_item.
                boundingRect().width() / 2, y1 - 25)
        else:
            text_item.setPos((x1 + x2) / 2 - text_item.
                boundingRect().width() / 2, y1 + 5)

    def addVerticalDimension(self, x1, y1, x2, y2, text, pen):
        self.scene.addLine(x1, y1, x2, y2, pen)
        arrow_size = 5
        ext_length = 10
        self.scene.addLine(x1 - ext_length/2, y1, x1 + ext_length
            /2, y1, pen)
```

```
        self.scene.addLine(x2 - ext_length/2, y2, x2 + ext_length
            /2, y2, pen)

        if y2 > y1:
            points_top = [
                (x1, y1),
                (x1 - arrow_size/2, y1 + arrow_size),
                (x1 + arrow_size/2, y1 + arrow_size)
            ]
            polygon_top = self.scene.addPolygon(QPolygonF([
                QPointF(x, y) for x, y in points_top]), pen)
            polygon_top.setBrush(QBrush(Qt.black))

            points_bottom = [
                (x2, y2),
                (x2 - arrow_size/2, y2 - arrow_size),
                (x2 + arrow_size/2, y2 - arrow_size)
            ]
            polygon_bottom = self.scene.addPolygon(QPolygonF([
                QPointF(x, y) for x, y in points_bottom]), pen)
            polygon_bottom.setBrush(QBrush(Qt.black))
        else:
            points_top = [
                (x2, y2),
                (x2 - arrow_size/2, y2 + arrow_size),
                (x2 + arrow_size/2, y2 + arrow_size)
            ]
            polygon_top = self.scene.addPolygon(QPolygonF([
                QPointF(x, y) for x, y in points_top]), pen)
            polygon_top.setBrush(QBrush(Qt.black))

            points_bottom = [
                (x1, y1),
                (x1 - arrow_size/2, y1 - arrow_size),
```

```
                (x1 + arrow_size/2, y1 - arrow_size)
            ]
            polygon_bottom = self.scene.addPolygon(QPolygonF([
                QPointF(x, y) for x, y in points_bottom]), pen)
            polygon_bottom.setBrush(QBrush(Qt.black))

        text_item = self.scene.addText(text)
        font = QFont()
        font.setPointSize(5)
        text_item.setFont(font)

        if x1 < 0:
            text_item.setPos(x1 - 10 - text_item.boundingRect().
                width(), (y1 + y2) / 2 - text_item.boundingRect().
                height() / 2)
        else:
            text_item.setPos(x1 + 15, (y1 + y2) / 2 - text_item.
                boundingRect().height() / 2)
```

7.3.3 Spacing Window Code Explanation

Module Overview: `EndPlateDetailer`

The `EndPlateDetailer` class is a PyQt5-based GUI application that generates 2D engineering drawings of end plate connections used in structural steel design. It visualizes bolt arrangements, welds, and dimension annotations critical for construction documentation and validation.

Class: `EndPlateDetailer(QMainWindow)`

Constructor

```
def __init__(self, connection_obj, rows=3, cols=2, main=None)
```

Parameters:

- `connection_obj`: Object containing end plate design parameters.
- `rows`: Number of bolt rows (default: 3).
- `cols`: Number of bolt columns (default: 2).
- `main`: Main application reference containing design outputs.

Important Instance Variables

- `self.plate_height`, `self.plate_width`: Dimensions of the end plate.
- `self.hole_dia`: Bolt hole diameter.
- `self.weldsize`, `self.weldgap`: Weld properties.
- `self.scene`, `self.view`: Graphics objects for rendering.
- `self.connection`: Input design object.

Core Methods

`initUI()`

Initializes the layout, sets window properties, and prepares the QGraphicsScene and QGraphicsView. The layout consists of:

- **Left Panel:** Parameter display (`pitch`, `end`, `gauge`, `edge`).
- **Right Panel:** Drawing canvas.

`get_parameters()`

Maps output values to internal parameters:

```
KEY_OUT_PITCH     → 'pitch'
KEY_OUT_END_DIST  → 'end'
KEY_OUT_GAUGE1    → 'gauge1'
```

```
KEY_OUT_GAUGE2    → 'gauge2'
KEY_OUT_GAUGE     → 'gauge'
KEY_OUT_EDGE_DIST → 'edge'
```

Returns a dictionary containing bolt spacing and hole geometry.

createDrawing(params)

Draws the end plate, including:

- Plate outline as a black rectangle
- Bolt holes as blue circles placed using the following logic:

```
x_center = plate_width - edge
for i in range(col):
    x_center -= gauge1 if i % 2 == 0 else gauge2
y_center = end + row * pitch
```

- Weld fillets as red rectangles, placed at mid-height with spacing based on `weldgap`
- Dimension lines are added via `addDimensions()`

addDimensions(params, pen)

Adds annotated dimension lines for:

- Horizontal: Edge and gauge spacing
- Vertical: Pitch, end distances, and total height
- All lines have arrowheads and numeric text labels

addHorizontalDimension(...) & addVerticalDimension(...)

Draw dimension lines with:

- Main dimension line between two points
- Extension lines at each end

- Inward-pointing triangular arrows
- Text centered along the dimension line

Technical Details

Graphics Configuration

- **QGraphicsScene**: Drawing container
- **QGraphicsView**: Rendered view of the scene
- **QPen/QBrush**: For color, thickness, and fill

Coordinate System

- Origin at top-left (0,0)
- +X rightward, +Y downward
- Dimensions in millimeters

Color Convention

- **Black**: Plate outline and dimensions
- **Blue**: Bolt holes
- **Red**: Weld fillets

Usage Example

```
detailer = EndPlateDetailer(connection_obj, rows=3, cols=2, main=main_app)
detailer.show()
```

Dependencies

- **PyQt5**: GUI framework
- `sys`: System utilities
- `..Common`: Constant definitions (KEY_OUT_*)
- `.additionalfns`: Utility functions
- `..design_type.connection.end_plate_connection`

Error Handling

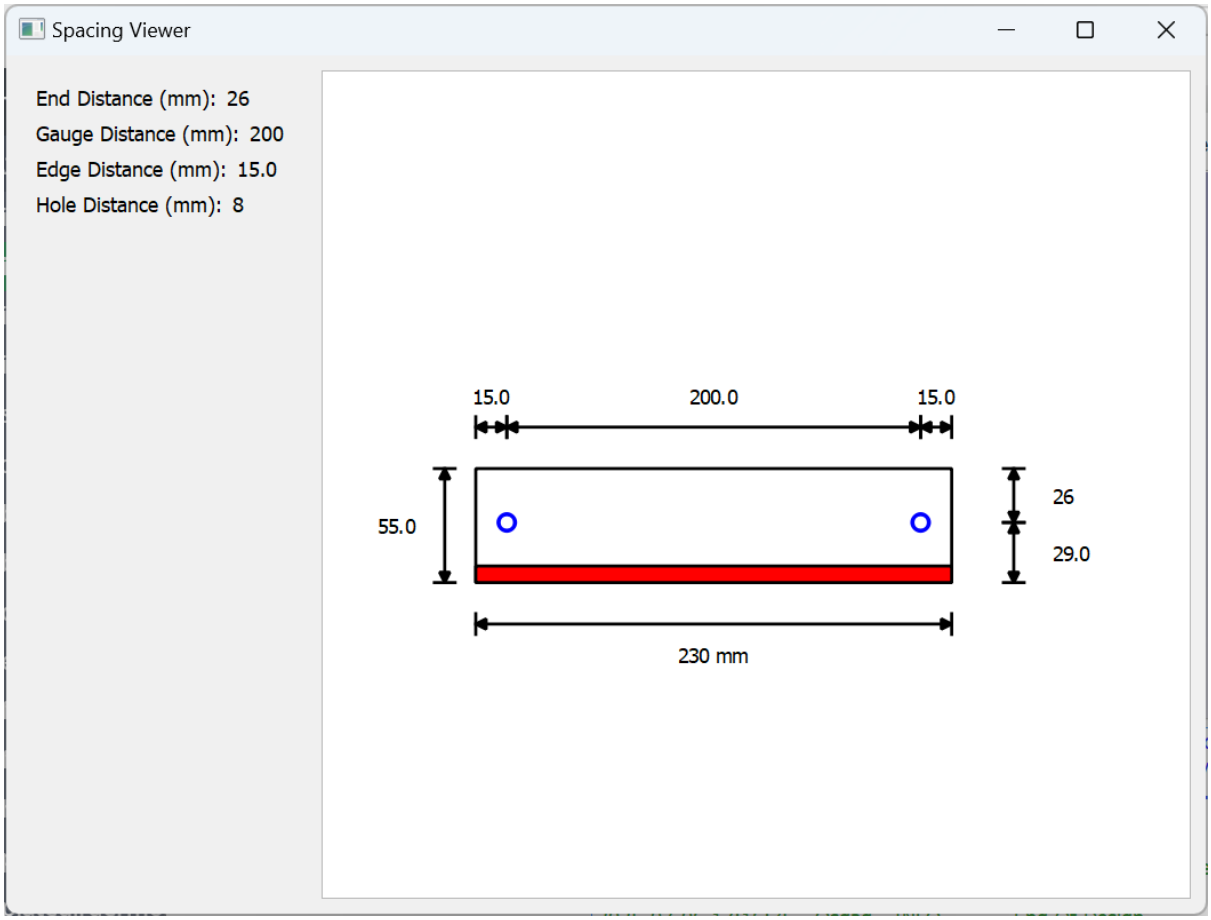
- Null checks for `main` input
- Validation for parameter values before rendering
- Scene initialization handled to avoid runtime graphics errors

7.4 Shear Connection - Seated Angle

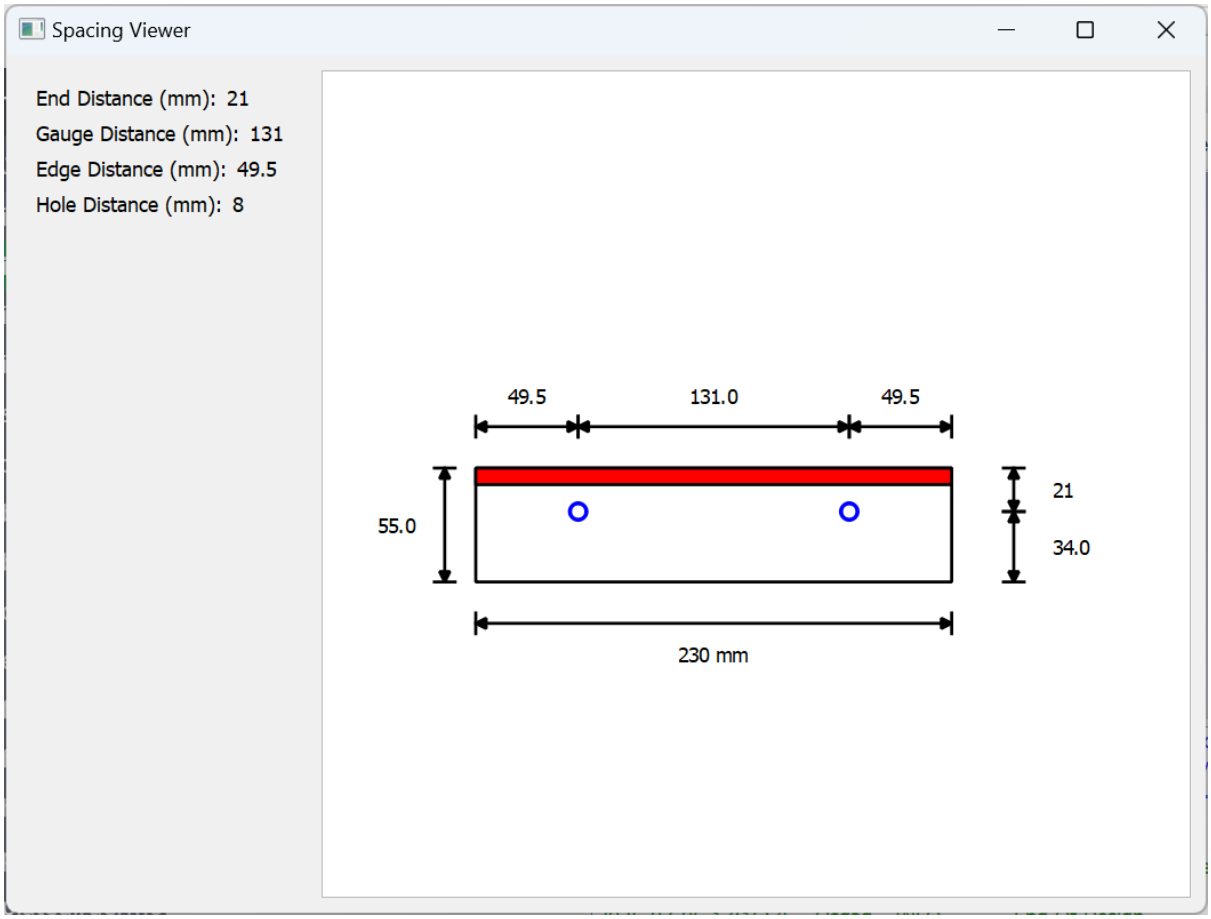
7.4.1 Seated Angle Connection

Window Image

On Column



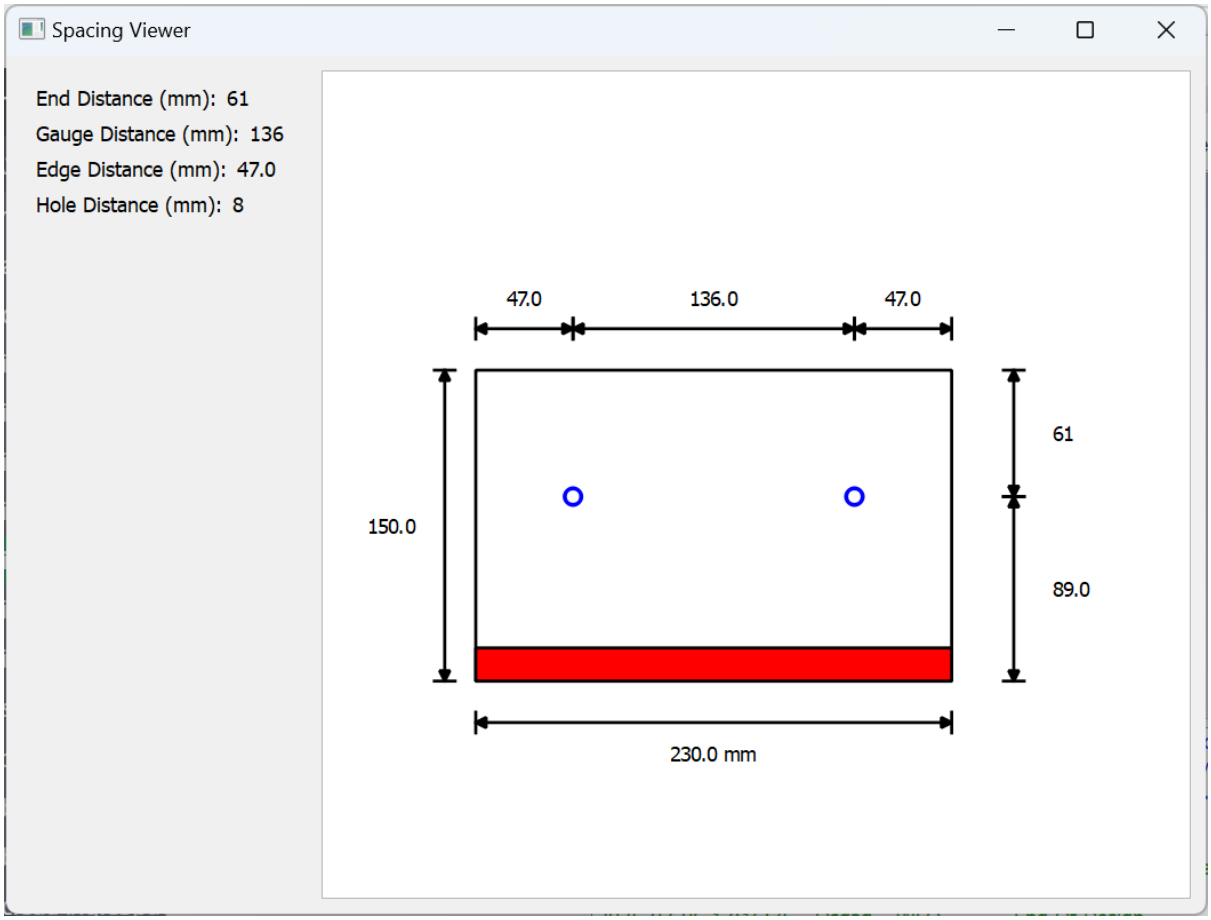
On Beam


Spacing Viewer
End Distance (mm): 21
Gauge Distance (mm): 131
Edge Distance (mm): 49.5
Hole Distance (mm): 8
49.5
131.0
49.5
55.0
21
34.0
230 mm

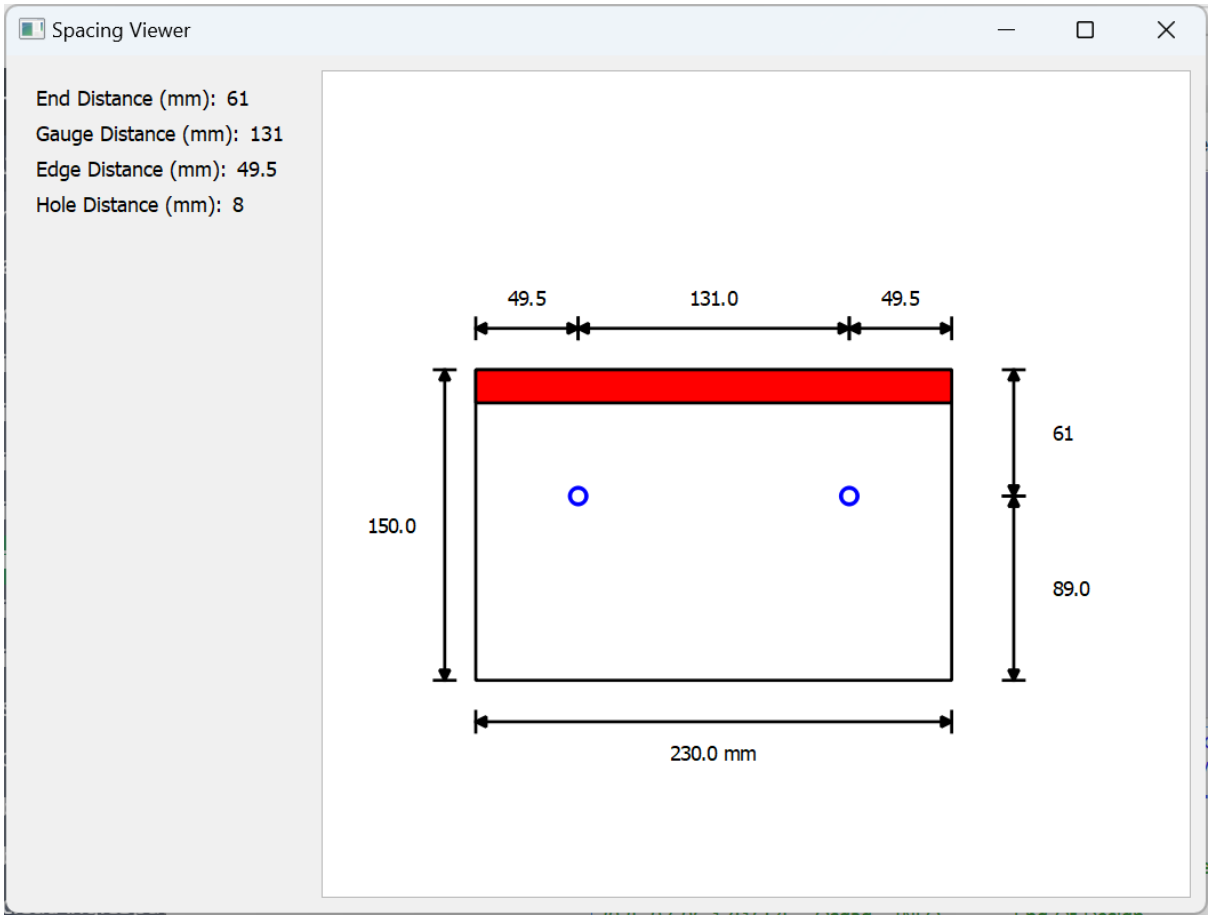
7.4.2 Top Angle

Window Image

On Column



On Beam



7.4.3 Code

```
import sys
from PyQt5.QtWidgets import (QApplication, QMainWindow, QWidget,
    QVBoxLayout,
                             QHBoxLayout, QLabel, QGraphicsView,
                             QGraphicsScene)
from PyQt5.QtGui import QPixmap
from PyQt5.QtCore import Qt, QRectF
from PyQt5.QtGui import QPainter, QPen, QFont
from PyQt5.QtGui import QPolygonF, QBrush
from PyQt5.QtCore import QPointF
from ..Common import *
from .additionalfns import calculate_total_width
```

```
class SeatedanglespacingOnCol(QMainWindow):
    def __init__(self, connection_obj, rows=3, cols=2 , main =
       None):
        super().__init__()

        self.connection = connection_obj
        self.val=rows

        if self.val==3 or self.val==4:
            self.plate_width=main.seated_angle.width
            self.plate_length=main.seated_angle.leg_a_length
            self.plate_thickness=float(main.seated_angle.
               designation.split(" x ")[-1])
        else:
            self.plate_width=main.top_angle.width
            self.plate_length=main.top_angle.leg_a_length
            self.plate_thickness=float(main.top_angle.designation
               .split(" x ")[-1])


        arr=[main.top_spacing_col(main,True),main.
           top_spacing_beam(main,True),main.seated_spacing_col(
           main,True),
             main.seated_spacing_beam(main,True)]
        val=self.val-1
        print(val)
        # print(arr[0],len(arr[0]))
        # print('\n\n')
        # for i in range(len(arr[0])):
        #     print(f"INDEX : {i}  : {arr[0][i]} , {arr[0][i
           ][3]}")
        for i in arr[2]:
            print(i)
            print('\n\n')
```

```
        data = {entry[0]: entry[3] for entry in arr[val] if entry
            [0]}
        print(data)
        self.rows  = data['Bolt.Rows']
        self.cols  = data['Bolt.Cols']
        self.End   = data['Bolt.EndDist']
        self.Gauge = data['Bolt.Gauge']
        if self.Gauge==0 and 'Bolt.GaugeCentral' in data:
            self.Gauge=data['Bolt.GaugeCentral']
        self.Edge  = data['Bolt.EdgeDist']
        # return
        print(f"""
        Plate Dimensions
        ----------------
        Plate Width  : {self.plate_width} mm
        Plate Length : {self.plate_length} mm

        Bolt Layout
        -----------
        Rows         : {self.rows}
        Columns      : {self.cols}
        End Distance : {self.End} mm
        Gauge        : {self.Gauge} mm
        Edge Distance: {self.Edge} mm
        """)
        # self.initUI()
        self.param_map = {
    'end': self.End,
    'gauge': self.Gauge,
    'edge': self.Edge,
     'hole': main.bolt.bolt_diameter_provided
}
```

```
        print(self.param_map)
        self.initUI()
    def initUI(self):
        self.setWindowTitle('Bolt Pattern Generator')
        self.setGeometry(100, 100, 1050, 750)

        # Main layout
        main_layout = QHBoxLayout()

        # Left panel for parameter display
        left_panel = QWidget()
        left_layout = QVBoxLayout()
        params=self.param_map
        # Parameter display labels
        # Display the parameter values
        for key, value in params.items():
            param_layout = QHBoxLayout()
            param_label = QLabel(f'{key.title()} Distance (mm):')
            value_label = QLabel(f'{value}')
            param_layout.addWidget(param_label)
            param_layout.addWidget(value_label)
            left_layout.addLayout(param_layout)

        left_layout.addStretch()
        left_panel.setLayout(left_layout)

        # Right panel for the drawing using QGraphicsView
        self.scene = QGraphicsScene()
        self.view = QGraphicsView(self.scene)
        self.view.setRenderHint(QPainter.Antialiasing)

        # Create and add the drawing to the scene
        self.createDrawing(params)
```

```
        # Add panels to main layout
        main_layout.addWidget(left_panel, 1)
        main_layout.addWidget(self.view, 3)

        # Set main widget
        main_widget = QWidget()
        main_widget.setLayout(main_layout)
        self.setCentralWidget(main_widget)

        # Ensure the view shows all content
        self.view.fitInView(self.scene.sceneRect(), Qt.
            KeepAspectRatio)



    def createDrawing(self, params):

        # Extract parameters

        end = params['end']
        if 'gauge' in params:
            gauge = params['gauge']
        edge = params['edge']
        hole_diameter = params['hole']
        print(f"rows: {self.rows}, cols: {self.cols}")
        # Calculate dimensions

        width = self.plate_width

        height = self.plate_length
        # Set up pens
        outline_pen = QPen(Qt.blue, 2)
        dimension_pen = QPen(Qt.black, 1.5)
        weld_fill = QBrush(Qt.red)
```

```
        # Dimension offsets
        h_offset = 40
        v_offset = 60

        # Create scene rectangle with extra space for dimensions
        self.scene.setSceneRect(-h_offset, -v_offset,
                                width + 2*v_offset, height + 2*
                                   h_offset)

        # Draw rectangle
        self.scene.addRect(0, 0, width, height, dimension_pen)

        if self.val==3 or self.val==1:
            self.scene.addRect(0, height-self.plate_thickness,
               width, self.plate_thickness, dimension_pen,
               weld_fill)
        elif self.val==4 or self.val==2:
            self.scene.addRect(0, 0, width, self.plate_thickness,
                dimension_pen, weld_fill)


        # Draw holes
        for row in range(self.rows):
            for col in range(self.cols):
                # Start from edge distance (center of first hole)
                x_center = edge
                for i in range(col):
                    x_center += gauge

                # Center of hole is at (x_center, y_center)
                # Subtract hole_diameter/2 to draw ellipse
                   properly from top-left
                x = x_center - hole_diameter / 2
```

```
                y_center = end
                y = y_center - hole_diameter / 2

                print(f"row: {row}, col: {col}, x: {x}, y: {y}")
                self.scene.addEllipse(x, y, hole_diameter,
                   hole_diameter, outline_pen)
        print(params,dimension_pen)
        # Add dimensions
        self.addDimensions(params, dimension_pen)

    def addDimensions(self, params, pen):
        # Extract parameters
        end = params['end']
        if 'gauge' in params:
            gauge = params['gauge']

        edge = params['edge']



        width=self.plate_width
        height=self.plate_length

        # Offsets for dimension lines
        h_offset = 20
        v_offset = 30

        # Add horizontal dimensions
        x_start = 0
        segments = []
        # First edge
        segments.append(('edge', x_start, x_start + edge))
        x_start += edge
        segments.append(('edge' ,x_start,x_start+gauge ))
```

```
        x_start+=gauge
        # Last edge
        segments.append(('edge', x_start, x_start + edge))

        # Draw each segment
        for label, x1, x2 in segments:
            value = x2 - x1
            self.addHorizontalDimension(x1, -h_offset, x2, -
               h_offset, f"{value:.1f}", pen)
        self.addHorizontalDimension(0,height+h_offset,width,
           height+h_offset,f"{width} mm" , pen)
        # Add vertical dimensions
        # Add top end distance dimension (from 0 to end)
        self.addVerticalDimension(width + v_offset, 0, width +
           v_offset, end, str(end), pen)

        # Add remaining distance from end to height
        self.addVerticalDimension(width + v_offset , end, width +
            v_offset , height, str(height - end), pen)

        # Add left side dimension
        total_height = 2 * end + (self.rows - 1)
        self.addVerticalDimension(-v_offset/2, 0, -v_offset/2,
           height, str(height), pen)

    def addHorizontalDimension(self, x1, y1, x2, y2, text, pen):
        self.scene.addLine(x1, y1, x2, y2, pen)
        arrow_size = 5
        ext_length = 10
        self.scene.addLine(x1, y1 - ext_length/2, x1, y1 +
           ext_length/2, pen)
        self.scene.addLine(x2, y2 - ext_length/2, x2, y2 +
           ext_length/2, pen)
```

```
        points_left = [
            (x1, y1),
            (x1 + arrow_size, y1 - arrow_size/2),
            (x1 + arrow_size, y1 + arrow_size/2)
        ]
        polygon_left = self.scene.addPolygon(QPolygonF([QPointF(x
            , y) for x, y in points_left]), pen)
        polygon_left.setBrush(QBrush(Qt.black))

        points_right = [
            (x2, y2),
            (x2 - arrow_size, y2 - arrow_size/2),
            (x2 - arrow_size, y2 + arrow_size/2)
        ]
        polygon_right = self.scene.addPolygon(QPolygonF([QPointF(
            x, y) for x, y in points_right]), pen)
        polygon_right.setBrush(QBrush(Qt.black))

        text_item = self.scene.addText(text)
        font = QFont()
        font.setPointSize(5)
        text_item.setFont(font)

        if y1 < 0:
            text_item.setPos((x1 + x2) / 2 - text_item.
                boundingRect().width() / 2, y1 - 25)
        else:
            text_item.setPos((x1 + x2) / 2 - text_item.
                boundingRect().width() / 2, y1 + 5)

    def addVerticalDimension(self, x1, y1, x2, y2, text, pen):
        self.scene.addLine(x1, y1, x2, y2, pen)
        arrow_size = 5
        ext_length = 10
```

```
        self.scene.addLine(x1 - ext_length/2, y1, x1 + ext_length
            /2, y1, pen)
        self.scene.addLine(x2 - ext_length/2, y2, x2 + ext_length
            /2, y2, pen)

        if y2 > y1:
            points_top = [
                (x1, y1),
                (x1 - arrow_size/2, y1 + arrow_size),
                (x1 + arrow_size/2, y1 + arrow_size)
            ]
            polygon_top = self.scene.addPolygon(QPolygonF([
                QPointF(x, y) for x, y in points_top]), pen)
            polygon_top.setBrush(QBrush(Qt.black))

            points_bottom = [
                (x2, y2),
                (x2 - arrow_size/2, y2 - arrow_size),
                (x2 + arrow_size/2, y2 - arrow_size)
            ]
            polygon_bottom = self.scene.addPolygon(QPolygonF([
                QPointF(x, y) for x, y in points_bottom]), pen)
            polygon_bottom.setBrush(QBrush(Qt.black))
        else:
            points_top = [
                (x2, y2),
                (x2 - arrow_size/2, y2 + arrow_size),
                (x2 + arrow_size/2, y2 + arrow_size)
            ]
            polygon_top = self.scene.addPolygon(QPolygonF([
                QPointF(x, y) for x, y in points_top]), pen)
            polygon_top.setBrush(QBrush(Qt.black))

            points_bottom = [
```

```
                (x1, y1),
                (x1 - arrow_size/2, y1 - arrow_size),
                (x1 + arrow_size/2, y1 - arrow_size)
            ]
            polygon_bottom = self.scene.addPolygon(QPolygonF([
                QPointF(x, y) for x, y in points_bottom]), pen)
            polygon_bottom.setBrush(QBrush(Qt.black))

        text_item = self.scene.addText(text)
        font = QFont()
        font.setPointSize(5)
        text_item.setFont(font)

        if x1 < 0:
            text_item.setPos(x1 - 10 - text_item.boundingRect().
                width(), (y1 + y2) / 2 - text_item.boundingRect().
                height() / 2)
        else:
            text_item.setPos(x1 + 15, (y1 + y2) / 2 - text_item.
                boundingRect().height() / 2)
```

7.4.4 Code Explanation

Module Overview: `SeatedanglespacingOnCol`

The `Seatedanglespacing.py` module provides a PyQt5-based GUI component for visualizing bolt patterns on seated angle connections used in structural steel design. It creates an interactive window that displays a 2D technical drawing with full dimensioning.

Class: `SeatedanglespacingOnCol(QMainWindow)`

Constructor

```
def __init__(self, connection_obj, rows=3, cols=2, main=None)
```

Parameters:

- `connection_obj`: Design object containing connection parameters
- `rows`: Number of bolt rows (default: 3)
- `cols`: Number of bolt columns (default: 2)
- `main`: Reference to main application with design values

Instance Variables

- `plate_width`, `plate_length`, `plate_thickness`: Geometry of the angle plate
- `rows`, `cols`: Bolt grid layout configuration
- `End`, `Gauge`, `Edge`: Bolt spacing dimensions
- `connection`: Reference to the design object
- `val`: Configuration identifier for row type
- `param_map`: Dictionary of all spacing parameters

Core Methods

`initUI()`

Initializes the graphical interface with a horizontal layout:

- Left panel (1/4): Displays design parameters
- Right panel (3/4): Contains the drawing canvas (QGraphicsView)

Window Properties

- Title: `"Bolt Pattern Generator"`
- Size: 1050x750 pixels
- Anti-aliasing: Enabled for smoother drawings

`createDrawing(params)`

Generates the primary technical drawing with:

- **Plate Outline:** Drawn as a rectangle
- **Weld Zones:** Red-filled regions indicating weld areas
- **Bolt Holes:** Blue circular features positioned via:

```
x_center = edge + (col * gauge)
y_center = end
x = x_center - hole_diameter / 2
y = y_center - hole_diameter / 2
```

- **Dimension Lines:** Added via `addDimensions()`

`addDimensions(params, pen)`

Adds comprehensive dimensioning annotations for:

- **Horizontal:**
  - Left edge to first bolt column
  - Inter-column gauge distances
  - Right edge to last bolt column
  - Total plate width
- **Vertical:**
  - Top edge to bolt row
  - Remaining distance to bottom
  - Total plate height

`addHorizontalDimension(x1, y1, x2, y2, text, pen)`

Draws a horizontal dimension line with:

- Extension lines at both ends (10mm length)
- Triangular arrowheads (5mm base)
- Centered text label above/below the line

Arrow Example:

```
points_left = [
    (x1, y1),
    (x1 + arrow_size, y1 - arrow_size/2),
    (x1 + arrow_size, y1 + arrow_size/2)
]
```

`addVerticalDimension(x1, y1, x2, y2, text, pen)`

Draws vertical dimension lines with:

- Direction-aware arrows
- Left/right-aligned text
- Horizontal extension lines at endpoints

Technical Specifications

Coordinate System

- Origin: Top-left corner (0,0)
- +X: Rightward, +Y: Downward
- Units: Millimeters

Drawing Configuration

- **Arrow Size:** 5mm
- **Extension Lines:** 10mm
- **Font Size:** 5pt
- **Pen Widths:** 2px for outlines, 1.5px for dimensions

Offset Settings

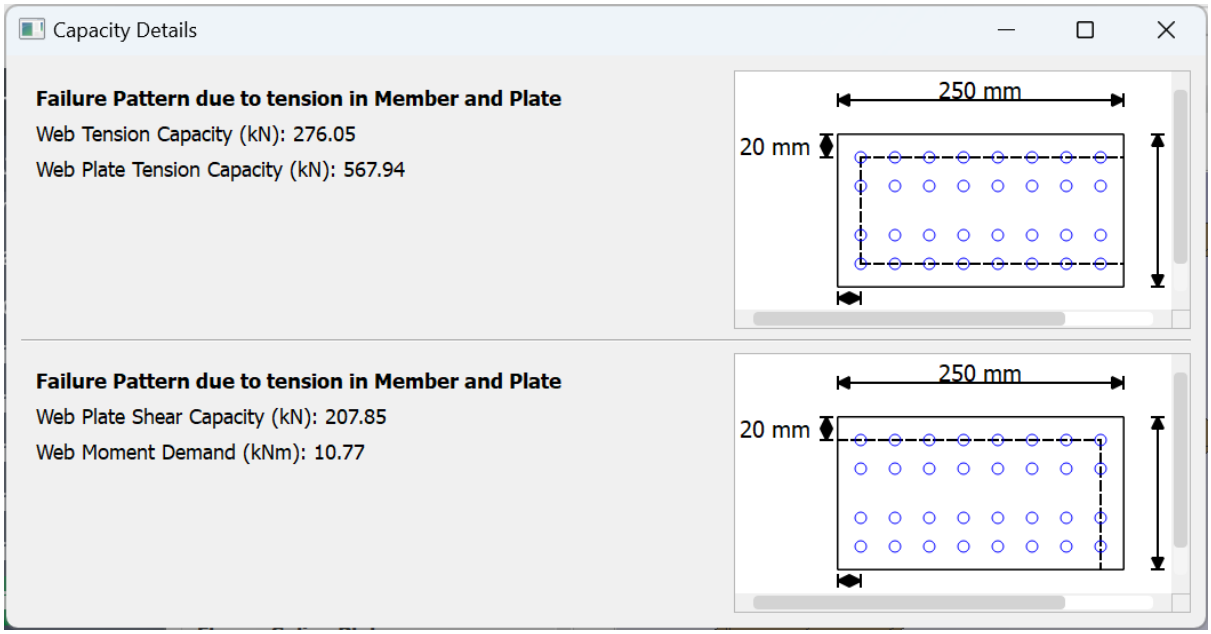
- Horizontal Offset: 40mm
- Vertical Offset: 60mm
- Dimension Line Offsets: 20mm (horizontal), 30mm (vertical)

Chapter 8

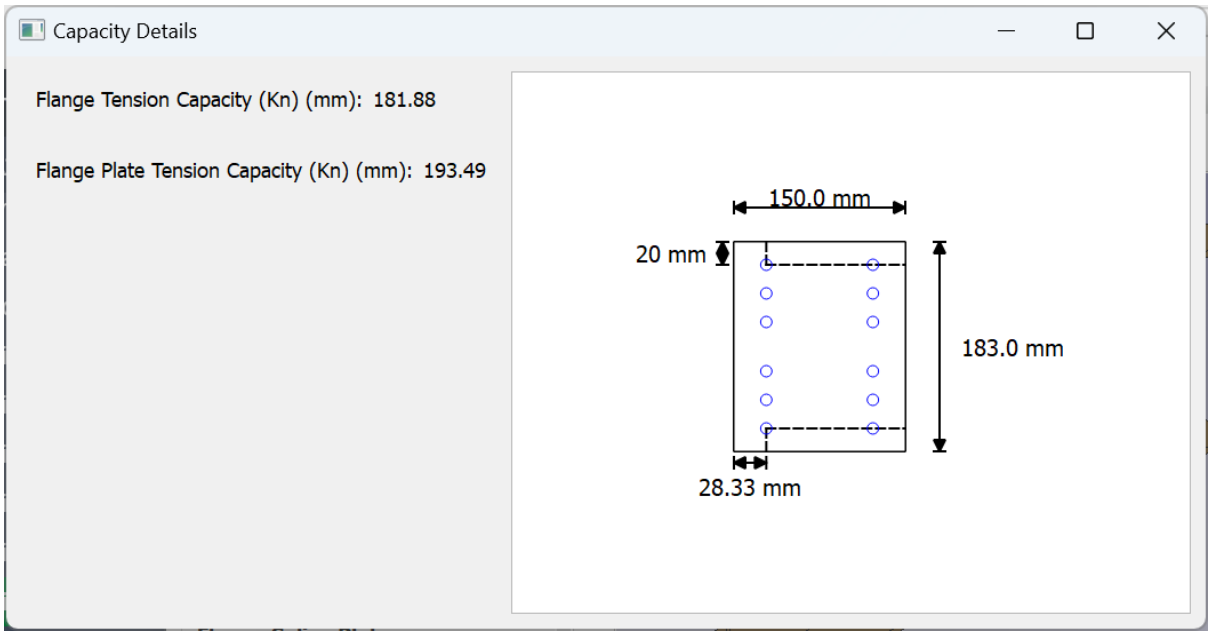
Capacity Details and Windows

8.1 Beam to Beam Splice - Cover Plate Bolted

Web Capacity



Flange Capacity



8.1.1 Code

```
import sys
from PyQt5.QtWidgets import (QApplication, QMainWindow, QWidget,
   QVBoxLayout,
                             QHBoxLayout, QLabel, QGraphicsView,
                             QGraphicsScene,QGraphicsRectItem,
                                QFrame)
from PyQt5.QtGui import QPixmap
from PyQt5.QtCore import Qt, QRectF
from PyQt5.QtGui import QPainter, QPen, QFont, QColor
from PyQt5.QtGui import QPolygonF, QBrush
from PyQt5.QtCore import QPointF
from ..Common import *
from .additionalfns import calculate_total_width


try:
    pen_style_dash = Qt.PenStyle.DashLine
except AttributeError:
```

```
    raise RuntimeError("Your PyQt5 version does not support
       dashed lines via Qt.PenStyle.DashLine. Please update PyQt5
       .")

class B2Bcoverplate_capacity_details(QMainWindow):
    def __init__(self, connection_obj, rows=3, cols=2 , main =
       None):
        print(main)

        if main:
            self.drawing_type=main[2]
            self.web=main[1]
            web=main[1]
            main=main[0]
        super().__init__()
        self.connection = connection_obj
        # return
        data=main.output_values(main,True)
        print(type(main))
        dict1={i[0] : i[3] for i in data}



        for i in dict1:
            print(f'{i} : {dict1[i]}')


        if web==True:
            self.plate_length=dict1['Web_Plate.Height (mm)']
            self.plate_width=dict1['Web_Plate.Width']
            self.bolt_diameter=dict1['Bolt.Diameter']
            web_capcity=dict1['Web_plate.spacing'][1]
            print(web_capcity(main,True))
            data2=web_capcity(main,True)
```

```
            for i in range(len(data2)):
                print(f"{i} : {data2[i]}")
            self.pitch=data2[2][3]
            self.End=data2[3][3]
            self.Gauge=data2[4][3]
            self.Edge=data2[5][3]
            bolt_cap=dict1['Web Bolt.Capacities'][1]
            print(bolt_cap(main,True))
            bolt_cap=bolt_cap(main,True)
        elif web==False:
            self.plate_length=dict1['Flange_Plate.Width (mm)']
            self.plate_width=dict1['flange_plate.Length']
            self.bolt_diameter=dict1['Bolt.Diameter']
            flange_capcity=dict1['Flange_plate.spacing'][1]
            data2=flange_capcity(main,True)
            self.pitch=data2[2][3]
            self.End=data2[3][3]
            self.Gauge=data2[4][3]
            self.Edge=data2[5][3]
            bolt_cap=dict1['Bolt.Capacities'][1]
            print(bolt_cap(main,True))
            bolt_cap=bolt_cap(main,True)



        #capacity
        if web==True and self.drawing_type=="capacity":
            #web capacity details
            web_capacity_fnc=dict1['section.web_capacities'][1]
            web_capacity_val=web_capacity_fnc(main,True)
            self.web_capacity_details = {item[1]: float(item[3])
               for item in web_capacity_val if item[2] == '
               TextBox'}
```

```
        #capacity
        elif web==False and self.drawing_type=="capacity":
            #flange capacity details
            flange_capacity_fnc=dict1['section.flange_capacity'
               ][1]
            flange_capacity_val=flange_capacity_fnc(main,True)
            self.flange_capacity_details={item[1]: float(item[3])
                for item in flange_capacity_val if item[2] == '
               TextBox'}




        self.cols=bolt_cap[1][3]
        self.rows=bolt_cap[2][3]/self.cols
        self.initUI()


    def initUI(self):
        self.setWindowTitle('Bolt Pattern Generator')
        self.setGeometry(100, 100, 1050, 500)

        # Print summary (optional debug/log info)
        print(f"""
        ----------------------------------------
            Plate & Bolt Configuration Summary
        ----------------------------------------
        Plate Length            : {self.plate_length} mm
        Plate Width             : {self.plate_width} mm
        Bolt Diameter           : {self.bolt_diameter} mm

        Bolt Spacing Details:
        ---------------------
        Pitch Distance          : {self.pitch} mm
```

```
        End Distance            : {self.End} mm
        Gauge Distance          : {self.Gauge} mm
        Edge Distance           : {self.Edge} mm

        Bolt Arrangement:
        ----------------
        Number of Columns       : {self.cols}
        Number of Rows          : {self.rows}
        """)

        # Main layout
        main_layout = QHBoxLayout()

        # Left panel for parameter display
        left_panel = QWidget()
        left_layout = QVBoxLayout()

        # Get parameter dictionary
        params = self.get_parameters((self.web,self.drawing_type)
           )
        count=0
        for key, value in params.items():
            if self.web==False and self.drawing_type=="capacity":
                param_layout = QHBoxLayout()
                space_label = QLabel('  ')
                param_label = QLabel(f'{key.title()} (mm):')
                value_label = QLabel(f'{value}')
                param_layout.addWidget(param_label)
                param_layout.addWidget(value_label)
                left_layout.addLayout(param_layout)
                # Add a blank label for vertical spacing
                left_layout.addWidget(QLabel(''))
            elif self.web==True and self.drawing_type=="capacity"
               :
```

```
                param_layout = QHBoxLayout()
                space_label = QLabel('  ')
                param_label = QLabel(f'{key.title()} (mm):')
                value_label = QLabel(f'{value}')
                param_layout.addWidget(param_label)
                param_layout.addWidget(value_label)
                left_layout.addLayout(param_layout)
                # Add a blank label for vertical spacing
                left_layout.addWidget(QLabel(''))

                count+=1
            else:
                param_layout = QHBoxLayout()
                param_label = QLabel(f'{key.title()} (mm):')
                value_label = QLabel(f'{value}')
                param_layout.addWidget(param_label)
                param_layout.addWidget(value_label)
                left_layout.addLayout(param_layout)

        left_layout.addStretch()
        left_panel.setLayout(left_layout)

        # Determine font and arrow size based on plate size
        self.fontsize = 10
        self.arrowsize = 10
        if self.plate_length > 1200 or self.plate_width > 1200:
            self.fontsize = 12
            self.arrowsize = 12
        elif self.plate_length > 600 or self.plate_width > 600:
            self.fontsize = 7.5
            self.arrowsize = 7.5

        if self.web == True and self.drawing_type == "capacity":
```

```
        # Two drawings, each with its own parameter set,
            separated by a horizontal line
        self.scene1 = QGraphicsScene()
        self.view1 = QGraphicsView(self.scene1)
        self.view1.setRenderHint(QPainter.Antialiasing)

        self.scene2 = QGraphicsScene()
        self.view2 = QGraphicsView(self.scene2)
        self.view2.setRenderHint(QPainter.Antialiasing)

        self.createDrawing((self.web, self.drawing_type),
            self.scene1)
        self.createDrawing((self.web, self.drawing_type),
            self.scene2)

        if self.plate_length > 1200 or self.plate_width >
            1200:
             self.view1.resetTransform()
             self.view1.scale(0.35, 0.35)
             self.view2.resetTransform()
             self.view2.scale(0.35, 0.35)
        elif self.plate_length > 600 or self.plate_width >
            600:
             self.view1.resetTransform()
             self.view1.scale(0.5, 0.5)
             self.view2.resetTransform()
             self.view2.scale(0.5, 0.5)

        # Split parameters into two groups (first two, next
            two)
        params = list(self.get_parameters((self.web, self.
            drawing_type)).items())
        params1 = params[:2]
        params2 = params[2:]
```

```
        # Section 1: first two parameters and first drawing
        section1_layout = QHBoxLayout()
        section1_text_widget = QWidget()
        section1_text_layout = QVBoxLayout(
           section1_text_widget)
        param_label = QLabel('Failure Pattern due to tension
           in Member and Plate')
        param_label.setFont(QFont('Arial',12,QFont.Bold))
        section1_text_layout.addWidget(param_label)
        for key, value in params1:
            param_label = QLabel(f'{key}: {value}')
            section1_text_layout.addWidget(param_label)
        section1_text_layout.addStretch()
        section1_text_widget.setMinimumWidth(180)
        section1_layout.addWidget(section1_text_widget, 1)
        section1_layout.addWidget(self.view1, 2)

        # Section 2: next two parameters and second drawing
        section2_layout = QHBoxLayout()
        section2_text_widget = QWidget()
        section2_text_layout = QVBoxLayout(
           section2_text_widget)
        param_label = QLabel('Failure Pattern due to tension
           in Member and Plate')
        param_label.setFont(QFont('Arial',12,QFont.Bold))
        section2_text_layout.addWidget(param_label)
        for key, value in params2:
            param_label = QLabel(f'{key}: {value}')
            section2_text_layout.addWidget(param_label)
        section2_text_layout.addStretch()
        section2_text_widget.setMinimumWidth(180)
        section2_layout.addWidget(section2_text_widget, 1)
        section2_layout.addWidget(self.view2, 2)
```

```
            # Horizontal line between sections
            line = QFrame()
            line.setFrameShape(QFrame.HLine)
            line.setFrameShadow(QFrame.Sunken)

            # Main vertical layout
            main_vlayout = QVBoxLayout()
            main_vlayout.addLayout(section1_layout)
            main_vlayout.addWidget(line)
            main_vlayout.addLayout(section2_layout)

            self.view1.setMaximumWidth(400)
            self.view2.setMaximumWidth(400)

            main_widget = QWidget()
            main_widget.setLayout(main_vlayout)
            self.setCentralWidget(main_widget)
        else:
            # Only one drawing (original layout)
            left_panel = QWidget()
            left_layout = QVBoxLayout()
            params = self.get_parameters((self.web, self.
               drawing_type))
            for key, value in params.items():
                if self.web==False and self.drawing_type=="
                   capacity":
                    param_layout = QHBoxLayout()
                    space_label = QLabel('  ')
                    param_label = QLabel(f'{key.title()} (mm):')
                    value_label = QLabel(f'{value}')
                    param_layout.addWidget(param_label)
                    param_layout.addWidget(value_label)
                    left_layout.addLayout(param_layout)
```

```
                left_layout.addWidget(QLabel(''))
            else:
                param_layout = QHBoxLayout()
                param_label = QLabel(f'{key.title()} (mm):')
                value_label = QLabel(f'{value}')
                param_layout.addWidget(param_label)
                param_layout.addWidget(value_label)
                left_layout.addLayout(param_layout)
        left_layout.addStretch()
        left_panel.setLayout(left_layout)

        self.scene = QGraphicsScene()
        self.view = QGraphicsView(self.scene)
        self.view.setRenderHint(QPainter.Antialiasing)
        self.createDrawing((self.web, self.drawing_type),
           self.scene)

        if self.plate_length > 1200 or self.plate_width >
           1200:
            self.view.resetTransform()
            self.view.scale(0.35, 0.35)
        elif self.plate_length > 600 or self.plate_width >
           600:
            self.view.resetTransform()
            self.view.scale(0.5, 0.5)

        main_layout = QHBoxLayout()
        main_layout.addWidget(left_panel, 1)
        main_layout.addWidget(self.view, 3)
        main_widget = QWidget()
        main_widget.setLayout(main_layout)
        self.setCentralWidget(main_widget)

    # Automatically adjust view to fit scene
```

```
    def get_parameters(self,type_):
        if (type_[0]==True and type_[1]=="spacing") or (type_
           [0]==False and type_[1]=="spacing") :
            return {
            'Plate Length': self.plate_length,
            'Plate Width': self.plate_width,
            'Bolt Diameter': self.bolt_diameter,
            'Pitch Distance': self.pitch,
            'End Distance': self.End,
            'Gauge Distance': self.Gauge,
            'Edge Distance': self.Edge,
            'Number of Columns': self.cols,
            'Number of Rows': self.rows
            }
        elif type_[0]==True and type_[1]=="capacity":
            return self.web_capacity_details
        elif type_[0]==False and type_[1]=="capacity":
            return self.flange_capacity_details

    def createDrawing(self, type_, scene):



        try:
            plate_length = float(self.plate_length)
            plate_width = float(self.plate_width)
        except (TypeError, ValueError):
            print("Invalid plate dimensions")
            return
        rect = QRectF(0, 0, plate_length, plate_width)
        # Create a rectangle item
        rect_item = QGraphicsRectItem(rect)

        # Set pen and brush (black border, transparent fill)
```

```
        pen = QPen(QColor('black'))
        pen.setWidth(2)
        rect_item.setPen(pen)
        rect_item.setBrush(QBrush())  # Default is NoBrush

        # Add rectangle to the scene
        scene.addItem(rect_item)



        # Extract parameters
        outline_pen = QPen(QColor('black'))
        outline_pen.setWidth(1)

        # === Draw Base Plate Rectangle ===
        rect_item = QGraphicsRectItem(QRectF(0, 0, plate_length,
           plate_width))
        rect_item.setPen(outline_pen)
        rect_item.setBrush(QBrush(QColor('white')))
        scene.addItem(rect_item)
        dashed_pen = QPen(QColor('black'))
        dashed_pen.setStyle(pen_style_dash)
        dashed_pen.setWidth(2)
        if type_[0]==False and type_[1]=="capacity":

            #top drawing
            scene.addLine(self.Edge, 0, self.Edge, self.End,
               dashed_pen)

            scene.addLine(self.Edge, self.End, plate_length, self
               .End, dashed_pen)



            #bottom drawing
```

```
            scene.addLine(self.Edge, plate_width, self.Edge,
                plate_width-self.End, dashed_pen)

            scene.addLine(self.Edge, plate_width-self.End,
                plate_length, plate_width-self.End, dashed_pen)

        elif type_[0]==True and type_[1]=="capacity" and scene==
            self.scene1:

            scene.addLine(self.End, self.Edge, self.End,
                plate_width-self.End, dashed_pen)

            scene.addLine(self.End, self.Edge, plate_length, self
                .Edge, dashed_pen)

            scene.addLine(self.End, plate_width-self.Edge,
                plate_length, plate_width-self.Edge, dashed_pen)


        elif type_[0]==True and type_[1]=="capacity" and scene==
            self.scene2:

            scene.addLine(0, self.Edge, plate_length-self.End,
                self.Edge, dashed_pen)

            scene.addLine(plate_length-self.End, self.Edge,
                plate_length-self.End, plate_width, dashed_pen)


        # === Center of the base plate ===
        center_x = plate_length / 2
        center_y = plate_width / 2
        self.addHorizontalDimension(
            0, -30,  # x1 at left edge, y above plate
```

```
            self.plate_length, -30,  # x2 at right edge, same y
            f"{self.plate_length} mm", pen, scene
        )

        # Vertical dimension for plate width (to the left of the
           plate)
        self.addVerticalDimension(
            self.plate_length+30, 0,  # x left of plate, y1 at
               top
            self.plate_length+30, self.plate_width,  # x2 same,
               y2 at bottom
            f"{self.plate_width} mm", pen, scene
        )

        rows=int(self.rows)
        cols=int(self.cols)
        pitch=self.pitch
        gauge=self.Gauge
        end=self.End
        edge=self.Edge
        hole_dia = self.bolt_diameter
        radius = hole_dia / 2
        y_center = end  # Y position is fixed for top row
        # Center row if rows is odd
        outline_pen = QPen(QColor('blue'))
        outline_pen.setWidth(1)
        if rows % 2 != 0:
            y_center = self.plate_width / 2
            for i in range(cols // 2):
                x_center = edge + i * gauge
                scene.addEllipse(
                    x_center - radius,
                    y_center - radius,
                    hole_dia,
```

```
                hole_dia,
                outline_pen,
            )

        for i in range(cols // 2):
            x_center = self.plate_length - edge - i * gauge
            scene.addEllipse(
                x_center - radius,
                y_center - radius,
                hole_dia,
                hole_dia,
                outline_pen,
            )

        # Center bolt if cols is also odd
        if cols % 2 != 0:
            x_center = self.plate_length / 2
            scene.addEllipse(
                x_center - radius,
                y_center - radius,
                hole_dia,
                hole_dia,
                outline_pen,
            )

    # Center column if cols is odd (and rows is even)
    if cols % 2 != 0 and rows % 2 == 0:
        for j in range(rows // 2):
            y_center_top = end + j * pitch
            y_center_bottom = self.plate_width - end - j *
               pitch

            x_center = self.plate_length / 2
            scene.addEllipse(
```

```
                x_center - radius,
                y_center_top - radius,
                hole_dia,
                hole_dia,
                outline_pen,
            )
            scene.addEllipse(
                x_center - radius,
                y_center_bottom - radius,
                hole_dia,
                hole_dia,
                outline_pen,
            )

        # Draw left half bolts
        for row in range(int(rows)):
            if row < rows // 2:
                y_center = end + row * pitch
            else:
                row_from_bottom = row - rows // 2
                y_center = self.plate_width - end -
                   row_from_bottom * pitch

            # Left half bolts
            for i in range(cols // 2):
                x_center = edge + i * gauge
                scene.addEllipse(
                    x_center - radius,
                    y_center - radius,
                    hole_dia,
                    hole_dia,
                    outline_pen,
                )
```

```
            # Right half bolts
            for i in range(cols // 2):
                x_center = self.plate_length - edge - i * gauge
                scene.addEllipse(
                    x_center - radius,
                    y_center - radius,
                    hole_dia,
                    hole_dia,
                    outline_pen,
                )
        self.addHorizontalDimension(
            0, self.plate_width+10,  # x1 at left edge, y above
               plate
            self.Edge, self.plate_width+10,  # x2 at right edge,
               same y
            f"{self.Edge} mm", pen, scene
        )
        self.addVerticalDimension(
            -10, 0,  # x left of plate, y1 at top
            -10, self.End,  # x2 same, y2 at bottom
            f"{self.End} mm", pen, scene
        )
        # self.addHorizontalDimension(
        #     self.Edge-hole_dia/2, 10,
        #     self.Edge+hole_dia/2,10,
        #     f"{hole_dia} mm", pen
        # )
    def addHorizontalDimension(self, x1, y1, x2, y2, text, pen,
       scene):
        scene.addLine(x1, y1, x2, y2, pen)
        arrow_size = int(self.arrowsize)
        ext_length = 10
        scene.addLine(x1, y1 - ext_length/2, x1, y1 + ext_length
           /2, pen)
```

```
        scene.addLine(x2, y2 - ext_length/2, x2, y2 + ext_length
            /2, pen)

        points_left = [
            (x1, y1),
            (x1 + arrow_size, y1 - arrow_size/2),
            (x1 + arrow_size, y1 + arrow_size/2)
        ]
        polygon_left = scene.addPolygon(QPolygonF([QPointF(x, y)
            for x, y in points_left]), pen)
        polygon_left.setBrush(QBrush(QColor('black')))

        points_right = [
            (x2, y2),
            (x2 - arrow_size, y2 - arrow_size/2),
            (x2 - arrow_size, y2 + arrow_size/2)
        ]
        polygon_right = scene.addPolygon(QPolygonF([QPointF(x, y)
            for x, y in points_right]), pen)
        polygon_right.setBrush(QBrush(QColor('black')))

        text_item = scene.addText(text)
        font = QFont()
        font.setPointSize(int(self.fontsize))
        text_item.setFont(font)

        if y1 < 0:
            text_item.setPos((x1 + x2) / 2 - text_item.
                boundingRect().width() / 2, y1 - 25)
        else:
            text_item.setPos((x1 + x2) / 2 - text_item.
                boundingRect().width() / 2, y1 + 5)
```

```
    def addVerticalDimension(self, x1, y1, x2, y2, text, pen,
        scene):
        scene.addLine(x1, y1, x2, y2, pen)
        arrow_size = int(self.arrowsize)
        ext_length = 10
        scene.addLine(x1 - ext_length/2, y1, x1 + ext_length/2,
            y1, pen)
        scene.addLine(x2 - ext_length/2, y2, x2 + ext_length/2,
            y2, pen)

        if y2 > y1:
            points_top = [
                (x1, y1),
                (x1 - arrow_size/2, y1 + arrow_size),
                (x1 + arrow_size/2, y1 + arrow_size)
            ]
            polygon_top = scene.addPolygon(QPolygonF([QPointF(x,
                y) for x, y in points_top]), pen)
            polygon_top.setBrush(QBrush(QColor('black')))

            points_bottom = [
                (x2, y2),
                (x2 - arrow_size/2, y2 - arrow_size),
                (x2 + arrow_size/2, y2 - arrow_size)
            ]
            polygon_bottom = scene.addPolygon(QPolygonF([QPointF(
                x, y) for x, y in points_bottom]), pen)
            polygon_bottom.setBrush(QBrush(QColor('black')))
        else:
            points_top = [
                (x2, y2),
                (x2 - arrow_size/2, y2 + arrow_size),
                (x2 + arrow_size/2, y2 + arrow_size)
            ]
```

```
            polygon_top = scene.addPolygon(QPolygonF([QPointF(x,
                y) for x, y in points_top]), pen)
            polygon_top.setBrush(QBrush(QColor('black')))

            points_bottom = [
                (x1, y1),
                (x1 - arrow_size/2, y1 - arrow_size),
                (x1 + arrow_size/2, y1 - arrow_size)
            ]
            polygon_bottom = scene.addPolygon(QPolygonF([QPointF(
                x, y) for x, y in points_bottom]), pen)
            polygon_bottom.setBrush(QBrush(QColor('black')))

        text_item = scene.addText(text)
        font = QFont()
        font.setPointSize(int(self.fontsize))
        text_item.setFont(font)

        if x1 < 0:
            text_item.setPos(x1 - 10 - text_item.boundingRect().
                width(), (y1 + y2) / 2 - text_item.boundingRect().
                height() / 2)
        else:
            text_item.setPos(x1 + 15, (y1 + y2) / 2 - text_item.
                boundingRect().height() / 2)
```

8.1.2 Code Explanation

Class: `B2Bcoverplate_capacity_details(QMainWindow)`

Overview

The `B2Bcoverplate_capacity_details` class is a PyQt5-based GUI tool for visualizing and analyzing beam-to-beam cover plate connections in structural engineering. It sup-

ports detailed 2D drawings for bolt layouts, spacing configurations, and capacity analysis (web and flange plates).

Constructor

```
def __init__(self, connection_obj, rows=3, cols=2, main=None)
```

Parameters:

- `connection_obj`: Design object containing spacing and capacity data
- `rows`: Number of bolt rows (default: 3)
- `cols`: Number of bolt columns (default: 2)
- `main`: Tuple with reference to main design and settings

Key Instance Variables

- `plate_length`, `plate_width`, `bolt_diameter`: Cover plate dimensions
- `pitch`, `End`, `Gauge`, `Edge`: Bolt spacing values
- `cols`, `rows`: Bolt layout configuration
- `web`, `drawing_type`: Configuration flags
- `web_capacity_details`, `flange_capacity_details`: Capacity data dictionaries
- `fontsize`, `arrowsize`: Adaptive display settings

Core Methods

`initUI(self)`

Initializes the window layout and content:

- Sets window title, geometry, and rendering properties
- Adds parameter panel and drawing scene
- Supports both single and dual drawing modes

- Adjusts scale based on plate dimensions

Layout Modes:

- Single: One drawing and one parameter panel
- Dual: Two drawings for web capacity analysis (left and right failure patterns)

`get_parameters(self, type_)`

Returns parameter dictionary based on drawing mode.

Parameters:

- `type_`: Tuple (`web_flag`, `drawing_type`)

Returns:

- Bolt spacing parameters (spacing mode)
- Capacity-specific parameters (capacity mode)

`createDrawing(self, type_, scene)`

Creates the 2D engineering drawing with:

- **Plate Outline:** White rectangle representing the cover plate
- **Bolt Holes:** Blue circular holes positioned via spacing rules
- **Failure Patterns:** Dashed lines for tension/shear failure zones
- **Dimensions:** Precise measurement annotations

Bolt Placement Algorithm:

- Symmetrical around center for aesthetic and structural clarity
- Adjusts for odd/even rows and columns
- Works for both web and flange plates

`addHorizontalDimension(x1, y1, x2, y2, text, pen, scene)`

Draws a horizontal dimension line:

- Adds arrowheads and extension lines
- Displays dimension value centered above/below the line
- Adjusts label position for overlap prevention

`addVerticalDimension(x1, y1, x2, y2, text, pen, scene)`

Draws a vertical dimension line:

- Similar to horizontal but vertically aligned
- Handles both upward and downward arrows
- Auto positions text for readability

Advanced Features

Adaptive Scaling

Drawing content is scaled based on plate length:

- **¿1200mm**: 35% scale
- **¿600mm**: 50% scale
- **600mm**: 100% scale

Dual Drawing Support

In web capacity analysis mode:

- Left and right failure patterns are drawn separately
- Each drawing uses a different parameter set

Usage Scenarios

1. **Spacing Analysis:** Verify bolt layout, edge distances, and overall plate geometry
2. **Capacity Analysis:** Visualize failure mechanisms and stress zones
3. **Design Review:** Validate user input and show design consistency

Technical Setup

Coordinate System

- Origin: Top-left (0,0)
- +X: Right, +Y: Down
- Units: Millimeters

Drawing Appearance

- Blue: Bolt Holes
- Black: Plate outline and dimension lines
- Dashed: Failure pattern zones

Dependencies

External

- `PyQt5`: GUI components and drawing
- `sys`: System parameters

Internal

- `Common`: Contains constants
- `additionalfns`: Utility calculations

Output

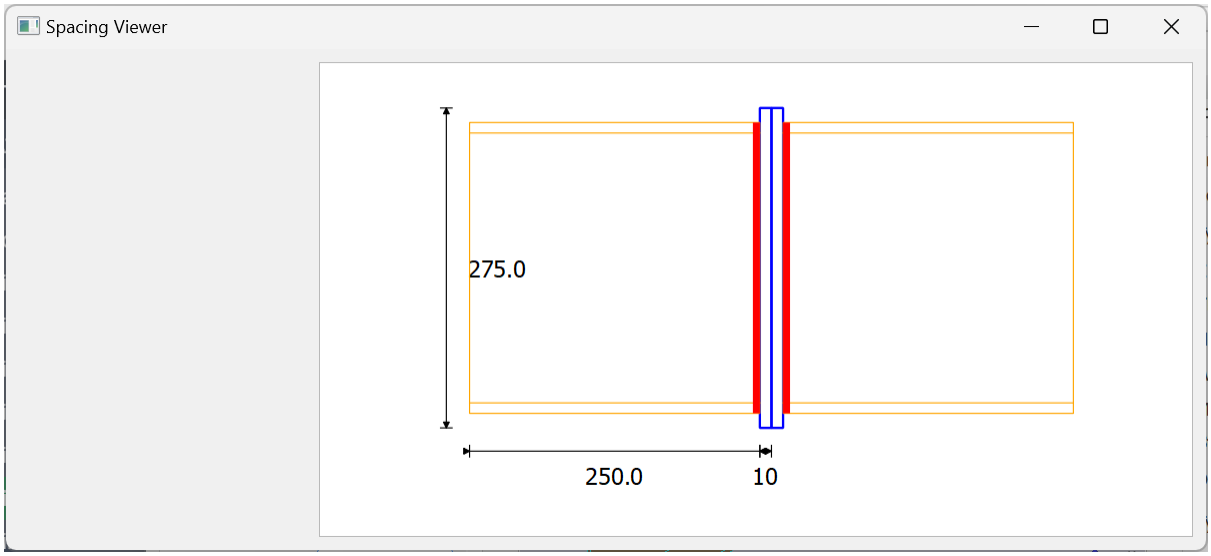
The application produces a high-quality engineering drawing showing:

- Cover plate layout and geometry
- Bolt arrangement and hole pattern
- Failure patterns (if applicable)
- Dimension annotations and parameter summaries

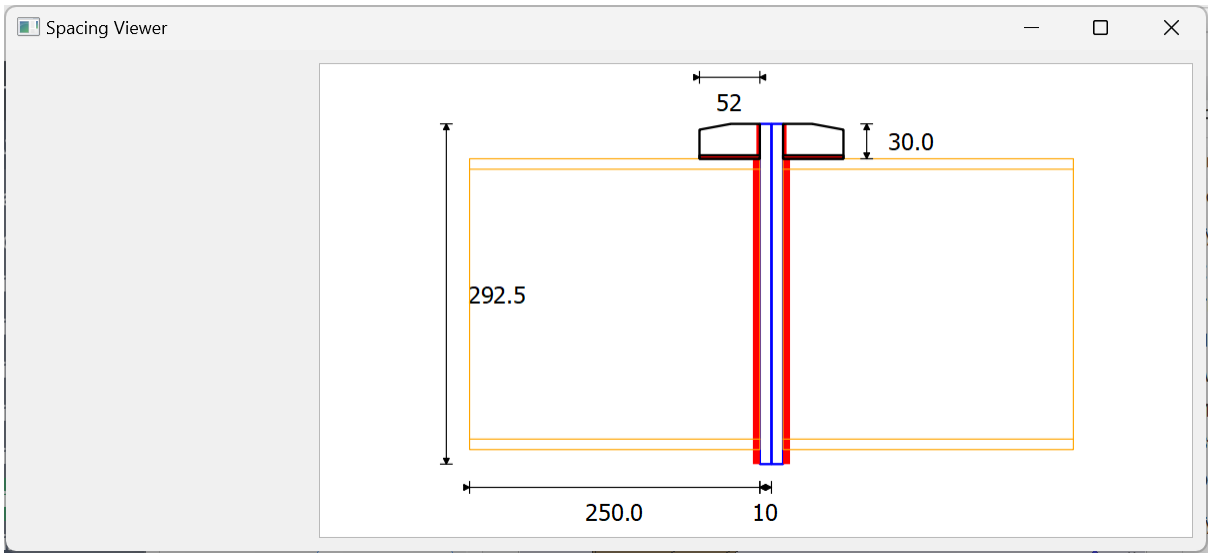
8.2 Beam to Beam Splice - End Plate

8.2.1 Window Images

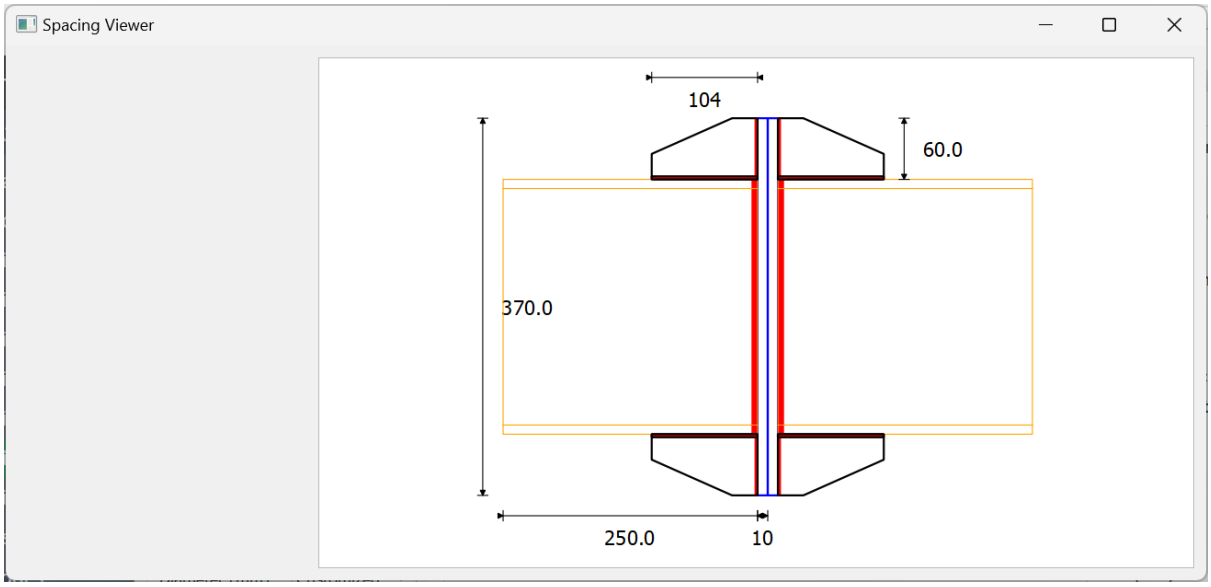
Flushed - Reversible Moment



Extended One Way - Irreversible Moment



Extended Both Ways - Reversible Moment



8.2.2 Code

```
import sys
from PyQt5.QtWidgets import (QApplication, QMainWindow, QWidget,
    QVBoxLayout,
                             QHBoxLayout, QLabel, QGraphicsView,
```

```
                                QGraphicsScene,QGraphicsRectItem,
                                    QFrame)
from PyQt5.QtGui import QPixmap
from PyQt5.QtCore import Qt, QRectF
from PyQt5.QtGui import QPainter, QPen, QFont, QColor
from PyQt5.QtGui import QPolygonF, QBrush
from PyQt5.QtCore import QPointF
from ..Common import *
from .additionalfns import calculate_total_width

try:
    pen_style_dash = Qt.PenStyle.DashLine
except AttributeError:
    raise RuntimeError("Your PyQt5 version does not support
       dashed lines via Qt.PenStyle.DashLine. Please update PyQt5
       .")

class B2Bcoverplate_capacity_details(QMainWindow):
    def __init__(self, connection_obj, rows=3, cols=2 , main =
       None):
        print(main)

        if main:
            self.drawing_type=main[2]
            self.web=main[1]
            web=main[1]
            main=main[0]
        super().__init__()
        self.connection = connection_obj
        # return
        data=main.output_values(main,True)
        print(type(main))
        dict1={i[0] : i[3] for i in data}
```

```
        for i in dict1:
            print(f'{i} : {dict1[i]}')


        if web==True:
            self.plate_length=dict1['Web_Plate.Height (mm)']
            self.plate_width=dict1['Web_Plate.Width']
            self.bolt_diameter=dict1['Bolt.Diameter']
            web_capcity=dict1['Web_plate.spacing'][1]
            print(web_capcity(main,True))
            data2=web_capcity(main,True)
            for i in range(len(data2)):
                print(f"{i} : {data2[i]}")
            self.pitch=data2[2][3]
            self.End=data2[3][3]
            self.Gauge=data2[4][3]
            self.Edge=data2[5][3]
            bolt_cap=dict1['Web Bolt.Capacities'][1]
            print(bolt_cap(main,True))
            bolt_cap=bolt_cap(main,True)
        elif web==False:
            self.plate_length=dict1['Flange_Plate.Width (mm)']
            self.plate_width=dict1['flange_plate.Length']
            self.bolt_diameter=dict1['Bolt.Diameter']
            flange_capcity=dict1['Flange_plate.spacing'][1]
            data2=flange_capcity(main,True)
            self.pitch=data2[2][3]
            self.End=data2[3][3]
            self.Gauge=data2[4][3]
            self.Edge=data2[5][3]
            bolt_cap=dict1['Bolt.Capacities'][1]
            print(bolt_cap(main,True))
```

```
            bolt_cap=bolt_cap(main,True)



        #capacity
        if web==True and self.drawing_type=="capacity":
            #web capacity details
            web_capacity_fnc=dict1['section.web_capacities'][1]
            web_capacity_val=web_capacity_fnc(main,True)
            self.web_capacity_details = {item[1]: float(item[3])
               for item in web_capacity_val if item[2] == '
               TextBox'}

        #capacity
        elif web==False and self.drawing_type=="capacity":
            #flange capacity details
            flange_capacity_fnc=dict1['section.flange_capacity'
               ][1]
            flange_capacity_val=flange_capacity_fnc(main,True)
            self.flange_capacity_details={item[1]: float(item[3])
                for item in flange_capacity_val if item[2] == '
               TextBox'}



        self.cols=bolt_cap[1][3]
        self.rows=bolt_cap[2][3]/self.cols
        self.initUI()


    def initUI(self):
        self.setWindowTitle('Bolt Pattern Generator')
        self.setGeometry(100, 100, 1050, 500)
```

```
        # Print summary (optional debug/log info)
        print(f"""
        ----------------------------------------
            Plate & Bolt Configuration Summary
        ----------------------------------------
        Plate Length            : {self.plate_length} mm
        Plate Width             : {self.plate_width} mm
        Bolt Diameter           : {self.bolt_diameter} mm

        Bolt Spacing Details:
        ---------------------
        Pitch Distance          : {self.pitch} mm
        End Distance            : {self.End} mm
        Gauge Distance          : {self.Gauge} mm
        Edge Distance           : {self.Edge} mm

        Bolt Arrangement:
        -----------------
        Number of Columns       : {self.cols}
        Number of Rows          : {self.rows}
        """)

        # Main layout
        main_layout = QHBoxLayout()

        # Left panel for parameter display
        left_panel = QWidget()
        left_layout = QVBoxLayout()

        # Get parameter dictionary
        params = self.get_parameters((self.web,self.drawing_type)
            )
        count=0
```

```
        for key, value in params.items():
            if self.web==False and self.drawing_type=="capacity":
                param_layout = QHBoxLayout()
                space_label = QLabel('  ')
                param_label = QLabel(f'{key.title()} (mm):')
                value_label = QLabel(f'{value}')
                param_layout.addWidget(param_label)
                param_layout.addWidget(value_label)
                left_layout.addLayout(param_layout)
                # Add a blank label for vertical spacing
                left_layout.addWidget(QLabel(''))
            elif self.web==True and self.drawing_type=="capacity"
               :
                param_layout = QHBoxLayout()
                space_label = QLabel('  ')
                param_label = QLabel(f'{key.title()} (mm):')
                value_label = QLabel(f'{value}')
                param_layout.addWidget(param_label)
                param_layout.addWidget(value_label)
                left_layout.addLayout(param_layout)
                # Add a blank label for vertical spacing
                left_layout.addWidget(QLabel(''))

                count+=1
            else:
                param_layout = QHBoxLayout()
                param_label = QLabel(f'{key.title()} (mm):')
                value_label = QLabel(f'{value}')
                param_layout.addWidget(param_label)
                param_layout.addWidget(value_label)
                left_layout.addLayout(param_layout)

        left_layout.addStretch()
        left_panel.setLayout(left_layout)
```

```
        # Determine font and arrow size based on plate size
        self.fontsize = 10
        self.arrowsize = 10
        if self.plate_length > 1200 or self.plate_width > 1200:
            self.fontsize = 12
            self.arrowsize = 12
        elif self.plate_length > 600 or self.plate_width > 600:
            self.fontsize = 7.5
            self.arrowsize = 7.5

        if self.web == True and self.drawing_type == "capacity":
            # Two drawings, each with its own parameter set,
               separated by a horizontal line
            self.scene1 = QGraphicsScene()
            self.view1 = QGraphicsView(self.scene1)
            self.view1.setRenderHint(QPainter.Antialiasing)

            self.scene2 = QGraphicsScene()
            self.view2 = QGraphicsView(self.scene2)
            self.view2.setRenderHint(QPainter.Antialiasing)

            self.createDrawing((self.web, self.drawing_type),
               self.scene1)
            self.createDrawing((self.web, self.drawing_type),
               self.scene2)

            if self.plate_length > 1200 or self.plate_width >
               1200:
                self.view1.resetTransform()
                self.view1.scale(0.35, 0.35)
                self.view2.resetTransform()
                self.view2.scale(0.35, 0.35)
```

```
        elif self.plate_length > 600 or self.plate_width >
            600:
             self.view1.resetTransform()
             self.view1.scale(0.5, 0.5)
             self.view2.resetTransform()
             self.view2.scale(0.5, 0.5)

        # Split parameters into two groups (first two, next
            two)
        params = list(self.get_parameters((self.web, self.
            drawing_type)).items())
        params1 = params[:2]
        params2 = params[2:]

        # Section 1: first two parameters and first drawing
        section1_layout = QHBoxLayout()
        section1_text_widget = QWidget()
        section1_text_layout = QVBoxLayout(
            section1_text_widget)
        param_label = QLabel('Failure Pattern due to tension
            in Member and Plate')
        param_label.setFont(QFont('Arial',12,QFont.Bold))
        section1_text_layout.addWidget(param_label)
        for key, value in params1:
             param_label = QLabel(f'{key}: {value}')
             section1_text_layout.addWidget(param_label)
        section1_text_layout.addStretch()
        section1_text_widget.setMinimumWidth(180)
        section1_layout.addWidget(section1_text_widget, 1)
        section1_layout.addWidget(self.view1, 2)

        # Section 2: next two parameters and second drawing
        section2_layout = QHBoxLayout()
        section2_text_widget = QWidget()
```

```
            section2_text_layout = QVBoxLayout(
                section2_text_widget)
            param_label = QLabel('Failure Pattern due to tension
                in Member and Plate')
            param_label.setFont(QFont('Arial',12,QFont.Bold))
            section2_text_layout.addWidget(param_label)
            for key, value in params2:
                param_label = QLabel(f'{key}: {value}')
                section2_text_layout.addWidget(param_label)
            section2_text_layout.addStretch()
            section2_text_widget.setMinimumWidth(180)
            section2_layout.addWidget(section2_text_widget, 1)
            section2_layout.addWidget(self.view2, 2)

            # Horizontal line between sections
            line = QFrame()
            line.setFrameShape(QFrame.HLine)
            line.setFrameShadow(QFrame.Sunken)

            # Main vertical layout
            main_vlayout = QVBoxLayout()
            main_vlayout.addLayout(section1_layout)
            main_vlayout.addWidget(line)
            main_vlayout.addLayout(section2_layout)

            self.view1.setMaximumWidth(400)
            self.view2.setMaximumWidth(400)

            main_widget = QWidget()
            main_widget.setLayout(main_vlayout)
            self.setCentralWidget(main_widget)
        else:
            # Only one drawing (original layout)
            left_panel = QWidget()
```

```
            left_layout = QVBoxLayout()
            params = self.get_parameters((self.web, self.
               drawing_type))
            for key, value in params.items():
                if self.web==False and self.drawing_type=="
                   capacity":
                    param_layout = QHBoxLayout()
                    space_label = QLabel('  ')
                    param_label = QLabel(f'{key.title()} (mm):')
                    value_label = QLabel(f'{value}')
                    param_layout.addWidget(param_label)
                    param_layout.addWidget(value_label)
                    left_layout.addLayout(param_layout)
                    left_layout.addWidget(QLabel(''))
                else:
                    param_layout = QHBoxLayout()
                    param_label = QLabel(f'{key.title()} (mm):')
                    value_label = QLabel(f'{value}')
                    param_layout.addWidget(param_label)
                    param_layout.addWidget(value_label)
                    left_layout.addLayout(param_layout)
            left_layout.addStretch()
            left_panel.setLayout(left_layout)

            self.scene = QGraphicsScene()
            self.view = QGraphicsView(self.scene)
            self.view.setRenderHint(QPainter.Antialiasing)
            self.createDrawing((self.web, self.drawing_type),
               self.scene)

            if self.plate_length > 1200 or self.plate_width >
               1200:
                self.view.resetTransform()
                self.view.scale(0.35, 0.35)
```

```
            elif self.plate_length > 600 or self.plate_width >
                600:
                 self.view.resetTransform()
                 self.view.scale(0.5, 0.5)

            main_layout = QHBoxLayout()
            main_layout.addWidget(left_panel, 1)
            main_layout.addWidget(self.view, 3)
            main_widget = QWidget()
            main_widget.setLayout(main_layout)
            self.setCentralWidget(main_widget)

        # Automatically adjust view to fit scene
    def get_parameters(self,type_):
        if (type_[0]==True and type_[1]=="spacing") or (type_
            [0]==False and type_[1]=="spacing") :
             return {
             'Plate Length': self.plate_length,
             'Plate Width': self.plate_width,
             'Bolt Diameter': self.bolt_diameter,
             'Pitch Distance': self.pitch,
             'End Distance': self.End,
             'Gauge Distance': self.Gauge,
             'Edge Distance': self.Edge,
             'Number of Columns': self.cols,
             'Number of Rows': self.rows
             }
        elif type_[0]==True and type_[1]=="capacity":
             return self.web_capacity_details
        elif type_[0]==False and type_[1]=="capacity":
             return self.flange_capacity_details

    def createDrawing(self, type_, scene):
```

```
try:
    plate_length = float(self.plate_length)
    plate_width = float(self.plate_width)
except (TypeError, ValueError):
    print("Invalid plate dimensions")
    return
rect = QRectF(0, 0, plate_length, plate_width)
# Create a rectangle item
rect_item = QGraphicsRectItem(rect)

# Set pen and brush (black border, transparent fill)
pen = QPen(QColor('black'))
pen.setWidth(2)
rect_item.setPen(pen)
rect_item.setBrush(QBrush())  # Default is NoBrush

# Add rectangle to the scene
scene.addItem(rect_item)



# Extract parameters
outline_pen = QPen(QColor('black'))
outline_pen.setWidth(1)

# === Draw Base Plate Rectangle ===
rect_item = QGraphicsRectItem(QRectF(0, 0, plate_length,
   plate_width))
rect_item.setPen(outline_pen)
rect_item.setBrush(QBrush(QColor('white')))
scene.addItem(rect_item)
dashed_pen = QPen(QColor('black'))
dashed_pen.setStyle(pen_style_dash)
```

```
        dashed_pen.setWidth(2)
        if type_[0]==False and type_[1]=="capacity":

            #top drawing
            scene.addLine(self.Edge, 0, self.Edge, self.End,
                dashed_pen)

            scene.addLine(self.Edge, self.End, plate_length, self
                .End, dashed_pen)


            #bottom drawing
            scene.addLine(self.Edge, plate_width, self.Edge,
                plate_width-self.End, dashed_pen)

            scene.addLine(self.Edge, plate_width-self.End,
                plate_length, plate_width-self.End, dashed_pen)

        elif type_[0]==True and type_[1]=="capacity" and scene==
            self.scene1:

            scene.addLine(self.End, self.Edge, self.End,
                plate_width-self.End, dashed_pen)

            scene.addLine(self.End, self.Edge, plate_length, self
                .Edge, dashed_pen)

            scene.addLine(self.End, plate_width-self.Edge,
                plate_length, plate_width-self.Edge, dashed_pen)


        elif type_[0]==True and type_[1]=="capacity" and scene==
            self.scene2:
```

```
            scene.addLine(0, self.Edge, plate_length-self.End,
                self.Edge, dashed_pen)

            scene.addLine(plate_length-self.End, self.Edge,
                plate_length-self.End, plate_width, dashed_pen)


        # === Center of the base plate ===
        center_x = plate_length / 2
        center_y = plate_width / 2
        self.addHorizontalDimension(
            0, -30,  # x1 at left edge, y above plate
            self.plate_length, -30,  # x2 at right edge, same y
            f"{self.plate_length} mm", pen, scene
        )

        # Vertical dimension for plate width (to the left of the
            plate)
        self.addVerticalDimension(
            self.plate_length+30, 0,  # x left of plate, y1 at
                top
            self.plate_length+30, self.plate_width,  # x2 same,
                y2 at bottom
            f"{self.plate_width} mm", pen, scene
        )

        rows=int(self.rows)
        cols=int(self.cols)
        pitch=self.pitch
        gauge=self.Gauge
        end=self.End
        edge=self.Edge
        hole_dia = self.bolt_diameter
        radius = hole_dia / 2
```

```
        y_center = end  # Y position is fixed for top row
        # Center row if rows is odd
        outline_pen = QPen(QColor('blue'))
        outline_pen.setWidth(1)
        if rows % 2 != 0:
            y_center = self.plate_width / 2
            for i in range(cols // 2):
                x_center = edge + i * gauge
                scene.addEllipse(
                    x_center - radius,
                    y_center - radius,
                    hole_dia,
                    hole_dia,
                    outline_pen,
                )

            for i in range(cols // 2):
                x_center = self.plate_length - edge - i * gauge
                scene.addEllipse(
                    x_center - radius,
                    y_center - radius,
                    hole_dia,
                    hole_dia,
                    outline_pen,
                )

            # Center bolt if cols is also odd
            if cols % 2 != 0:
                x_center = self.plate_length / 2
                scene.addEllipse(
                    x_center - radius,
                    y_center - radius,
                    hole_dia,
                    hole_dia,
```

```
                outline_pen,
            )

        # Center column if cols is odd (and rows is even)
        if cols % 2 != 0 and rows % 2 == 0:
            for j in range(rows // 2):
                y_center_top = end + j * pitch
                y_center_bottom = self.plate_width - end - j *
                    pitch

                x_center = self.plate_length / 2
                scene.addEllipse(
                    x_center - radius,
                    y_center_top - radius,
                    hole_dia,
                    hole_dia,
                    outline_pen,
                )
                scene.addEllipse(
                    x_center - radius,
                    y_center_bottom - radius,
                    hole_dia,
                    hole_dia,
                    outline_pen,
                )

        # Draw left half bolts
        for row in range(int(rows)):
            if row < rows // 2:
                y_center = end + row * pitch
            else:
                row_from_bottom = row - rows // 2
                y_center = self.plate_width - end -
                    row_from_bottom * pitch
```

```
            # Left half bolts
            for i in range(cols // 2):
                x_center = edge + i * gauge
                scene.addEllipse(
                    x_center - radius,
                    y_center - radius,
                    hole_dia,
                    hole_dia,
                    outline_pen,
                )

            # Right half bolts
            for i in range(cols // 2):
                x_center = self.plate_length - edge - i * gauge
                scene.addEllipse(
                    x_center - radius,
                    y_center - radius,
                    hole_dia,
                    hole_dia,
                    outline_pen,
                )
        self.addHorizontalDimension(
            0, self.plate_width+10,  # x1 at left edge, y above
               plate
            self.Edge, self.plate_width+10,  # x2 at right edge,
               same y
            f"{self.Edge} mm", pen, scene
        )
        self.addVerticalDimension(
            -10, 0,  # x left of plate, y1 at top
            -10, self.End,  # x2 same, y2 at bottom
            f"{self.End} mm", pen, scene
        )
```

```
        # self.addHorizontalDimension(
        #     self.Edge-hole_dia/2, 10,
        #     self.Edge+hole_dia/2,10,
        #     f"{hole_dia} mm", pen
        # )
    def addHorizontalDimension(self, x1, y1, x2, y2, text, pen,
        scene):
        scene.addLine(x1, y1, x2, y2, pen)
        arrow_size = int(self.arrowsize)
        ext_length = 10
        scene.addLine(x1, y1 - ext_length/2, x1, y1 + ext_length
            /2, pen)
        scene.addLine(x2, y2 - ext_length/2, x2, y2 + ext_length
            /2, pen)

        points_left = [
            (x1, y1),
            (x1 + arrow_size, y1 - arrow_size/2),
            (x1 + arrow_size, y1 + arrow_size/2)
        ]
        polygon_left = scene.addPolygon(QPolygonF([QPointF(x, y)
            for x, y in points_left]), pen)
        polygon_left.setBrush(QBrush(QColor('black')))

        points_right = [
            (x2, y2),
            (x2 - arrow_size, y2 - arrow_size/2),
            (x2 - arrow_size, y2 + arrow_size/2)
        ]
        polygon_right = scene.addPolygon(QPolygonF([QPointF(x, y)
            for x, y in points_right]), pen)
        polygon_right.setBrush(QBrush(QColor('black')))

        text_item = scene.addText(text)
```

```
        font = QFont()
        font.setPointSize(int(self.fontsize))
        text_item.setFont(font)

        if y1 < 0:
            text_item.setPos((x1 + x2) / 2 - text_item.
                boundingRect().width() / 2, y1 - 25)
        else:
            text_item.setPos((x1 + x2) / 2 - text_item.
                boundingRect().width() / 2, y1 + 5)

    def addVerticalDimension(self, x1, y1, x2, y2, text, pen,
        scene):
        scene.addLine(x1, y1, x2, y2, pen)
        arrow_size = int(self.arrowsize)
        ext_length = 10
        scene.addLine(x1 - ext_length/2, y1, x1 + ext_length/2,
            y1, pen)
        scene.addLine(x2 - ext_length/2, y2, x2 + ext_length/2,
            y2, pen)

        if y2 > y1:
            points_top = [
                (x1, y1),
                (x1 - arrow_size/2, y1 + arrow_size),
                (x1 + arrow_size/2, y1 + arrow_size)
            ]
            polygon_top = scene.addPolygon(QPolygonF([QPointF(x,
                y) for x, y in points_top]), pen)
            polygon_top.setBrush(QBrush(QColor('black')))

            points_bottom = [
                (x2, y2),
                (x2 - arrow_size/2, y2 - arrow_size),
```

```
                (x2 + arrow_size/2, y2 - arrow_size)
            ]
            polygon_bottom = scene.addPolygon(QPolygonF([QPointF(
                x, y) for x, y in points_bottom]), pen)
            polygon_bottom.setBrush(QBrush(QColor('black')))
        else:
            points_top = [
                (x2, y2),
                (x2 - arrow_size/2, y2 + arrow_size),
                (x2 + arrow_size/2, y2 + arrow_size)
            ]
            polygon_top = scene.addPolygon(QPolygonF([QPointF(x,
                y) for x, y in points_top]), pen)
            polygon_top.setBrush(QBrush(QColor('black')))

            points_bottom = [
                (x1, y1),
                (x1 - arrow_size/2, y1 - arrow_size),
                (x1 + arrow_size/2, y1 - arrow_size)
            ]
            polygon_bottom = scene.addPolygon(QPolygonF([QPointF(
                x, y) for x, y in points_bottom]), pen)
            polygon_bottom.setBrush(QBrush(QColor('black')))

        text_item = scene.addText(text)
        font = QFont()
        font.setPointSize(int(self.fontsize))
        text_item.setFont(font)

        if x1 < 0:
            text_item.setPos(x1 - 10 - text_item.boundingRect().
                width(), (y1 + y2) / 2 - text_item.boundingRect().
                height() / 2)
        else:
```

```
            text_item.setPos(x1 + 15, (y1 + y2) / 2 - text_item.
               boundingRect().height() / 2)
```

8.2.3 Code Explanation

Class: `B2Bcoverplate_capacity_details(QMainWindow)`

Overview

The `B2Bcoverplate_capacity_details` class is a PyQt5-based GUI module for visualizing and analyzing beam-to-beam cover plate connections in steel structures. It provides accurate 2D drawings for both bolt spacing and capacity analysis of web and flange cover plates.

Class Definition

```
class B2Bcoverplate_capacity_details(QMainWindow)
```

Key Dependencies

- `PyQt5`: Core GUI framework
- `PyQt5.QtWidgets`: UI components (QMainWindow, QLabel, QGraphicsView, etc.)
- `PyQt5.QtGui`: Graphics (QPen, QBrush, QFont)
- `PyQt5.QtCore`: Qt core elements (QPointF, QRectF, etc.)

Constructor

```
def __init__(self, connection_obj, rows=3, cols=2, main=None)
```

Parameters:

- `connection_obj`: Connection object containing design data
- `rows`, `cols`: Number of bolt rows and columns (default: 3x2)
- `main`: Tuple of (`main_app`, `web_flag`, `drawing_type`)

Important Instance Variables

- `plate_length`, `plate_width`: Cover plate dimensions (mm)
- `bolt_diameter`, `pitch`, `Gauge`, `Edge`, `End`: Bolt spacing configuration
- `rows`, `cols`: Bolt arrangement counts
- `web`, `drawing_type`: Drawing flags
- `fontsize`, `arrowsize`: Scaled font and arrow sizes
- `web_capacity_details`, `flange_capacity_details`: Capacity data

Core Methods

`initUI()`

Purpose: Sets up the window, GUI layout, and scene initialization.

Features:

- Sets window geometry, title, and layout
- Chooses drawing scale based on plate size:

```
if plate_length > 1200 or plate_width > 1200:
    scale = 0.35
elif plate_length > 600 or plate_width > 600:
    scale = 0.5
else:
    scale = 1.0
```

- Initializes either single or dual drawing modes

`get_parameters(type_)`

Purpose: Extracts parameter dictionary depending on mode and plate type.

Parameters:

- `type_`: A tuple (`web_flag`, `"spacing"/"capacity"`)

Returns:

- Spacing mode: Plate geometry and bolt spacing
- Capacity mode: Web or flange capacity parameters

`createDrawing(type_, scene)`

Purpose: Generates the complete technical drawing.

Components:

- Plate outline (black rectangle)
- Bolt holes (blue circles) placed via bolt algorithm
- Failure pattern lines (dashed)
- Dimension lines (horizontal and vertical)

Bolt Placement Logic:

1. Calculates center bolt row/column if odd
2. Distributes remaining rows and columns symmetrically
3. Applies pitch and gauge to locate each bolt

`addHorizontalDimension(x1, y1, x2, y2, text, pen, scene)`

Purpose: Adds a horizontal dimension line with:

- Arrowheads
- Extension lines
- Positioned text labels

`addVerticalDimension(x1, y1, x2, y2, text, pen, scene)`

Purpose: Adds vertical dimensions with:

- Auto-adjusted arrows and text position
- Support for upward/downward annotations

Drawing Modes

Spacing Mode

- Shows all bolt positions, plate boundaries, and spacing
- Suitable for visual layout verification
- Rendered in a single view

Capacity Mode

- Adds failure lines for capacity visualization
- **Web Plates:** Drawn twice for left and right failure patterns
- **Flange Plates:** Single failure pattern (top-bottom)

Failure Pattern Visualization

- **Web Drawing 1:** Failure line from left
- **Web Drawing 2:** Failure line from right
- **Flange Drawing:** Top and bottom tension/shear lines

Error Handling

- Catches TypeErrors or missing data gracefully
- Supports version-specific handling for dashed line rendering
- Verifies valid dimensions and coordinates before drawing

Usage Example

```
# For web capacity mode
window = B2Bcoverplate_capacity_details(
    connection_obj=connection,
    main=(main_app, True, "capacity")
```

```
)


# For flange spacing mode
window = B2Bcoverplate_capacity_details(
    connection_obj=connection,
    main=(main_app, False, "spacing")
)
```

Output

The drawing scene displays:

- Plate geometry with actual dimensions
- Bolt hole layout with accurate spacing
- Failure lines for capacity inspection (if applicable)
- Annotations for pitch, edge, gauge, end, and total distances

Chapter 9

Conclusions

9.1 Tasks Accomplished

Task Summary: All Completed Tasks

1. Welded Lap Joint

Objective: Create a CAD model for a welded lap joint using `pythonocc`.

Tasks Done:

- Developed the 3D model of the welded lap joint.
- Implemented a Python script to automate beam-to-column shear connection design.
- Contributed documentation for Osdag developer/user manual.

Tools Used: `pythonocc`, Osdag, IS Code, LaTeX.

2. Plate Girder CAD Model

Objective: Design a CAD model of a steel plate girder.

Tasks Done:

- Created an I-section model with stiffeners and flanges.
- Replaced computationally heavy `BRepAlgoAPIFuse` with `BRepBuilder()` for performance.

- Generated accurate geometry with chamfered stiffeners and welds.

Output: Highly modular, realistic, and colored 3D girder model.

Use Cases: Structural modeling, fabrication, education.

3. CAD Model Saving for Animation

Objective: Export Osdag-generated 3D CAD models to `.gltf` format.

Tasks Done:

- Located the right CAD generation point in `BCE_nutBoltPlacement.py`.
- Used `write_gltf_file` from OCC to save models individually (beams, columns, bolts, etc.).
- Combined all components using `BRepAlgoAPI_Fuse`.

Purpose: Enable use of models in animation tools like Blender.

4. LCC Graph Visualizations

You developed several interactive graph modules using **D3.js + PySide6** to visualize lifecycle cost data for bridges.

A. Horizontal Bar Graph

- Shows cost comparison between PSC and Steel bridges.
- 11 cost categories visualized side-by-side.
- Interactive hover, tooltips, and animations.

B. Pie Chart

- Displays the percentage breakdown of total lifecycle cost.
- Includes interactivity like legend toggling, hover highlights, and smooth transitions.

C. Bubble Graph

- Visualizes carbon emission cost types using bubble sizes.
- Shows cost and percentage in hover tooltips.
- Interactive design with D3 animations.

D. Radial Bar Graph

- Circular visualization of cost distribution across four lifecycle stages.
- Used in reporting and storytelling.

Tools Used: PySide6, Pandas, D3.js, HTML/CSS.

5. Spacing & Capacity Details Window for Shear Connection - Fin Plate

You created two PyQt5 GUI tools for fin plate connections.

A. Spacing Window

- Visualizes bolt pattern with dimension annotations.
- Uses QGraphicsScene to draw plates, holes, welds.
- Extracts parameters from Osdag design outputs.

B. Capacity Window

- Displays shear/tension failure patterns visually.
- Shows capacities like shear yield, rupture, block shear, tension capacities, etc.
- Organizes values and graphics in a dual-panel layout.

Final Output Types

- Multiple GUI windows for:
  - CAD model visualization
  - Bolt pattern drawing
  - Capacity checking
  - Graphical analytics
- Exported 3D models in `.gltf` for animation
- Technical LaTeX documentation with images, diagrams, and code explanations

9.2 Skills Developed

I gained skills the following skills during the course of 4 months in this internship,

- Generating CAD Models using Python OCC
- Deep understanding of classes and objects in python
- Team work
- Working with Javascript
- Debugging code
- Github
- Working with Opensource project
- pyqt 5

Chapter A

Appendix

A.1 Work Reports

Internship Work Report

Name:	Harshan S
Project:	Osdag
Internship:	FOSSEE Semester Long Internship 2025

DATE	DAY	TASK	Hours Worked
13-Feb-2025	Thursday	Tried Installing Osdag on my Laptop (Windows 11) using miniconda 3, got some installation error	4.5
14-Feb-2025	Friday	Was trying to fix the issues faced, got error in pyqt5 module	4.5
15-Feb-2025	Saturday	Fixed the pyqt5 issues, got database error while trying ro run osdag.osdagMainPage, asked about the issue in discord channel	5
16-Feb-2025	Sunday	Was working in the missing database error	4
17-Feb-2025	Monday	got fix in the missing database error from the semester long channel of osdag, had a meeting with Parth sir, Sachin, Aryan	4.5
18-Feb-2025	Tuesday	trying to read through the files, of how the osdag files are structured, had a meet with CAD team	5
19-Feb-2025	Wednesday	created a local branch with the repo(https://github.com/AjinkyaDahale/Osdag/tree/purlin-module) shared in discord channel.	5.5
20-Feb-2025	Thursday	discussion regarding task 1 with my teammate Sachin	6
21-Feb-2025	Friday	started to read the files of purlin module as a reference by aryan to understand general working	6
22-Feb-2025	Saturday	working with task 1	4
23-Feb-2025	Sunday	made document for task 1, with Sachin	4
24-Feb-2025	Monday	finished the task 1 along with Sachin	4
25-Feb-2025	Tuesday	started with task2, welded lap joint	3.5
26-Feb-2025	Wednesday	working with welded lap joint	4.5
27-Feb-2025	Thursday	Had meeting with CAD team, finished welded lap, was working with the minor issues stated in meeting	4
28-Feb-2025	Friday	Was in Holiday with permission due to exams	
1-Mar-2025	Saturday	Was in Holiday with permission due to exams	
2-Mar-2025	Sunday	Was in Holiday with permission due to exams	
3-Mar-2025	Monday	Was in Holiday with permission due to exams	
4-Mar-2025	Tuesday	Was in Holiday with permission due to exams	
5-Mar-2025	Wednesday	lap joint model was made, refined further with the image shared in discord channel	4
6-Mar-2025	Thursday	lap joint model was made, needed to confirm with the dimensions	2
7-Mar-2025	Friday	Discussed in the meeting regarding the dimensions, fixed the dimensions. New task of creating plate girder was assigned. Autocad file was shared in discod channel, used it as a reference to understand	4.5
8-Mar-2025	Saturday	working with plate girder model	4
9-Mar-2025	Sunday	-	
10-Mar-2025	Monday	created the model which was there as per the autocad file(plate girder)	3.5
11-Mar-2025	Tuesday	attended the CAD team meeting, new things were asked to add in the plate girder model	3
12-Mar-2025	Wednesday	working on with optimisation of the model rendering time	4.5
13-Mar-2025	Thursday	made the changes which was said on 11/3/25 meeting, like chamfer in the stiffner plate and outstand length, finished the model as per the requirements said on 11/3/25 meeting	4
14-Mar-2025	Friday	shared the image of plate girder in discord channel	
15-Mar-2025	Saturday	Was asked to add weld at the place where stiffner plate was there and was asked to add red color to it	3
16-Mar-2025	Sunday	confirmed where exactly in the plate girder model to add weld, and started to work on it	3.5
17-Mar-2025	Monday	worked on the weld, weld was added between the flange and stiffener plate, was asked to wait till the next meeting for further correction in the model	3.5
18-Mar-2025	Tuesday	was asked to share the image of the double lap joint and work on it	4
19-Mar-2025	Wednesday	changes suggested on double lap joint, regarding dimensions, working in the model	4
20-Mar-2025	Thursday	shared the image of double lap joint in discord channel, was working in the model	3
21-Mar-2025	Friday	was unable to attend the CAD team meet	
22-Mar-2025	Saturday	fixed bugs in the mis-placement of the stiffner plate	2.5
23-Mar-2025	Sunday	working on weld size, and weld placement in horizontal	4
24-Mar-2025	Monday	working on weld placement in vertical	1.5
25-Mar-2025	Tuesday	working on fixing the issues got while placement of vertical weld	2
26-Mar-2025	Wednesday	was working on chamfer in the stifferener plate, as first and last plate was not getting chamfered	1.5
27-Mar-2025	Thursday	was replacing I section with 3 plates, as Parth Sir said there will be varying flange thickness and width	4

Internship Work Report

Name:		Harshan S	
Project:		Osdag	
Internship:		FOSSEE Semester Long Internship 2025	
28-Mar-2025	Friday	bugs encountered while using 3 individual plates, one plate was flying, fixed that issue	2
29-Mar-2025	Saturday	was asked to add colors/materials to the web plate and stiffner plates	4
30-Mar-2025	Sunday	working on adding color for web by referring files sent by Aryan	4.5
31-Mar-2025	Monday	added color for web, and working on the optimising the model	3
1-Apr-2025	Tuesday	added material NOM_ALUMINIUM for stiffener plate	3.5
2-Apr-2025	Wednesday	optimising the model by removing fuse inside for loop and tried saving model inside list and fused finally	4
3-Apr-2025	Thursday	optimising the model	4
4-Apr-2025	Friday	optimising the model rendering	3.5
5-Apr-2025	Saturday	optimising the model rendering	3
6-Apr-2025	Sunday	optimising the model rendering	3
7-Apr-2025	Monday	optimising the model rendering	3
8-Apr-2025	Tuesday	attended the meet, and was asked to make the width of stiffener plate based on the top and bottom flange	2.5
9-Apr-2025	Wednesday	optimising the model by changing functions	4
10-Apr-2025	Thursday	optimising the model	3
11-Apr-2025	Friday	fixing bugs in model, which was formed while optimising the model	3.5
12-Apr-2025	Saturday	optimising the model	3
13-Apr-2025	Sunday	cleaning up the code, by creating functions	5
14-Apr-2025	Monday	changed the algo fuse approach to combine the model	4
15-Apr-2025	Tuesday	shared the code in discord	2
16-Apr-2025	Wednesday	working with optimisation of the model generation by various other approaches	4
17-Apr-2025	Thursday	working with optimisation of the model	3
18-Apr-2025	Friday	optimisation of the model	4
19-Apr-2025	Saturday	optimisation of the model by removing unwanted variables which was hardcoded	3.5
20-Apr-2025	Sunday	refining the model's code	4
21-Apr-2025	Monday	Was in Holiday with permission due to exams	
22-Apr-2025	Tuesday	Was in Holiday with permission due to exams	
23-Apr-2025	Wednesday	Was in Holiday with permission due to exams	
24-Apr-2025	Thursday	Was in Holiday with permission due to exams	
25-Apr-2025	Friday	Was in Holiday with permission due to exams	
26-Apr-2025	Saturday	reading Aryan's work on CAD integration with OsdagUI	4
27-Apr-2025	Sunday	trying to work with CAD integration	4.5
28-Apr-2025	Monday	optimisation of welded lap joint	3
29-Apr-2025	Tuesday	optimisation of welded double lap joint	3.5
30-Apr-2025	Wednesday	color of individual parts of lap joint for single lap joint was fixed	4
1-May-2025	Thursday	color of individual parts of lap joint for double lap joint was fixed	3.5
2-May-2025	Friday	optimising the lap joint model	3
3-May-2025	Saturday	optimising the double lap joint model	4
4-May-2025	Sunday	re formattin the code of plate girder by adding class to it, in order to integrate with the Osdag UI	4.5
5-May-2025	Monday	Had meeting with Neela Lakshmi, regarding the plate girder model integration, and worked	3
6-May-2025	Tuesday	Cad integration of plate girder model	4
7-May-2025	Wednesday	Was in Holiday with permission due to exams(end semester examinations), had meeting with Neela Lakshmi	
8-May-2025	Thursday	Was in Holiday with permission due to exams(end semester examinations)	
9-May-2025	Friday	Was in Holiday with permission due to exams(end semester examinations)	
10-May-2025	Saturday	Was in Holiday with permission due to exams(end semester examinations)	
11-May-2025	Sunday	Was in Holiday with permission due to exams(end semester examinations)	
12-May-2025	Monday	Was in Holiday with permission due to exams(end semester examinations)	
13-May-2025	Tuesday	Was in Holiday with permission due to exams(end semester examinations)	
14-May-2025	Wednesday	Was in Holiday with permission due to exams(end semester examinations)	
15-May-2025	Thursday	Was in Holiday with permission due to exams(end semester examinations)	
16-May-2025	Friday	Was in Holiday with permission due to exams(end semester examinations)	
17-May-2025	Saturday	Was in Holiday with permission due to exams(end semester examinations)	

Internship Work Report

Name:		Harshan S	
Project:		Osdag	
Internship:		FOSSEE Semester Long Internship 2025	
18-May-2025	Sunday	Was in Holiday with permission due to exams(end semester examinations)	
19-May-2025	Monday	Was in Holiday with permission due to exams(end semester examinations), had meeting with Ne	3
20-May-2025	Tuesday	tried CAD integration by what I understood from the meeting I had with Neela Lakshmi	4.5
21-May-2025	Wednesday	worked with CAD integration and also tried optimising the model	5.5
22-May-2025	Thursday	Had meeting with Aryan regarding CAD integration, and worked with the integration part	5
23-May-2025	Friday	Tried integrating the CAD model by making changes in ui_template.py, common_logic.py, created custom class for CAD model	6
24-May-2025	Saturday	Got errors while integrating in osdag	6
25-May-2025	Sunday	Assigned with new task of adding spacing details for moment splice connections under connection menu	6
26-May-2025	Monday	Discussed with Dhimanth about how to work with spacing details and worked with CAD integration	5.5
27-May-2025	Tuesday	Had exam, managed to work with CAD integration	3
28-May-2025	Wednesday	working with adding spacing details	5.5
29-May-2025	Thursday	referring Aryan's document of spacing details and was working in it	5
30-May-2025	Friday	was travelling, was able to work on CAD integration	4.5
31-May-2025	Saturday	Had meeting with Neela Lakshmi, Aman regarding the CAD model integation, Aman fixed the issue	3.5
1-Jun-2025	Sunday	Optimising the code, which reduced the rendering time of the model	5
2-Jun-2025	Monday	made stiffener plates by creating custom shape(offline)	6
3-Jun-2025	Tuesday	was assingned to work with Icc graphs(5 graphs)(offline)	7
4-Jun-2025	Wednesday	working on graph 1 (horizontal bar graph)(offline)	7.5
5-Jun-2025	Thursday	working on graph 1 (horizontal bar graph)(offline)	6
6-Jun-2025	Friday	working on graph 2 (radial bar graph)(offline)	7
7-Jun-2025	Saturday	working on graph 3 (bubble graph)(offline)	7
8-Jun-2025	Sunday	working on graph 4 (horizontal bar graph 2)(offline)	
9-Jun-2025	Monday	working on graph 5 (pi chart)(offline)	8
10-Jun-2025	Tuesday	worked with extracting CAD model and get in gltf format(offline)	8
11-Jun-2025	Wednesday	tried creating custom model replicating the end plate connection(offline)	8.5
12-Jun-2025	Thursday	worked with extracting CAD model and get in gltf format(offline)	8
13-Jun-2025	Friday	worked with extracting CAD model and get in gltf format(offline)	9
14-Jun-2025	Saturday	was working with spacing details,completed all 4 modules(offline)	8
15-Jun-2025	Sunday		

Bibliography

[1] Siddhartha Ghosh, Danish Ansari, Ajmal Babu Mahasrankintakam, Dharma Teja Nuli, Reshma Konjari, M. Swathi, Subhrajit Dutta, and Parth Karia. 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.