FOSSEE Winter Internship Report

On

Development of Bolt Spacing Diagram Modules for Osdag and Exploration of Node-Based Editors

Submitted by

Dhimanth Kumar Singh

4rd Year B.Tech Student, Department of Computer Science and Engineering

Manipal Institute of Technology

Manipal

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

August 5, 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.

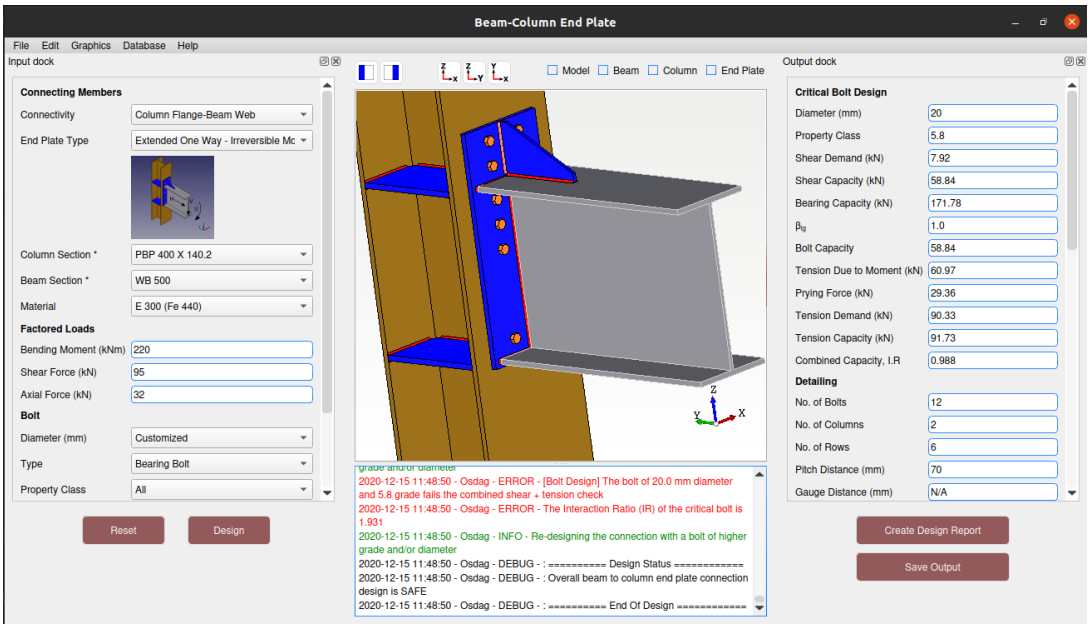

Figure 1.2: Osdag GUI

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 Assignment

2.1 Tasks Done

Task 1: 3D Modeling of a Laced Compound Column (using PythonOCC)

This task focuses on creating a detailed 3D model of a steel laced column using PythonOCC. The model includes:

- **I-Section Creation:**
  Custom I-sections are constructed by combining flanges and a web using `BRepPrimAPI_MakeBox`, and fused together using `BRepAlgoAPI_Fuse`.
  Dimensions: 5500mm (length), 100mm (width), 200mm (depth).

- **Plate Creation:**
  Rectangular end plates (300mm×430mm×10mm) are modeled and positioned at both ends of the column.

- **Diagonal Lace Creation:**
  Diagonal lacing elements are formed by defining 4-point polygonal faces and extruding them by 8mm to create solid braces.

- **Lace Generation:**
  Multiple diagonal laces are automatically generated between the two I-sections at regular intervals using computed polygon coordinates.

- **Assembly and Visualization:**

  The complete assembly (I-sections, plates, laces) is positioned in 3D space and rendered using the PythonOCC viewer. Color coding is used to visually distinguish components.

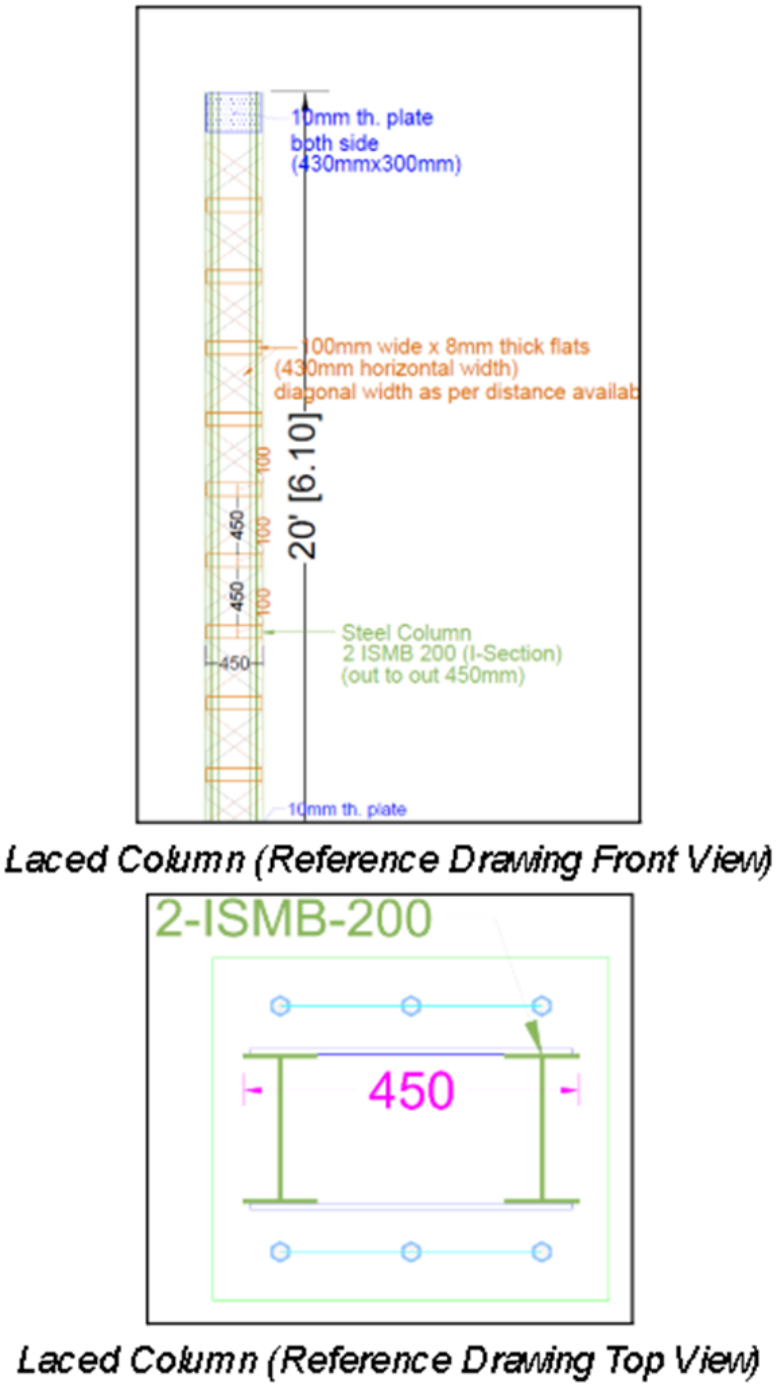

Figure 2.1: 3D Model of Laced Column using PythonOCC

Task 2: Shear Force and Bending Moment Diagram Plotting (using Matplotlib)

This task involves reading structural analysis data and generating visual diagrams:

- **CSV Reading:**

  A CSV file is read to extract values for distance (in meters), shear force (in kN), and bending moment (in kNm).

- **SFD/BMD Plotting:**

  Two subplots are generated using `Matplotlib`:

  - A Shear Force Diagram (SFD) is plotted using a filled step plot.
  - A Bending Moment Diagram (BMD) is plotted using a similar format.
  - Maximum and minimum points on both diagrams are annotated for better interpretation.

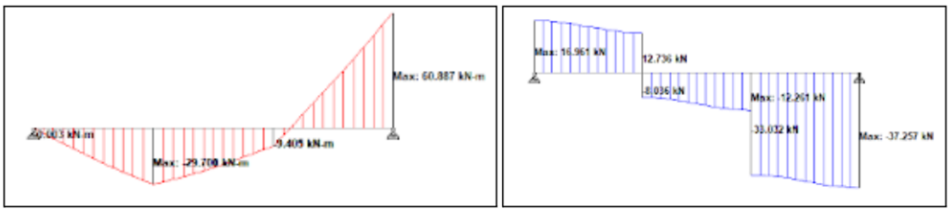

Figure 2.2: Shear Force and Bending Moment Diagrams

Chapter 3

Internship Task 1 Bolt Layout Generator for Steel Connections in Osdag

3.1 Problem Statement

In structural steel design, different types of connections such as beam-to-beam, beam-to-column, and base plates require accurate bolt layouts and spacing to ensure structural integrity and compliance with standards. Manually calculating and drafting bolt arrangements for each connection is time-consuming and error-prone. The objective of this task was to automate the generation and visualization of bolt layouts for various steel connections using Python and PyQt5, making it easier for engineers to verify connection adequacy and prepare fabrication drawings

3.2 Tasks Done

3.2.1 Overview

This task involved the design and development of a visual bolt layout generator for steel structural connections using Python and PyQt5. The goal was to automate the drafting of bolt patterns and dimension annotations for different types of connections such as base plates, end plates, and cover plates, commonly used in steel structures.

3.2.2 Methodology & Process Flow

The following structured process was followed throughout the task:

1. **Requirement Analysis**
    - Studied various steel connection types and their bolt arrangement standards (pitch, edge, gauge, end distances).
    - Identified which parameters affect bolt placement and how dimensioning is typically represented in structural drawings.
2. **Template Design (Reusable GUI Structure)**
    - Created a standard PyQt5-based GUI template with two main components:
        - **Left Panel**: Display of input parameters like plate size and bolt dimensions.
        - **Right Panel**: Dynamic graphical view of the bolt layout using `QGraphicsView`.
    - **Figure 1**: Template UI layout used across all connection types (Insert GUI screenshot here).
3. **Modular Code Implementation**
    - A base template was reused across all scripts; only the bolt layout logic in `createDrawing()` was customized per connection type.
    - Implemented support for:
        - Centering bolts
        - Drawing symmetrical patterns
        - Auto-scaling views based on plate dimensions
4. **Dynamic Parameter Integration**
    - Extracted design values (bolt diameter, edge distance, etc.) from the main connection object.
    - Displayed values in the UI and used them to drive drawing logic.
5. **Dimensioning Implementation**

- Developed two helper methods:
  - `addHorizontalDimension()`
  - `addVerticalDimension()`

  These methods draw arrows, dimension lines, and labels based on bolt and plate geometry.

3.2.3 of File Responsibilities

Script/File Name	Connection Type	Customization Focus
`b2bcoverplate.py`	Beam-to-beam (cover plate)	Top View of Connection
`baseplatedetailing.py`	Column base plate	Top View of Connection
`cleatangledetailing.py`	Angle cleat connections	Top View of Connection
`b2bEPsketch.py`	Beam-to-beam end plate	Side View of Connection
`baseplatehollow.py`	Hollow sections (base)	Top View of Connection
`b2cendplateSketch.py`	Beam-to-column end plate (sketch)	Side View of the Connection
`BC2Cendplate.py`	Beam-column-column end plate	Top View of Connection
`Beam2ColEnddetailing.py`	Beam-to-column end plate detailing	Top View of Connection

3.3 Python Code

This section presents Python scripts developed during the internship for visualizing bolt patterns in structural steel connections. The scripts are built using the PyQt5 framework and were tailored for various types of connections like base plates, cover plates, cleat angles, and end plates. All scripts follow a consistent structure and aim to assist engineers in checking bolt layouts against spacing rules visually and dimensionally.

3.3.1 Common Components Across All Scripts

1. **addHorizontalDimension() and addVerticalDimension()**

   These two helper functions are defined in every script to handle the drawing of dimension lines and labels. They:

   - Draw arrows and extension lines using `QGraphicsLineItem` and `QPolygonF`
   - Add dimension text with proper font size and position
   - Maintain visual clarity regardless of plate scale

   These methods ensure that all bolts and plate edges are clearly annotated for fabrication reference.

2. **initUI()**

   This method is also common across all scripts and performs the following:

   - Sets up the main window layout
   - Creates the left panel to show parameters like bolt diameter, edge distance, pitch, etc.
   - Creates the right panel (`QGraphicsView`) to display the bolt and plate drawing
   - Automatically scales the view for larger plates
   - Calls `createDrawing()` which is the only method that differs in each file

   This method ensures a consistent user interface across all connection types.

3. **__init__() (Constructor Method)**

   In each class (e.g., `B2Bcoverplate`, `baseplatedetailing`), the __init__() method is responsible for:

   - Receiving the connection object and determining whether it's a web, flange, or base connection
   - Extracting values like:
     - Plate dimensions (length/width/height)
     - Bolt diameter

  - Spacing values (pitch, gauge, edge, end)
  - Bolt capacity or arrangement
- Storing these values as object attributes
- Triggering the GUI with `self.initUI()`

The constructor is where the script becomes specialized for a particular connection type.

3.3.2 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.

3.3.3 Python Code

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

1 . b2bcoverplateweld.py – `createDrawing()`

The `createDrawing()` method in this file generates a top view of a welded cover plate used in beam-to-beam connections. It visually represents the plate geometry along with horizontal and vertical welds around its boundary.

Key Steps

- **Plate Drawing:** The base plate is drawn using:

```
rect_item = QGraphicsRectItem(QRectF(0, 0, plate_length,
   plate_width))
self.scene.addItem(rect_item)
```

- **Dimensioning:** Plate dimensions are annotated using:

```
self.addHorizontalDimension(0, -30, self.plate_length, -30, f
    "{self.plate_length} mm", pen)
self.addVerticalDimension(self.plate_length + 30, 0, self.
    plate_length + 30, self.plate_width, f"{self.plate_width}
    mm", pen)
```

- **Weld Visualization:** Welds are represented as red rectangles placed around the plate.

  – Horizontal welds on top and bottom:

```
top_weld = QGraphicsRectItem(QRectF(0, -weld_size,
    plate_length, weld_size))
bottom_weld = QGraphicsRectItem(QRectF(0, plate_width,
    plate_length, weld_size))
```

  – Vertical welds (left and right), split due to weld gap:

```
left_top = QGraphicsRectItem(QRectF(-weld_size, 0,
    weld_size, half_height))
right_bottom = QGraphicsRectItem(QRectF(plate_length,
    plate_width - half_height, weld_size, half_height))
```

This visual representation provides clarity for weld detailing and helps ensure correct fabrication of beam-to-beam cover plate connections.

2 . B2bEPSketch.py – `createDrawing()`

The `createDrawing()` method generates a top view layout of a dual-plate base system with beams and stiffeners on both sides. The drawing includes plate outlines, detailed stiffener arrangements, beam placements, and dimension annotations to represent a two-way structural connection, typically used for hollow sections or parallel beam connections.

Key Functionalities

- **Plate Layout:** Two vertical plates are drawn with appropriate spacing between them. The layout is centered in the view using:

```
self.scene.addRect(start_x, start_y, plate_thickness,
   plate_height, blue_pen)
self.scene.addRect(start_x + plate_thickness, start_y,
   plate_thickness, plate_height, blue_pen)
```

- **Stiffener Representation:** Horizontal and vertical stiffeners are added on both sides of each plate. Cap stiffeners are shown using polygon shapes like:

```
polygon_tl = QGraphicsPolygonItem(QPolygonF(points_tl))
self.scene.addItem(polygon_tl)
```

- **Beam Visualization:** Beams are placed on the left and right of the plate system to simulate a beam-to-plate-to-beam configuration:

```
self.scene.addRect(start_x - beam_width, beam_y, beam_width,
   beam_height, pen)
```

- **Cap Detailing:** Additional triangular or polygonal shapes are used to represent cap stiffeners at corners, giving a detailed visual of end welds or reinforcement areas.

- **Dimensioning:** Horizontal and vertical distances such as plate height, stiffener width, and beam spacing are added using:

```
self.addHorizontalDimension(...)
self.addVerticalDimension(...)
```

This method allows the engineer or fabricator to visually verify the complete geometry of a complex dual-sided plate and stiffener system in a clear, scalable format.

3 . baseplatedetailing.py – `createDrawing()`

The `createDrawing()` method generates a top view of a base plate showing the column section, flange/web geometry, optional stiffeners, and both outer and inner bolt placements. This layout helps in checking space constraints, fitment, and verifying spacing and edge rules for fabrication.

Key Functionalities

- **Base Plate and Column Geometry:** A rectangle is drawn to represent the base plate using:

```
rect_item = QGraphicsRectItem(QRectF(0, 0, plate_length,
    plate_width))
```

The column web is illustrated with two horizontal lines:

```
self.scene.addLine(web_left_x, web_top_y, web_right_x,
    web_top_y, outline_pen)
self.scene.addLine(web_left_x, web_bot_y, web_right_x,
    web_bot_y, outline_pen)
```

Flanges on both sides are drawn with vertical and horizontal lines.

- **Flange Stiffeners:** If `self.stiff_flange_length` is valid, stiffeners are added adjacent to each flange. For example:

```
self.scene.addRect(stiffener_left_top, outline_pen, red_brush
    )
```

- **Web Stiffeners Along and Across:** If `self.stiff_along_thickness` is specified:

```
self.scene.addRect(stiffener_web_left, outline_pen, red_brush
    )
self.scene.addRect(stiffener_web_right, outline_pen,
    red_brush)
```

And for across stiffeners:

```
self.scene.addRect(stiffener_web_top, outline_pen, red_brush)
self.scene.addRect(stiffener_web_bot, outline_pen, red_brush)
```

- **Outer Bolt Placement (Left and Right Edges):** The number of bolts is dynamically computed. Bolt holes are drawn using:

```
self.scene.addEllipse(x - radius, y_top - radius, 2 * radius,
    2 * radius, outline_pen)
```

- **Inner Bolt Placement (Around Web):** For example, if 4 bolts are required:

```
self.scene.addEllipse(x1 - radius, y1 - radius, 2 * radius, 2
    * radius, outline_pen)
```

- **Dynamic Edge Adjustments:** Edge distance is recalculated based on bolt count using:

$$\text{edge} = \frac{\text{plate\_length} - \text{column\_len} - 2 \cdot \text{flange\_thickness}}{4}$$

- **Final Dimensioning:** All relevant dimensions are annotated using:

```
self.addDimensions(black_pen)
```

This method provides a complete visual of base plate geometry including bolt spacing, column profiles, and optional stiffeners.

4 . baseplatehollow.py – `createDrawing()`

The `createDrawing()` method produces a top view layout of a base plate with a hollow column section (either circular or rectangular), vertical and horizontal stiffeners, and bolt placement at four corners. It dynamically adapts the drawing based on whether the section is RHS/SHS or a circular hollow section.

Key Functionalities

- **Base Plate and Column Section:**

  A base plate is drawn as a white rectangle. The column section is checked using `self.column_section.startswith('RHS')` or `'SHS'`.

  - For rectangular hollow sections, two nested rectangles are drawn to represent the column wall thickness.
  - For circular sections, nested ellipses are used to represent the outer and inner hollow boundaries:

```
self.scene.addEllipse(top_left_x, top_left_y, col_len,
    col_width, outline_pen)
```

- **Stiffener Drawing:**

  Stiffeners are placed as follows:

  – Vertically above and below the column along the plate edges:

  ```
  self.scene.addRect(stiff_x, stiff_top, stiff_thickness,
      stiff_height, outline_pen, QBrush(Qt.red))
  ```

  – Horizontally to the left and right of the column:

  ```
  self.scene.addRect(stiff_x, stiff_start_y, stiff_length,
      stiff_height, outline_pen, QBrush(Qt.red))
  ```

  The type of stiffener (B-type, D-type, or OD) is selected based on the section type.

- **Bolt Placement:**

  Four bolts are placed at the corners of the base plate using:

  ```
  self.scene.addEllipse(x - radius, y - radius, hole_dia,
      hole_dia, outline_pen)
  ```

  Dimensions from the plate edge (edge, end) are extracted from the input parameters.

- **Dimensioning:**

  The overall plate size is annotated using:

  ```
  self.addHorizontalDimension(0, -30, self.plate_length, -30, f
      "{self.plate_length} mm", pen)
  self.addVerticalDimension(self.plate_length+30, 0, self.
      plate_length+30, self.plate_width, f"{self.plate_width} mm
      ", pen)
  ```

This method adapts well to both rectangular and circular hollow columns and shows all key fabrication information, including bolt positioning and stiffener detailing.

5 . Beam2ColEnddetailing.py – `createDrawing()`

The `createDrawing()` method produces a top view or side view layout (based on a flag) of a beam-to-column end plate connection. The drawing includes flanges, web lines, stiffeners, and bolt placement. It dynamically adapts to either full plate detailing (`flag == 0`) or flange-only detailing (`flag != 0`).

Key Functionalities

- **View Toggle:**

  The function uses `self.flag` to switch between full plate view and flange-only detailing:

  - If `flag == 0`, a full top view of the plate is drawn.
  - If `flag != 0`, only the flange portion is visualized.

- **If `flag == 0` (Full Plate View):**

  - **Base Plate Geometry:**

    The entire end plate is drawn using:

    ```
    rect_item = QGraphicsRectItem(QRectF(0, 0,
       webdetailinglen, webdetailingwidth))
    ```

  - **Web and Flange Representation:**

    Orange horizontal and vertical lines depict flange and web profiles using:

    ```
    self.scene.addLine(...)
    ```

  - **Bolt Placement (Web):**

    Bolts are placed on both sides of the web, with columns calculated as:

    $$x_{\text{left}} = \text{center\_x} - \frac{\text{web\_thick}}{2} - \text{web\_end}$$

    $$x_{\text{right}} = \text{center\_x} + \frac{\text{web\_thick}}{2} + \text{web\_end}$$

    Rows use varying pitches like `pitch1`, `pitch2`, etc.

- **Dimensioning:**

  Horizontal and vertical dimensions for plate size, web-to-bolt distance, and bolt pitch are added using:

```
self.addHorizontalDimension(...)
self.addVerticalDimension(...)
```

- **If `flag != 0` (Flange-Only View):**

  - **Flange Plate and Stiffener Drawing:**

    A half-height flange plate is drawn along with a central stiffener using:

```
self.scene.addRect(center_x - stiff_thick / 2, 0,
   stiff_thick, stiff_len)
```

  - **Flange Web Boundary Lines:**

    Orange lines mark the flange top and bottom and web boundaries.

  - **Bolt Placement (Flange):**

    * Two bolts near the stiffener are placed at a horizontal offset of `flangeend`.
    * Additional web bolts are arranged vertically using the pitch cycle.

  - **Flexible Layout:**

    Bolt coordinates and counts adjust dynamically based on parameters like `self.web_bolts`, `pitch1--4`, and `flangeend`.

-

  This method provides a detailed layout for both full end plate and flange-only configurations in beam-to-column connections. It accurately shows bolt pitch, alignment, and stiffener geometry for fabrication and design validation.

6 . beam2beamcoverplatedetailing.py – `createDrawing()`

The `createDrawing()` method generates a top view layout of a beam-to-beam cover plate connection with customizable bolt rows and columns. It adapts to different combinations of odd and even bolt counts to create a balanced, symmetric pattern.

Key Functionalities

- **Base Plate Drawing:**

  A rectangle is drawn to represent the cover plate using `QGraphicsRectItem`. The plate dimensions are annotated using:

```
self.addHorizontalDimension(0, -30, self.plate_length, -30, f
    "{self.plate_length} mm", pen)
self.addVerticalDimension(self.plate_length+30, 0, self.
    plate_length+30, self.plate_width, f"{self.plate_width} mm
    ", pen)
```

- **Bolt Layout (Symmetrical):**

  Bolts are arranged based on the following input parameters: `rows`, `cols`, `pitch`, `gauge`, `edge`, `end`, and `bolt_diameter`.

  - For **odd rows**, bolts are drawn symmetrically across a central horizontal axis.
  - For **odd columns**, bolts are drawn symmetrically across a central vertical axis.
  - The method handles all combinations and ensures mirrored bolt positioning using nested loops.

  Example logic for odd rows:

```
x_center = self.plate_length / 2
y_center = end + row * pitch
y_center = self.plate_width - end - row * pitch
```

- **Bolt Drawing:**

  Each bolt is drawn using:

```
self.scene.addEllipse(x_center - radius, y_center - radius,
    hole_dia, hole_dia, outline_pen)
```

- **Bolt and Hole Dimensioning:**

  Edge distance, end distance, and hole diameter are annotated below or beside each bolt using:

```
self.addHorizontalDimension(...)
self.addVerticalDimension(...)
```

This method provides a flexible layout that adapts to any number of rows and columns, making it suitable for various bolt configurations in cover plate connections.

7 . cleatangledetailing.py – `createDrawing()`

The `createDrawing()` method generates a top view of a cleat angle bolt layout. It accounts for varying gauges, end and edge distances, and dynamically calculates bolt positions based on the number of rows and columns. The layout is typically asymmetric, used where cleats connect secondary structural members.

Key Functionalities

- **Parameter Handling:**
  The method extracts parameters like `pitch`, `end`, `edge`, `hole_diameter`, `rows`, and `cols` from the input dictionary `params`. It supports both:
  - a single `gauge` value, and
  - alternating values `gauge1` and `gauge2`.
- **Plate and Scene Setup:**
  The scene rectangle is defined with extra space for dimensioning using:

```
self.scene.setSceneRect(-h_offset, -v_offset, width + 2*
   v_offset, height + 2*h_offset)
self.scene.addRect(0, 0, width, height, dimension_pen)
```

- **Bolt Placement (Right to Left):**
  For each bolt in the grid, the X-position starts from the right edge and subtracts gauges in alternating sequence to support unequal spacing:
  - For even columns: subtract `gauge1`
  - For odd columns: subtract `gauge2`

The bolts are added using:

```
self.scene.addEllipse(x, y, hole_diameter, hole_diameter,
    outline_pen)
```

- **Dimensioning:**

  After plotting all bolt holes, the function calls:

```
self.addDimensions(params, dimension_pen)
```

  which draws standard pitch, edge, end, and gauge annotations on the plate layout.

This drawing function allows for accurate representation of cleat angles with irregular or asymmetric bolt arrangements and ensures all relevant dimensioning is visualized for fabrication.

3.4 Documentation

3.4.1 Directory Structure

The Osdag application follows the directory structure below:

- `osdagMainPage.py` – Main entry point of the application
- `Common.py` – Contains shared utility functions
- `ResourceFiles/` – Stores shared assets such as:
  - `images/` – For GUI illustrations
  - `last_designs/` – For storing recently used designs
- `design_type/` – Contains module-specific folders
- `design_report/` – Generates and saves report PDFs
- `cad/` – For exporting CAD views
- `gui/` – Contains GUI templates and all detailing scripts:
  - `ui_template.py` – The common UI base used by all detailing views

- `beam2beamcoverplatedetailing.py`
- `b2bcoverplateweld.py`
- `b2bEPsketch.py`
- `b2cendplateSketch.py`
- `baseplatedetailing.py`
- `baseplatedetailinghollow.py`
- `BC2Cendplate.py`
- `Beam2ColEnddetailing.py`
- `cleatangledetailing.py`
- `endplatecnndetailing.py`

3.4.2 Program Start

To start the Osdag application:

1. Open a terminal and navigate to the `Osdag` project root directory.
2. Run the following command:

```
python osdagMainPage.py
```

3. This launches the Osdag main interface, from which various structural design modules can be accessed.

3.4.3 Using the GUI Detailing Modules

Each structural design module includes a **"Detailing"** button within its GUI.

To generate a spacing diagram for bolts or plates:

1. Enter input parameters for the structural model (e.g., plate size, bolt pitch, edge distance).
2. Click the **Detailing** button.
3. The appropriate script (e.g., `baseplatedetailing.py` or `b2bcoverplateweld.py`) is triggered.

4. The corresponding `createDrawing()` function uses the `QGraphicsScene` to:

   - Draw the base plate using:

```
self.scene.addRect(0, 0, plate_length, plate_width, pen)
```

   - Position bolts using:

```
self.scene.addEllipse(x - r, y - r, 2 * r, 2 * r,
   outline_pen)
```

   - Add dimension lines using:

```
self.addHorizontalDimension(...)
self.addVerticalDimension(...)
```

The final output is a scaled, annotated diagram showing the bolt layout and dimensions, used for visual verification and documentation.

Chapter 4

Internship Task 2: Visual Node Editor Integration

4.1 4.1 Task 2: Problem Statement

Osdag currently uses a static CAD-style view for visualizing structural connections. To enhance interactivity, flexibility, and modular control, we explored replacing the fixed layout system with a visual node-based editor. This system would allow users to drag, connect, and configure components like plates, bolts, and stiffeners in a node graph, similar to popular visual programming tools.

The goal was to evaluate tools that can integrate smoothly into Osdag's `PyQt`-based environment while offering dynamic visual workflows.

4.2 4.2 Task 2: Tasks Done

I explored visual node editors for integration and comparison, specifically `NodeGraphQt` and `PyFlow`. Here is a summary of my findings:

Video Documentation

There are no official tutorial videos available for either `NodeGraphQt` or `PyFlow`. Therefore, I manually tested both libraries to assess their usability, documentation quality, and integration capabilities.

Additionally, I created a short demo video (without audio) to showcase how the `NodeGraphQt` editor looks and functions.

Environment Compatibility

- **PyFlow** is **not compatible** with the `osdag-editable` environment due to conflicting dependencies, especially with `PyQt`/`PySide` versions.
- **NodeGraphQt** works **smoothly** within Osdag and supports `PyQt5`.

Setup for NodeGraphQt

To install and test `NodeGraphQt`, follow the steps below:

1. Clone the repository:

```
git clone https://github.com/jchanvfx/NodeGraphQt.git
```

   (Optionally place it inside your Osdag project folder for local reference.)

2. Install the package:

```
pip install NodeGraphQt
```

Creating and Connecting Nodes via Code

`NodeGraphQt` provides full programmatic control over the node graph. Nodes can be created dynamically and connected using the following commands:

- `graph.create_node(...)`
- `node.set_input(...)`
- `node.set_output(...)`
- `port.connect_to(...)`

This enables flexible, data-driven workflows that can be embedded into custom GUIs. It is particularly useful for modular design applications like Osdag where multiple design objects (plates, bolts, welds) need to be visually and functionally linked.

4.3 4.3 Task 2: Documentation

NodeGraphQt Sample Nodes and Code Snippets

This section documents various node types explored and tested within `NodeGraphQt`. Each node is initialized using a unique creation function and includes configuration options such as custom names, colors, and icons.

BasicNodeA

Code to create node:

```
n_basic_a = graph.create_node('nodes.basic.BasicNodeA',
    text_color='#feab20')
n_basic_a.set_disabled(True)
```

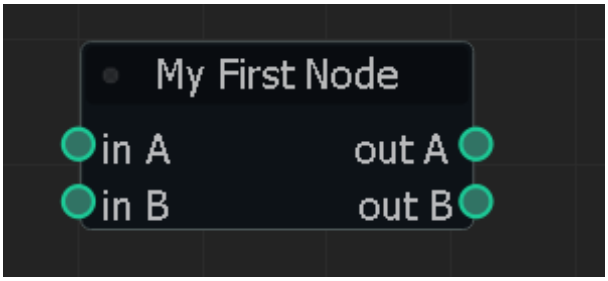

Figure 4.1: BasicNodeA Example

BasicNodeB (with Custom Icon)

Code to create node:

```
n_basic_b = graph.create_node('nodes.basic.BasicNodeB', name='
    custom icon')
n_basic_b.set_icon(Path(BASE_PATH, 'star.png'))
```

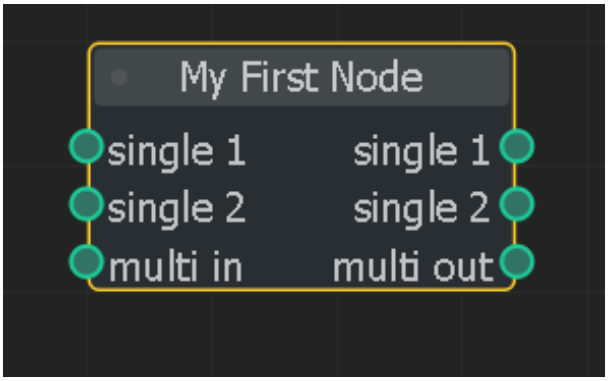

Figure 4.2: BasicNodeB with Icon

CustomPortsNode

Code to create node:

```
n_custom_ports = graph.create_node('nodes.custom.ports.
    CustomPortsNode', name='custom ports')
```

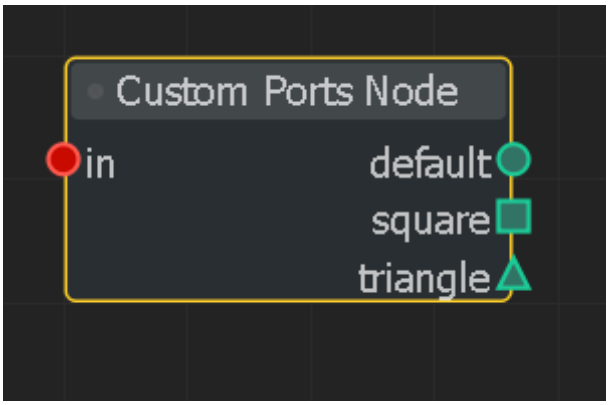

Figure 4.3: CustomPortsNode Example

TextInputNode

Code to create node:

```
n_text_input = graph.create_node('nodes.widget.TextInputNode',
    name='text node', color='#0a1e20')
```

Figure 4.4: TextInputNode Example

CheckboxNode

Code to create node:

```
n_checkbox = graph.create_node('nodes.widget.CheckboxNode', name=
    'checkbox node')
```

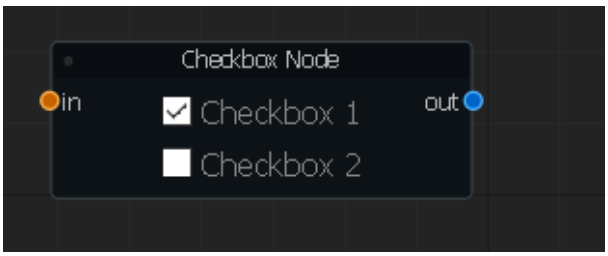

Figure 4.5: CheckboxNode Example

DropdownMenuNode

Code to create node:

```
n_combo_menu = graph.create_node('nodes.widget.DropdownMenuNode',
    name='combobox node')
```

Figure 4.6: DropdownMenuNode Example

CircleNode

Code to create node:

```
n_circle = graph.create_node('nodes.basic.CircleNode', name='
    circle node')
```

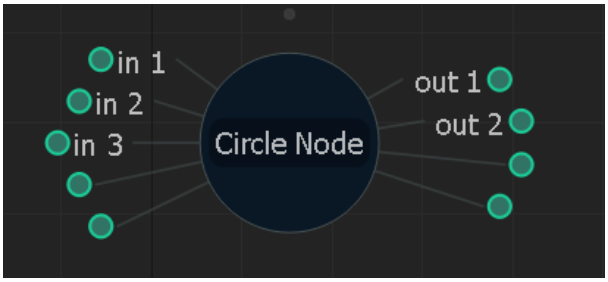

Figure 4.7: CircleNode Example

GroupNode

Code to create node:

```
n_group = graph.create_node('nodes.group.MyGroupNode')
```

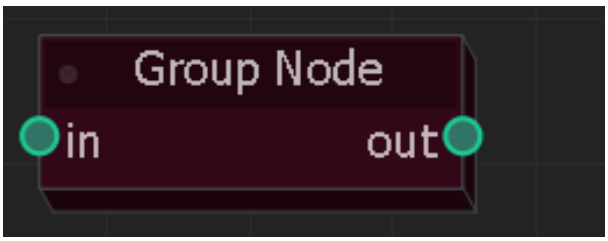

Figure 4.8: GroupNode Example

Backdrop Node

Code to create node:

```
n_backdrop = graph.create_node('Backdrop')
n_backdrop.wrap_nodes([n_custom_ports, n_combo_menu])
```

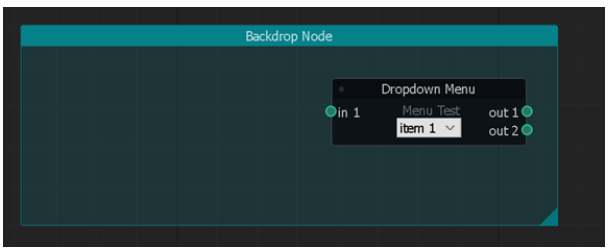

Figure 4.9: Backdrop Node Example

Connecting Nodes in NodeGraphQt

Step 1: Create Two Nodes

```
n_text_input = graph.create_node('nodes.widget.TextInputNode',
    name='Text Input')
n_checkbox = graph.create_node('nodes.widget.CheckboxNode', name=
    'Checkbox')
```

Step 2: Connect Output Port to Input Port

- **Method 1: Using `set_output`**

```
n_text_input.set_output(0, n_checkbox.input(0))
```

- **Method 2: Using `set_input`**

```
n_checkbox.set_input(0, n_text_input.output(0))
```

- **Method 3: Using `connect_to()` from the port object**

```
out_port = n_text_input.output(0)
in_port = n_checkbox.input(0)
out_port.connect_to(in_port)
```

Chapter 5

Conclusions

5.1 5.1 Tasks Accomplished

During the course of the internship, I completed two primary tasks:

Task 1: Bolt Layout Visualization GUI Modules

Developed multiple Python scripts using PyQt5 and QGraphicsScene to visualize bolt layouts for various steel connection types. Each script allows users to input parameters like plate dimensions, bolt count, edge/pitch/gauge distances, and generates annotated spacing diagrams. These models include base plates, beam-to-beam and beam-to-column connections, cleat angles, and end plates. All modules are integrated with Osdag's UI structure and follow a unified design template.

Task 2: Exploration of Interactive Node-Based UIs

Evaluated visual node editors to modernize and modularize Osdag's interface. Specifically explored NodeGraphQt and PyFlow for suitability:

- Created a silent demo video for NodeGraphQt
- Compared setup complexity and environment compatibility
- Successfully installed and tested NodeGraphQt in `osdag-editable`, enabling dynamic creation and connection of nodes through Python code
- Documented PyFlow incompatibility due to PyQt conflicts

This dual-track effort contributed to both functional bolt design tools and forward-looking UI possibilities for future development in Osdag.

5.2 5.2 Skills Developed

Over the internship period, I developed the following skills:

Technical Skills

- Proficient in PyQt5 GUI development, including widget layout, graphics rendering, and event-driven interactions
- Experience with QGraphicsScene, QGraphicsItem, and dynamic drawing methods like `addEllipse`, `addRect`, and custom dimension lines
- Learned to integrate Python-based visual editors like NodeGraphQt into an existing software environment
- Familiarity with handling parametric inputs, calculating geometry, and generating real-time drawings

Software & Tooling

- Used tools like Git, VS Code
- Worked within Osdag's modular project structure and contributed GUI code in a structured, reusable format

Professional Skills

- Collaborated with mentors to align work with project goals
- Maintained consistent documentation and project reporting
- Explored, compared, and tested third-party libraries independently

This internship significantly improved my confidence in GUI-based application development and laid the foundation for contributing to open-source structural engineering tools.

Appendix A: Internship Work Report

Name: Dhimanth Kumar Singh

Project: Osdag

Internship: Semester Long Intern

Date	**Tasks Done**	**Remarks**
19 May 2025	Getting familiar with Osdag codebase and structure	Setup environment, explored repo
20 May 2025	Explored UI architecture, QGraphicsScene basics	Studied existing GUI logic
21 May 2025	Understood detailing module structure	Focused on `ui_template.py`
22–23 May 2025	Developed `baseplatedetailing.py`	Added bolt drawing & dimensions
24–25 May 2025	Completed `baseplatedetailinghollow.py`	Hollow section handling
26–27 May 2025	Built `beam2beamcoverplatedetailing.py`	Symmetric bolt logic
28–29 May 2025	Worked on `b2bcoverplateweld.py`	Weld detailing + red brush visuals

Date	Tasks Done	Remarks
30–31 May 2025	Developed `cleatangledetailing.py`	Unequal gauge pattern support
1–2 June 2025	Finished `b2bEPsketch.py`	End plate logic for beam-beam
3–4 June 2025	Created `b2cendplateSketch.py`	Basic layout for beam-column
5–6 June 2025	Implemented `BC2Cendplate.py`	Compact detailing, ellipse logic
7–8 June 2025	Finalized `Beam2ColEnddetailing.py`	Flag-based dual view logic
9–10 June 2025	Added `endplatecnndetailing.py`	Special end plate with pitch calc
11–13 June 2025	Unified UI behavior across all detailing files	Refactored to match `ui_template`
14–17 June 2025	Testing & visual tweaks across all models	Verified bolt positioning accuracy
18 June 2025	Started research into node-based UI systems	Planning UI upgrade ideas
19–21 June 2025	Explored NodeGraphQt: installation and testing	Confirmed it works with Osdag
22–24 June 2025	Created demo video of NodeGraphQt usage	Silent clip for documentation
25–27 June 2025	Tested PyFlow and noted compatibility issues	PyQt/PySide conflict observed
28–30 June 2025	Drafted documentation for node editor integration	Added to Section 4 of report

Bibliography

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

[2] FOSSEE Project. FOSSEE News - January 2018, vol 1 issue 3. Accessed: 2025-07-01.

[3] FOSSEE Project. Osdag website. Accessed: 2025-07-01.