FOSSEE Semester Long Internship Report

On

Development of Asymmetric Weld Distribution & CAD Generation in OSDAG

Submitted by

Omkar Arvind Kottawar

School of Computer Science and Engineering
VIT Bhopal University
Sehore, Madhya Pradesh, India

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

May 2026

Abstract

This report presents the work carried out during the internship on the topic of **Development of Asymmetric Weld Distribution & CAD Generation in OSDAG**. The project focused on improving the weld design and CAD generation workflow for welded tension and compression members connected to end gusset plates.

The primary objective was to implement centroid-based asymmetric weld distribution logic and ensure that the generated CAD geometry accurately represented the underlying weld demand calculations. Existing implementations used symmetric flange weld geometry despite centroid-based weld calculations already being present in the design layer. The workflow was traced from weld calculation to final CAD generation, and the dependent functions were updated to preserve asymmetric weld behavior throughout the process.

Key contributions include implementing separate heel-side and toe-side flange weld calculations, supporting independent weld objects in the CAD workflow, and updating weld placement logic for both tension and compression members. The updated implementation preserves existing design checks while enabling accurate asymmetric weld representation across supported section profiles.

This report documents the problem statements, workflow analysis, implementation methodology, Python code changes, and final CAD outputs developed during the course of the project.

Keywords: *structural steel design, welded connections, CAD generation, asymmetric weld distribution, centroid-based weld logic*

Acknowledgments

I would like to express my sincere gratitude to everyone who supported and guided me throughout the course of this project. This experience has been both enriching and enlightening, and it would not have been possible without the collective efforts and encouragement of many individuals and institutions.

I am deeply thankful to **Ajmal babu M S, Parth Karia, Ajinkya Dahale** for their constant support, mentorship, and technical guidance throughout the project.

I extend my heartfelt thanks to **Prof. Siddhartha Ghosh**, Department of Civil Engineering, Indian Institute of Technology Bombay, for their vision and leadership, which served as the foundation of this work.

I am also grateful to *FOSSEE PI Prof. Kannan M. Moudgalya, FOSSEE Project Investigator, Department of Chemical Engineering, IIT Bombay* for providing me with the opportunity to contribute to this impactful initiative.

Special thanks to *Parth Karia* and the entire *OSDAG* for their support and coordination during my project tenure.

I would also like to acknowledge the support of *National Mission on Education through Information and Communication Technology (ICT), Ministry of Education (MoE), Government of India*, whose initiative and resources made this project possible.

Finally, I sincerely thank my institute, **VIT Bhopal University**, and the **School of Computer Science and Engineering** for their academic support and for providing me with the foundation and encouragement to pursue this opportunity.

Omkar Arvind Kottawar
VIT Bhopal University
May 2026

Contents

List of Figures

List of Tables

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.

Figure 1.1. FOSSEE Projects and Activities

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.

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.

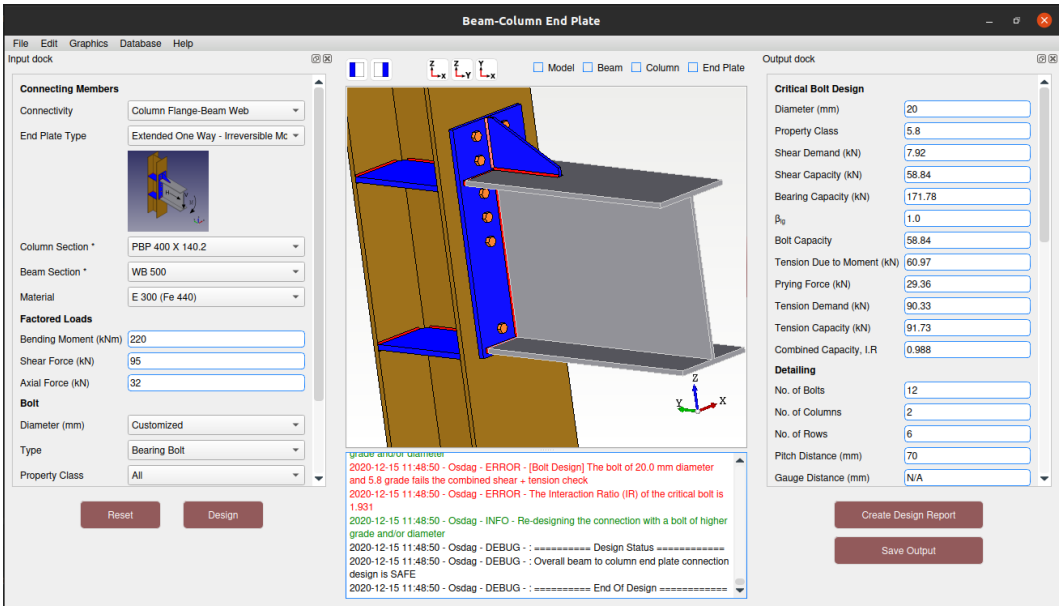

Figure 1.2. Osdag GUI

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

Parametric 3D CAD Model of Steel Girder Bridge

2.1 Problem Statement

Generating accurate three-dimensional CAD models of steel girder bridges typically requires manual geometry creation inside dedicated CAD software. This process is time-consuming and produces static models that must be recreated from scratch whenever a design parameter changes. The absence of a programmatic, parametric approach makes it difficult to explore design variations, test different span configurations, or integrate bridge geometry into automated workflows.

The objective of this project was to implement a fully parametric, modular 3D CAD model of a short-span steel girder bridge using the pythonOCC library in Python. All bridge dimensions and component counts were to be controlled through a set of top-level configuration variables, allowing the entire model to be regenerated by changing a single parameter without modifying the underlying geometry code.

The implementation needed to ensure that:

- **Parametric control:** All dimensions, counts, and layout values were driven by adjustable variables defined at the top of the script.
- **Modular architecture:** Each bridge component was encapsulated in its own class, with shared functionality provided through a common base class.
- **Accurate geometry:** I-section girders, the concrete deck slab, and parapets were constructed using solid geometry operations from the OpenCASCADE kernel.
- **Skew support:** An optional skew angle of 0–15 degrees could be applied to the bridge layout without restructuring the model.
- **Export capability:** The final assembled model could be exported to industry-standard STEP and native BREP file formats for use in downstream CAD tools.

2.2 Existing Workflow Analysis

Prior to this project, bridge geometry generation within the associated codebase was limited to isolated factory functions that produced individual geometric primitives such as I-sections and rectangular prisms. These functions operated independently and had no mechanism for coordinating multiple components into a complete assembly.

The existing approach had the following structural gaps:

1. **No assembly layer:** The factory functions in `draw_i_section.py` and `draw_rectangular_pri` created standalone shapes but provided no class or method to collect, position, and combine multiple shapes into a single compound model.
2. **No parametric layout logic:** Derived quantities such as deck width, girder Y-positions, and elevation levels had to be computed manually for each new bridge configuration. There was no centralised function to calculate these values from a given set of input parameters.
3. **No component abstraction:** Each geometry type was handled by its own standalone factory call with no shared interface. Adding a new component type required writing entirely separate code with no reuse of transformation or state management logic.
4. **No visualisation or export pipeline:** The existing scripts had no integrated mechanism to display an assembled model in an interactive viewer or to export it to STEP or BREP format. These operations would need to be implemented from scratch for each use case.

Because of these limitations, the project required designing a complete object-oriented assembly framework on top of the existing factory functions, while retaining those functions as the low-level geometry primitives for shape creation.

2.3 Implementation

2.3.1 Architecture Overview

The project was structured around a two-tier class hierarchy. A base class, `BridgeComponent`, provided common shape management and geometric transformation methods. Three component subclasses — `Girder`, `Deck`, and `Parapet` — inherited from this base class and each implemented their own `create_geometry()` method. A separate orchestrator class, `BridgeModel`, was responsible for computing the bridge layout, instantiating all components, positioning them in 3D space, assembling them into a compound shape, and handling visualisation and export.

2.3.2 Key Implementation Features

- **Top-level parameter block:** All geometry-controlling variables (`SPAN_LENGTH_L`, `N_GIRDERS`, `GIRDER_SECTION_D`, etc.) were defined at the top of `bridge_model.py` so that the entire model could be reconfigured without touching the class or function bodies.
- **Shared base class:** `BridgeComponent` provided `set_shape()`, `get_shape()`, `translate()`, and `rotate()` methods, eliminating duplicated transformation boilerplate across component subclasses.
- **I-section construction:** Girder geometry was built by fusing three `BRepPrimAPI_MakeBox` solids (bottom flange, top flange, web) using `BRepAlgoAPI_Fuse`, then translating flanges and the web to their correct elevations.
- **Layout computation:** `compute_layout()` derived deck width, girder Y-positions, and deck elevation from the input parameters, removing the need for manual geometric calculations when parameters change.
- **Compound assembly:** `assemble()` used `BRep_Builder` and `TopoDS_Compound` to combine all component shapes into a single topological compound without merging their individual topologies.
- **Colour-coded visualisation:** The `display()` method rendered girders in steel gray, the deck in concrete gray, and parapets in light beige using `Quantity_Color` with RGB values.
- **Dual-format export:** `export()` wrote the assembled compound to STEP format using `write_step_file()` and to BREP format using `breptools.Write()`, both controlled by top-level boolean flags.
- **Skew angle support:** A non-zero `SKEW_ANGLE` parameter could be applied via the `rotate()` method of `BridgeComponent` to produce skewed bridge geometries without modifying the assembly logic.

2.4 Python Code

This section presents the key classes and functions implemented in `bridge_model.py`. Each snippet is accompanied by an explanation of its role in the parametric assembly workflow.

2.4.1 File: `bridge_model.py`

Parameter Block

```
1  UNITS = "mm"
2  SPAN_LENGTH_L = 12000
3  N_GIRDERS = 3
4  GIRDER_CENTROID_SPACING = 3000
5  GIRDER_OFFSET_FROM_EDGE = 500
6  SKEW_ANGLE = 10
7
8  GIRDER_SECTION_D = 900
9  GIRDER_SECTION_BF = 300
10 GIRDER_SECTION_TF = 16
11 GIRDER_SECTION_TW = 10
12
13 DECK_THICKNESS = 200
14 PARAPET_WIDTH = 300
15 PARAPET_HEIGHT = 1000
16
17 SAVE_STEP = True
18 STEP_FILENAME = "bridge_model.step"
19 SAVE_BREP = True
20 BREP_FILENAME = "bridge_model.brep"
```

Listing 2.1. Top-level configuration parameters in `bridge_model.py`

- **Lines 1–6:** Global layout parameters control the bridge span, number of girders, girder spacing, edge overhang, and optional skew angle. All values are in millimetres.
- **Lines 8–11:** I-section geometry parameters define the depth, flange width, flange thickness, and web thickness of each steel girder.
- **Lines 13–15:** Deck and parapet dimensions are specified independently, allowing the deck thickness and parapet profile to be adjusted without affecting girder geometry.
- **Lines 17–20:** Boolean flags and filename strings control whether STEP and BREP export files are written after model assembly.

I-Section Factory Function

```
1  def create_i_section(d, bf, tf, tw, length):
2      web_height = d - 2 * tf
3
4      bottom_flange = BRepPrimAPI_MakeBox(length, bf, tf).Shape()
5
6      top_flange = BRepPrimAPI_MakeBox(length, bf, tf).Shape()
7      trsf = gp_Trsf()
```

```
8       trsf.SetTranslation(gp_Vec(0, 0, d - tf))
9       top_flange = BRepBuilderAPI_Transform(top_flange, trsf,
            True).Shape()
10
11      web = BRepPrimAPI_MakeBox(length, tw, web_height).Shape()
12      trsf = gp_Trsf()
13      trsf.SetTranslation(gp_Vec(0, (bf - tw) / 2, tf))
14      web = BRepBuilderAPI_Transform(web, trsf, True).Shape()
15
16      shape = BRepAlgoAPI_Fuse(bottom_flange, top_flange).Shape()
17      shape = BRepAlgoAPI_Fuse(shape, web).Shape()
18      return shape
```

Listing 2.2. Custom I-section geometry factory in `bridge_model.py`

- **Line 2:** The web height is derived by subtracting both flange thicknesses from the total section depth.
- **Lines 4–8:** The bottom and top flanges are created as rectangular box solids. The top flange is translated upward by `(d - tf)` to sit at the correct elevation.
- **Lines 10–13:** The web solid is created and translated so that it is horizontally centred between the flanges and sits directly above the bottom flange.
- **Lines 15–17:** The three solids are fused together using `BRepAlgoAPI_Fuse` to produce a single connected I-section solid, which is returned for use by the `Girder` class.

Class: **`BridgeComponent`**

```
1   class BridgeComponent:
2       def __init__(self):
3           self.shape = None
4
5       def set_shape(self, s):
6           self.shape = s
7
8       def get_shape(self):
9           return self.shape
10
11      def translate(self, dx, dy, dz):
12          trsf = gp_Trsf()
13          trsf.SetTranslation(gp_Vec(dx, dy, dz))
14          self.shape = BRepBuilderAPI_Transform(
```

```
            self.shape, trsf, True
        ).Shape()

    def rotate(self, axis_point, axis_direction, angle_deg):
        axis = gp_Ax1(axis_point, axis_direction)
        trsf = gp_Trsf()
        trsf.SetRotation(axis, math.radians(angle_deg))
        self.shape = BRepBuilderAPI_Transform(
            self.shape, trsf, True
        ).Shape()
```

Listing 2.3. `BridgeComponent` base class with transformation methods

- **Lines 2–3:** The constructor initialises `self.shape` to `None`. Concrete subclasses call `set_shape()` after creating their geometry.
- **Lines 5–9:** `set_shape()` and `get_shape()` provide a uniform interface for storing and retrieving the OpenCASCADE solid associated with each component.
- **Lines 11–16:** `translate()` constructs a `gp_Trsf` translation and applies it to `self.shape` using `BRepBuilderAPI_Transform`, replacing the component's shape in place.
- **Lines 18–24:** `rotate()` constructs a rotation transform around a user-supplied axis and angle (converted from degrees to radians), enabling skew and orientation adjustments without subclass-specific code.

Component Subclasses

```
class Girder(BridgeComponent):
    def __init__(self, d, bf, tf, tw, length):
        super().__init__()
        self.d, self.bf = d, bf
        self.tf, self.tw, self.length = tf, tw, length

    def create_geometry(self):
        self.set_shape(
            create_i_section(
                self.d, self.bf, self.tf, self.tw, self.length
            )
        )

class Deck(BridgeComponent):
    def __init__(self, width, thickness, length):
```

```
16          super().__init__()
17          self.width, self.thickness, self.length = width,
                thickness, length
18
19      def create_geometry(self):
20          self.set_shape(
21              create_rectangular_prism(self.width, self.thickness,
                    self.length)
22          )
23
24  class Parapet(BridgeComponent):
25      def __init__(self, width, height, length):
26          super().__init__()
27          self.width, self.height, self.length = width, height,
                length
28
29      def create_geometry(self):
30          self.set_shape(
31              create_rectangular_prism(self.width, self.height,
                    self.length)
32          )
```

Listing 2.4. Component subclasses `Girder`, `Deck`, and `Parapet`

- **Lines 1–11:** `Girder` stores the full I-section dimensions and delegates geometry creation to `create_i_section()`, which constructs and fuses the three solid primitives.
- **Lines 13–21:** `Deck` represents the concrete slab as a single rectangular prism whose width spans the full deck width computed by `compute_layout()`.
- **Lines 23–31:** `Parapet` uses the same rectangular prism factory as the deck, parameterised with its own width and height. Two parapet instances are created and positioned at the left and right deck edges.

Class: **`BridgeModel`** — Layout and Assembly

```
1  def compute_layout(self):
2      self.deck_width = (
3          self.spacing * (self.n_girders - 1) + 2 * self.overhang
4      )
5      start_y = -self.spacing * (self.n_girders - 1) / 2
6      self.girder_positions_y = [
7          start_y + i * self.spacing for i in range(self.n_girders)
```

```
    ]
    self.deck_z_level = self.girder_section_d

def create_components(self):
    for y in self.girder_positions_y:
        g = Girder(self.girder_section_d, self.girder_section_bf,
                   self.girder_section_tf, self.girder_section_tw,
                   self.span_length)
        g.create_geometry()
        self.girders.append(g)
    self.deck = Deck(self.deck_width, self.deck_thickness,
       self.span_length)
    self.deck.create_geometry()
    for _ in range(2):
        p = Parapet(PARAPET_WIDTH, PARAPET_HEIGHT,
           self.span_length)
        p.create_geometry()
        self.parapets.append(p)

def position_components(self):
    for g, y in zip(self.girders, self.girder_positions_y):
        g.translate(0, y, 0)
    self.deck.translate(0, -self.deck_width / 2, self.deck_z_level)
    parapet_z = self.deck_z_level + self.deck_thickness
    half_w = self.deck_width / 2
    self.parapets[0].translate(0, -half_w - PARAPET_WIDTH / 2,
       parapet_z)
    self.parapets[1].translate(0,  half_w - PARAPET_WIDTH / 2,
       parapet_z)

def assemble(self):
    builder = BRep_Builder()
    compound = TopoDS_Compound()
    builder.MakeCompound(compound)
    for g in self.girders:
        builder.Add(compound, g.get_shape())
    builder.Add(compound, self.deck.get_shape())
    for p in self.parapets:
        builder.Add(compound, p.get_shape())
    self.assembly_shape = compound
```

Listing 2.5. `BridgeModel` layout, component creation, and assembly methods

- **Lines 1–9:** `compute_layout()` derives the total deck width from girder spacing and overhang, distributes girder Y-positions symmetrically about the bridge centreline, and sets the deck elevation to match the top of the girder section depth.
- **Lines 11–23:** `create_components()` instantiates one `Girder` object per computed Y-position, a single `Deck` object spanning the full computed width, and two `Parapet` objects. All components call `create_geometry()` to build their solid shapes immediately after instantiation.
- **Lines 25–31:** `position_components()` translates each girder to its computed Y-position, places the deck at the correct elevation and lateral offset, and positions the two parapets at the outer edges of the deck on top of the deck surface.
- **Lines 33–41:** `assemble()` adds all component shapes into a `TopoDS_Compound` using `BRep_Builder`, producing a single topological container that represents the complete bridge without merging individual solid topologies.

Class: **`BridgeModel`** — Visualisation and Export

```
def display(self):
    display, start_display, *_ = init_display()
    view = display.View
    view.SetBackgroundColor(
        Quantity_Color(1.0, 1.0, 1.0, Quantity_TOC_RGB)
    )
    display.display_triedron()
    for g in self.girders:
        display.DisplayShape(
            g.get_shape(),
            color=Quantity_Color(0.7, 0.7, 0.75, Quantity_TOC_RGB)
        )
    display.DisplayShape(
        self.deck.get_shape(),
        color=Quantity_Color(0.8, 0.8, 0.8, Quantity_TOC_RGB)
    )
    for p in self.parapets:
        display.DisplayShape(
            p.get_shape(),
            color=Quantity_Color(0.9, 0.9, 0.85, Quantity_TOC_RGB)
        )
    display.FitAll()
    start_display()

```

```
25 def export(self):
26     if SAVE_STEP:
27         write_step_file(self.assembly_shape, STEP_FILENAME)
28     if SAVE_BREP:
29         breptools.Write(self.assembly_shape, BREP_FILENAME)
```

Listing 2.6. `BridgeModel` display and export methods

- **Lines 1–6:** The OCC display is initialised and the background colour is set to white using `Quantity_Color` with RGB values `(1.0, 1.0, 1.0)`.
- **Lines 7–21:** Each component type is displayed with a distinct colour: steel gray (`0.7, 0.7, 0.75`) for girders, concrete gray (`0.8, 0.8, 0.8`) for the deck slab, and light beige (`0.9, 0.9, 0.85`) for parapets. The viewer fits the entire assembly in view before starting the interactive display loop.
- **Lines 25–28:** `export()` checks the `SAVE_STEP` and `SAVE_BREP` flags and writes the assembled compound to the configured filenames. STEP export uses `write_step_file()` from `OCC.Extend.DataExchange`; BREP export uses `breptools.Write()` from the OpenCASCADE core.

2.5 Documentation

The implemented parametric bridge model successfully produces a fully adjustable 3D CAD model of a short-span steel girder bridge from a single Python script. All component geometry is driven by top-level configuration variables, and the model can be regenerated for any valid combination of span length, girder count, girder spacing, section dimensions, and skew angle without modifying any class or function bodies.

The object-oriented architecture cleanly separates geometry creation, spatial positioning, assembly, visualisation, and export into distinct methods within the `BridgeModel` class. The shared `BridgeComponent` base class eliminates duplicated transformation code across the three component types and provides a consistent interface for shape management.

The export pipeline produces both STEP and BREP output files, making the generated geometry immediately usable in external CAD tools such as AutoCAD, SolidWorks, and CATIA, as well as in further pythonOCC processing workflows.

2.5.1 Default Configuration

The default parameter block produces a bridge with the following specifications:

No.	Parameter	Value	Unit
1	Span Length	12,000	mm
2	Number of Girders	3	—
3	Girder Spacing	3,000	mm
4	Overall Deck Width	7,000	mm
5	Girder Section Depth	900	mm
6	Girder Flange Width	300	mm
7	Girder Flange Thickness	16	mm
8	Girder Web Thickness	10	mm
9	Deck Thickness	200	mm
10	Parapet Width	300	mm
11	Parapet Height	1,000	mm
12	Skew Angle	10	degrees

Table 2.1. Default parameter configuration for the bridge model

2.5.2 Directory Structure

```
project/
        bridge_model.py              # Main script
   parameters, classes, entry point
        draw_i_section.py            # I-section factory function
        draw_rectangular_prism.py    # Rectangular prism factory
   function
        portal_frame.py              # Reference example
        test_bridge_model.py         # Unit tests
        bridge_model.step            # Exported STEP file
        bridge_model.brep            # Exported BREP file
        DOCUMENTATION.md             # Project documentation
        README.md                    # Repository overview
        requirements.txt             # Python dependencies
```

Listing 2.7. Repository file structure for the parametric bridge model project

2.5.3 Known Limitations and Future Enhancements

The current implementation has several known limitations that represent opportunities for future development:

- The concrete deck is modelled as a single rectangular prism with no segmentation, which does not reflect the pour-cast segmentation typical in real bridge construction.
- No diaphragms, cross-bracing, or transverse stiffeners are included between girders.

- Parapet geometry is simplified to a rectangular cross-section with no profile detailing.
- Bearing details at girder supports and connection plates between the deck and girders are not modelled.

Planned enhancements include deck segmentation, diaphragm and cross-bracing components, bearing details, connection plates, varying girder cross-sections along the span, a command-line interface for parameter input, and automated dimension validation checks.

Note

The model is built and run entirely from `bridge_model.py`. Running `python bridge_model.py` with a Python environment that has `pythonocc-core` installed will generate the geometry, open the interactive 3D viewer, and write the STEP and BREP export files to the working directory.

CHAPTER 3

Asymmetric Weld Logic for Tension Member

3.1 Problem Statement

For the existing implementation of tension members welded to end gusset plates, the weld geometry generated in CAD was symmetric across both flange sides. This behaviour was inconsistent with the underlying weld length calculations, since the required weld demand was already being computed using centroid-based values such as `Cy` and `Cz`.

Although the design-side calculations accounted for eccentricity and unequal load distribution, the downstream weld distribution logic simplified the flange welds into equal lengths on both sides. As a result, the generated CAD geometry did not accurately represent the actual weld demand for different section profiles such as angles, back-to-back angles, star angles, channels, and back-to-back channels.

The objective of this task was to introduce asymmetric weld logic so that flange weld lengths vary according to the centroid location of the section. This required updating both the design logic and CAD generation flow to preserve the unequal weld distribution from the calculation stage through final geometry creation.

The implementation needed to ensure that:

- **Design integrity:** Existing weld strength and design checks remained unchanged.
- **Length consistency:** Total weld length provided remained consistent with the original design requirements.
- **Separate weld values:** Separate weld values could be maintained for heel and toe flanges.
- **Accurate CAD output:** Final CAD geometry accurately reflected the calculated weld distribution for all supported section profiles.

3.2 Existing Workflow Analysis

The weld distribution logic originally flowed through four major stages: weld demand calculation, CAD object preparation, weld length assignment, and final geometry creation.

1. `angle_weld_length()` in `component.py` computed the required flange weld length based on weld strength, applied force, centroid location (`Cy` / `Cz`), and section dimensions. This function already accounted for eccentricity and unequal force distribution in the weld demand calculation and returned a single length comprising both flange welds.

2. `createTensionCAD()` in `common_logic.py` collected the design outputs and prepared the CAD objects required for geometry creation. At this stage, weld size, weld length, plate dimensions, and member geometry were packaged and passed forward into the CAD layer.

3. `weld_plate_length()` in the design type file `tension_welded.py` determined the final weld length values used for design and CAD generation. Although the weld demand coming from `angle_weld_length()` already reflected centroid-based behaviour, this function simplified the weld distribution by assigning equal flange weld lengths on both sides. As a result, only a single `flange_weld` value was stored and reused throughout the rest of the workflow.

4. `createWeldGeometry()` in the respective CAD files used the stored weld values to place and generate the final weld solids. Since the upstream logic only provided a single flange weld length, the CAD generation stage created symmetric weld geometry for the upper and lower flanges regardless of the actual centroid location of the section.

Because of this workflow, the design calculations internally accounted for centroid effects, but the CAD output still displayed equal flange welds, creating a mismatch between the design intent and the final geometry.

3.3 Tasks Done

3.3.1 Methodology and Process

Stage 1: Tracing Existing Logic

The first step was to understand how weld-related values moved through the codebase from design calculation to CAD generation. Since the weld geometry was being generated symmetrically despite centroid-based calculations already existing in the design layer, it was necessary to identify where the asymmetric behaviour was being lost.

The next step was to trace where this weld length was being consumed. The computed weld values were passed through `createTensionCAD()` and `createStrutWeldedCAD()` in `common_logic.py`, where weld dimensions, plate dimensions, and member geometry were packaged into CAD-ready objects. At this stage, the total flange weld length

returned from `angle_weld_length()` was generally divided equally into two parts and assigned to a single `inline_weld` object, which was then reused for both flange sides.

After this, the focus shifted to `weld_plate_length()` in the respective design type files. This function did not support storing separate weld values for heel and toe flanges. Instead, it maintained only a single `flange_weld` value, which was reused for both flange sides. As a result, even though the total flange weld demand already reflected centroid-based behaviour, the function still treated both flange welds as equal.

Finally, `createWeldGeometry()` in the CAD files was traced to understand how the weld objects were being placed. The CAD class initialisation itself only supported a single `inline_weld` object for horizontal welds. Since this same weld object was duplicated and reused for both upper and lower flange weld segments, the final CAD geometry always produced symmetric flange welds regardless of the section centroid location.

Tracing this dependency chain made it possible to identify the four functions that required modification:

- `angle_weld_length()`
- `createTensionCAD()`
- `weld_plate_length()`
- `createWeldGeometry()`

This also helped establish the correct implementation order, starting from the independent weld calculation function and moving progressively toward the dependent CAD geometry functions.

Stage 2: Implementing New Logic

The implementation was carried out by updating the logic progressively from weld calculation to CAD generation so that the asymmetric weld demand could be preserved throughout the workflow.

- **`component.py` — `angle_weld_length()`:** The function was updated to split the flange weld demand into separate heel and toe weld lengths instead of returning a single weld length value. The web weld force was first deducted from the total force demand. The remaining force was then distributed between the heel and toe flange sides using the centroid location (`Cy` / `Cz`) and section depth. The heel-side and toe-side force shares were converted into separate weld lengths and returned as a tuple containing `(l_heel, l_toe)`. This ensured that the centroid-based weld demand could be preserved directly instead of being re-split equally in later stages.

- **common_logic.py — createTensionCAD():** These functions were updated to create separate `FilletWeld` objects for heel and toe flange welds instead of relying on a single `inline_weld` object. New CAD parameters such as `flange_weld_heel` and `flange_weld_toe` were introduced and passed into `inline_weld_heel` and `inline_weld_toe`. The CAD object initialisation was also updated so that the weld CAD classes could receive and store separate weld objects for both flange sides instead of using a single weld length for all horizontal welds.
- **tension_welded.py — weld_plate_length():** The weld distribution logic was updated to support separate heel and toe flange weld values. The tuple returned from `angle_weld_length()` was unpacked into `l_heel` and `l_toe`, which were then rounded independently and stored as `flange_weld_heel` and `flange_weld_toe`. Additional checks were introduced to handle cases where either heel or toe weld length became non-positive due to excessive web weld demand. The total weld length calculation was also updated to include both heel and toe flange weld contributions instead of assuming equal flange weld lengths.
- **WeldedCAD.py — __init__() and createWeldGeometry():** The CAD class initialisation `__init__()` was updated to accept separate `inline_weld_heel` and `inline_weld_toe` objects. Separate copies of these weld objects were then assigned to the upper and lower horizontal weld segments so that different flange weld lengths could be maintained throughout geometry creation. The weld geometry placement logic in `createWeldGeometry()` was updated to use `inline_weld_heel.L` and `inline_weld_toe.L` independently while positioning the weld origins. As a result, the upper and lower flange weld segments no longer share the same weld length, and the final CAD model reflects the centroid-based asymmetric weld distribution.

3.3.2 Key Implementation Features

- **Centroid-based weld splitting:** Flange weld demand is now calculated separately for heel and toe sides using centroid-based values (`Cy` / `Cz`).
- **Updated return type:** `angle_weld_length()` now returns a tuple of heel and toe weld lengths instead of a single flange weld value.
- **Separate weld attributes:** Separate weld attributes are maintained for `flange_weld_heel` and `flange_weld_toe` throughout the design workflow.
- **Backward compatibility:** Existing weld strength checks and total weld length calculations remain unchanged.
- **Edge-case validation:** New validation checks handle cases where heel or toe weld lengths become non-positive.

- **Dual `FilletWeld` objects:** `createTensionCAD()` and `createStrutWeldedCAD()` now support separate `FilletWeld` objects for heel and toe flange welds.
- **Updated CAD initialisation:** CAD class initialisation supports separate inline weld objects for both flange sides.
- **Independent weld placement:** `createWeldGeometry()` now places heel and toe welds independently instead of reusing a single weld length.
- **Accurate CAD output:** Final CAD geometry reflects centroid-based asymmetric weld distribution across all supported section profiles.
- **Preserved symmetric behaviour:** Existing symmetric behaviour for channels and back-to-back channels is preserved where centroid-based asymmetry is not required.

3.3.3 Process Flow

Step	File	Function	Purpose
1	`component.py`	`angle_weld_length()`	Split flange weld demand into separate heel and toe weld lengths using centroid values and web weld contribution
2	`tension_welded.py`	`weld_plate_length(`	Unpack heel and toe weld lengths, store them separately, and update total weld length calculation
3	`common_logic.py`	`createTensionCAD()` / `createStrutWeldedCAD()`	Create separate `FilletWeld` objects for heel and toe flange welds and pass them into the CAD layer
4	`WeldedCAD.py`	`__init__()`	Update CAD class initialisation to support separate inline weld objects for heel and toe flanges
5	`WeldedCAD.py`	`createWeldGeometry()`	Use independent heel and toe weld lengths while placing and generating the final weld geometry

Table 3.1. Process flow for Asymmetric Weld Logic implementation

3.4 Python Code

This section presents the code changes introduced across the four modified files. Each snippet is accompanied by an explanation of the key modifications and their role in preserving asymmetric weld behaviour through the workflow.

3.4.1 File: component.py

Function: angle_weld_length()

```
1  def angle_weld_length(self, weld_strength, depth_weld, force, C,
       depth):
2      # Force carried by the web (outstanding-leg) weld
3      f_web = weld_strength * depth_weld
4
5      # Force share on the heel side
6      f_heel = force * (1 - C / depth) - f_web / 2
7
8      # Force share on the toe side
9      f_toe = force * (C / depth) - f_web / 2
10
11     # Convert force shares to required weld lengths
12     l_heel = f_heel / weld_strength
13     l_toe = f_toe / weld_strength
14
15     return l_heel, l_toe
```

Listing 3.1. Updated `angle_weld_length()` in `component.py` — returns separate heel and toe weld lengths

- **Lines 2–3:** The web weld force contribution is computed first by multiplying weld strength by the web weld depth.
- **Lines 6–7:** The heel-side force share is derived from the total force demand, accounting for the centroid offset `(1 - C / depth)`, with half the web weld contribution deducted.
- **Lines 10–11:** The toe-side force share is computed symmetrically using the centroid ratio `(C / depth)`, again with half the web weld contribution deducted.
- **Lines 14–15:** Both force shares are converted into independent weld lengths by dividing by the weld strength.
- **Line 17:** The function now returns a tuple `(l_heel, l_toe)` instead of a single value, allowing downstream logic to preserve the asymmetric weld demand.

3.4.2 File: tension_welded.py

Function: weld_plate_length()

```
l_heel, l_toe = self.section_size_1.angle_weld_length(
    weld_strength=self.weld.strength,
    depth_weld=self.web_weld,
    force=self.res_force,
    C=self.section_size_1.Cy,
    depth=self.section_size_1.max_leg
)

self.flange_weld_heel = round_up(l_heel, 1, 10)
self.flange_weld_toe = round_up(l_toe, 1, 10)

self.flange_weld = self.flange_weld_heel

self.weld.length = (
    self.web_weld
    + self.flange_weld_heel
    + self.flange_weld_toe
)
```

Listing 3.2. Updated `weld_plate_length()` in `tension_welded.py` — stores heel and toe weld lengths separately

- **Lines 1–7:** The updated call to `angle_weld_length()` passes all centroid-related parameters and unpacks the returned tuple directly into `l_heel` and `l_toe`.
- **Lines 9–10:** Both weld lengths are rounded independently using `round_up()` and stored as separate instance attributes.
- **Line 12:** The original `flange_weld` attribute is retained for backward compatibility with downstream plate sizing logic.
- **Lines 14–18:** The total weld length is recalculated to include both the heel and toe flange contributions, replacing the previous assumption of equal flange weld lengths.

3.4.3 File: common_logic.py

Function: createTensionCAD()

```
inline_weld_toe = FilletWeld(
    b=float(T.weld.size),
    h=float(T.weld.size),
    L=float(T.flange_weld_toe)
```

```
)

inline_weld_heel = FilletWeld(
    b=float(T.weld.size),
    h=float(T.weld.size),
    L=float(T.flange_weld_heel)
)

tensionCAD = TensionAngleWeldCAD(
    T, member, plate,
    inline_weld_toe, inline_weld_heel, opline_weld,
    weld_plate_array
)
```

Listing 3.3. Updated `createTensionCAD()` in `common_logic.py` — creates separate `FilletWeld` objects for heel and toe

- **Lines 1–5:** A dedicated `FilletWeld` object is created for the toe-side flange weld, using `T.flange_weld_toe` as its length.
- **Lines 7–11:** A second `FilletWeld` object is created for the heel-side flange weld, using `T.flange_weld_heel` as its length.
- **Lines 13–17:** Both weld objects are passed independently into the CAD class constructor, allowing the CAD layer to maintain different weld lengths for each flange side.

3.4.4 File: `WeldedCAD.py`

Function: `__init__()`

```
def __init__(self, Obj, member, plate, inline_weld_toe,
   inline_weld_heel,
             opline_weld, weld_plate_array):

    self.inline_weld_toe = inline_weld_toe
    self.inline_weld_heel = inline_weld_heel

    self.weldHL11 = copy.deepcopy(self.inline_weld_heel)
    self.weldHL12 = copy.deepcopy(self.inline_weld_toe)

    self.weldHR11 = copy.deepcopy(self.inline_weld_heel)
    self.weldHR12 = copy.deepcopy(self.inline_weld_toe)
```

Listing 3.4. Updated `__init__()` in `WeldedCAD.py` — accepts and stores separate weld objects for heel and toe flanges

- **Lines 1–2:** The constructor signature is updated to accept two distinct weld parameters: `inline_weld_toe` and `inline_weld_heel`.
- **Lines 4–5:** Both weld objects are stored as instance attributes for use during geometry creation.
- **Lines 7–11:** Deep copies of the heel and toe weld objects are assigned to the four horizontal weld segment attributes (`weldHL11`, `weldHL12`, `weldHR11`, `weldHR12`), ensuring that each flange side carries its correct weld length independently.

Function: `createWeldGeometry()`

```
1  weldHL12OriginL = numpy.array([
2      -self.plate_intercept + self.inline_weld_toe.L,
3      0.0,
4      -self.opline_weld.L / 2
5  ])
6
7  weldHR11OriginL = numpy.array([
8      -self.plate_intercept + self.member.L -
          self.inline_weld_heel.L,
9      0.0,
10     self.opline_weld.L / 2
11 ])
```

Listing 3.5. Updated `createWeldGeometry()` in `WeldedCAD.py` — uses independent heel and toe weld lengths for weld origin placement

- **Lines 1–5:** The origin of the toe-side weld segment (`weldHL12`) is computed using `inline_weld_toe.L`, positioning it correctly relative to the plate intercept.
- **Lines 7–11:** The origin of the heel-side weld segment (`weldHR11`) is computed using `inline_weld_heel.L`, offset from the far end of the member.
- Both origins are now calculated independently, ensuring that the heel and toe weld segments are placed at their respective correct positions rather than being mirrored from a single shared weld length.

3.5 Documentation

The updated implementation successfully preserves asymmetric weld distribution from the design stage through final CAD generation. Heel-side and toe-side flange weld lengths are now calculated independently using centroid-based values and are maintained separately throughout the workflow.

The final CAD geometry now reflects the actual weld demand instead of generating equal flange welds on both sides. This is especially visible for angle-based sections where the centroid location is closer to one flange side, resulting in different heel and toe weld lengths.

At the same time, the original design checks and total weld length calculations remain unchanged, ensuring that the new implementation does not affect the existing design validation process.

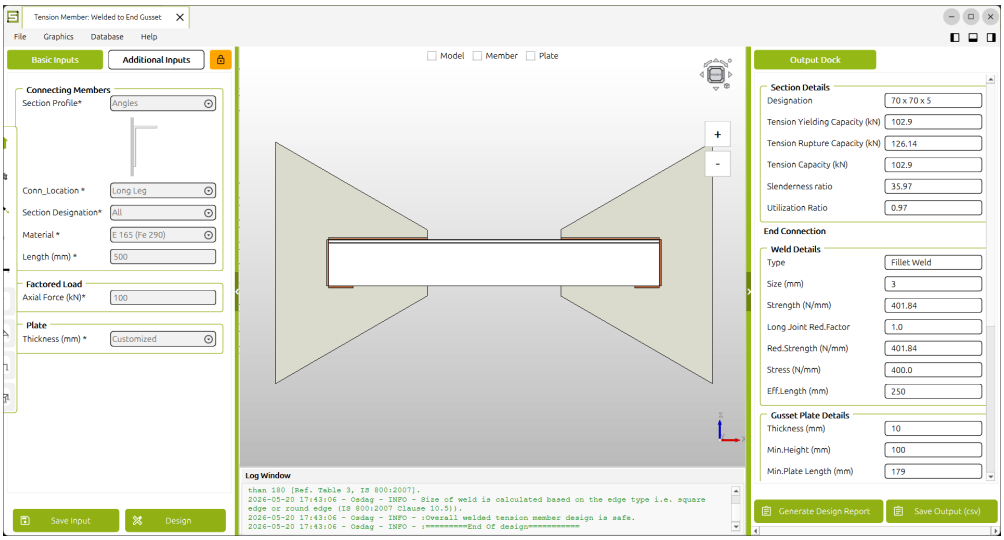

Figure 3.1. Final CAD output showing updated asymmetric weld geometry for an angle section tension member.

The final result is a more accurate weld representation for welded tension and compression members connected to end gusset plates, with CAD output that now matches the underlying centroid-based weld calculation logic.

Note

The changes span four files. Modifications should be applied in the order shown in the process flow table (Section 3.1), starting with `component.py` and ending with `WeldedCAD.py`, to ensure that each dependent stage receives the correct updated values.

CHAPTER 4

Asymmetric Weld Logic for Compression Member

4.1 Problem Statement

For the existing implementation of welded compression members connected to end gusset plates, the weld geometry generated in CAD was symmetric across both flange sides. This behaviour was inconsistent with the underlying weld demand calculation, since centroid-based parameters such as `Cy` and `Cz` already influenced the weld length requirement for angle-based and channel-based sections.

Although the weld calculation logic internally accounted for unequal force distribution caused by the section centroid location, the downstream workflow simplified the flange welds into equal lengths during CAD object creation and geometry placement. As a result, the final CAD model displayed symmetric flange welds even when one flange side carried a larger weld demand than the other.

The objective of this task was to extend the asymmetric weld distribution logic previously implemented for tension members to compression members welded to end gusset plates. This required updating the compression member design workflow and CAD generation pipeline so that separate heel-side and toe-side weld lengths could be preserved and reflected in the final geometry.

The implementation needed to ensure that:

- **Design integrity:** Existing compression member design checks remained unchanged.
- **Length consistency:** Total weld length calculations continued to satisfy the original weld demand.
- **Separate weld values:** Separate heel and toe flange weld values could be maintained throughout the workflow.
- **Accurate CAD output:** Final CAD geometry accurately represented centroid-based asymmetric weld distribution for all supported compression member section profiles.

4.2 Existing Workflow Analysis

The weld distribution workflow for compression members followed a structure similar to the welded tension member workflow, consisting of weld demand calculation, CAD object preparation, weld length assignment, and final geometry generation.

1. `angle_weld_length()` in `component.py` computed the required flange weld lengths using weld strength, applied compressive force, centroid location (`Cy` / `Cz`), and section dimensions. The updated implementation already supported separate heel-side and toe-side weld length calculation and returned both values independently.

2. `createStrutWeldedCAD()` in `common_logic.py` collected the compression member design outputs and created the CAD-ready weld objects. Prior to modification, the workflow reused symmetric inline weld handling for flange weld generation, which prevented the CAD layer from fully utilising the separate heel and toe weld lengths.

3. `weld_plate_length()` in the compression member design files handled weld length assignment and total weld length calculation. Although the updated weld calculation logic already produced separate flange weld values, the compression member workflow still required support for storing and propagating separate heel-side and toe-side weld attributes consistently through the design stage.

4. `createWeldGeometry()` and the CAD class initialisation in the compression member CAD files controlled the placement and generation of weld solids. The earlier implementation reused common inline weld placement assumptions for both flange sides, resulting in symmetric CAD geometry despite the availability of separate weld length values.

Because of this workflow, the design-side weld calculations already supported asymmetric weld distribution, but the compression member CAD generation pipeline still required updates to preserve and display the centroid-based weld behaviour in the final geometry.

4.3 Tasks Done

4.3.1 Tasks Completed

- Extended the asymmetric weld distribution workflow from welded tension members to welded compression members connected to end gusset plates.
- Updated the compression member design workflow to preserve separate heel-side and toe-side flange weld values.
- Modified `createStrutWeldedCAD()` in `common_logic.py` to support independent heel and toe `FilletWeld` objects.

- Updated compression member `weld_plate_length()` logic to unpack and store separate weld lengths returned from `angle_weld_length()`.
- Revised compression member CAD class initialisation to support separate inline weld objects for heel and toe flange welds.
- Updated `createWeldGeometry()` in compression member CAD files to place heel-side and toe-side welds independently.
- Preserved existing compression member weld strength checks and total weld length calculations.
- Ensured centroid-based weld distribution is reflected correctly in the final compression member CAD geometry across all supported section profiles.

4.3.2 Process Flow

Step	File	Function	Purpose
1	`component.py`	`angle_weld_length()`	Reused the updated centroid-based weld calculation logic that returns separate heel-side and toe-side flange weld lengths
2	Compression member design files	`weld_plate_length(`	Unpacked and stored separate heel and toe flange weld values while preserving total weld length calculations
3	`common_logic.py`	`createStrutWeldedCAD()`	Created separate `FilletWeld` objects for heel-side and toe-side flange welds and passed them into the CAD layer
4	Compression member CAD files	`__init__()`	Updated CAD class initialisation to support independent inline weld objects for heel and toe flange welds
5	Compression member CAD files	`createWeldGeometry()`	Used separate heel-side and toe-side weld lengths while placing and generating the final weld geometry

Table 4.1. Process flow for Asymmetric Weld Logic for Compression Member

4.4 Python Code

This section presents the code changes introduced across the four modified files. Each snippet is accompanied by an explanation of the key modifications and their role in preserving asymmetric weld behaviour through the compression member workflow.

4.4.1 File: component.py

Function: angle_weld_length()

```
1  def angle_weld_length(self, weld_strength, depth_weld, force, C,
       depth):
2      # Force carried by the web (outstanding-leg) weld
3      f_web = weld_strength * depth_weld
4
5      # Force share on the heel side     farther from base edge,
         closer to Cy
6      f_heel = force * (1 - C / depth) - f_web / 2
7
8      # Force share on the toe side     closer to base edge, farther
         from Cy
9      f_toe = force * (C / depth) - f_web / 2
10
11     # Convert force shares to required weld lengths
12     l_heel = f_heel / weld_strength
13     l_toe = f_toe / weld_strength
14
15     return l_heel, l_toe
```

Listing 4.1. Updated `angle_weld_length()` in `component.py` — returns separate heel and toe weld lengths for compression members

- **Lines 2–3:** The web weld force contribution is computed first by multiplying weld strength by the web weld depth.
- **Lines 6–7:** The heel-side force share is derived from the total compressive force demand, accounting for the centroid offset `(1 - C / depth)`, with half the web weld contribution deducted.
- **Lines 10–11:** The toe-side force share is computed symmetrically using the centroid ratio `(C / depth)`, again with half the web weld contribution deducted.
- **Lines 14–15:** Both force shares are converted into independent weld lengths by dividing by the weld strength.
- **Line 17:** The function returns a tuple `(l_heel, l_toe)`, serving as the foundation

for preserving asymmetric weld behaviour throughout the compression member workflow.

4.4.2 File: compression_welded.py

Function: weld_plate_length()

```
1  l_heel, l_toe = self.section_property.angle_weld_length(
2      self.weld.strength,
3      self.web_weld,
4      self.res_force,
5      self.section_property.Cy,
6      self.section_property.max_leg
7  )
8
9  _check_flange_positive(l_heel, l_toe, "Single Angle -- Long Leg")
10
11 self.flange_weld_heel = round_up(l_heel, 1, 10)
12 self.flange_weld_toe = round_up(l_toe, 1, 10)
13
14 self.flange_weld = self.flange_weld_heel
15
16 self.weld.length = (
17     self.web_weld
18     + self.flange_weld_heel
19     + self.flange_weld_toe
20 )
```

Listing 4.2. Updated `weld_plate_length()` in `compression_welded.py` — stores heel and toe weld lengths separately

- **Lines 1–7:** The updated call to `angle_weld_length()` passes all centroid-related parameters and unpacks the returned tuple directly into `l_heel` and `l_toe`.
- **Line 9:** A validation helper `_check_flange_positive()` is called to detect cases where either weld length becomes non-positive due to excessive web weld demand.
- **Lines 11–12:** Both weld lengths are rounded independently using `round_up()` and stored as separate instance attributes.
- **Line 14:** The original `flange_weld` attribute is retained for backward compatibility with downstream plate sizing logic.
- **Lines 16–20:** The total weld length is recalculated to include both heel and toe flange weld contributions, replacing the previous assumption of equal flange weld

lengths.

4.4.3 File: common_logic.py

Function: createStrutWeldedCAD()

```
1  inline_weld_toe = FilletWeld(
2      b=float(T.weld.size),
3      h=float(T.weld.size),
4      L=float(T.flange_weld_toe)
5  )
6
7  inline_weld_heel = FilletWeld(
8      b=float(T.weld.size),
9      h=float(T.weld.size),
10     L=float(T.flange_weld_heel)
11 )
12
13 strutCAD = StrutAngleWeldCAD(
14     T, member, plate,
15     inline_weld_toe, inline_weld_heel,
16     opline_weld, weld_plate_array
17 )
```

Listing 4.3. Updated `createStrutWeldedCAD()` in `common_logic.py` — creates separate `FilletWeld` objects for heel and toe

- **Lines 1–5:** A dedicated `FilletWeld` object is created for the toe-side flange weld, using `T.flange_weld_toe` as its length.
- **Lines 7–11:** A second `FilletWeld` object is created for the heel-side flange weld, using `T.flange_weld_heel` as its length.
- **Lines 13–17:** Both weld objects are passed independently into the CAD class constructor, allowing the CAD layer to preserve asymmetric weld lengths during placement and model generation.

4.4.4 File: WeldedCAD.py

Function: __init__()

```
1  def __init__(self, Obj, member, plate,
2               inline_weld_toe, inline_weld_heel,
3               opline_weld, weld_plate_array):
4
```

```
5       self.inline_weld_toe = inline_weld_toe
6       self.inline_weld_heel = inline_weld_heel
7
8       self.weldHL11 = copy.deepcopy(self.inline_weld_heel)
9       self.weldHL12 = copy.deepcopy(self.inline_weld_toe)
10
11      self.weldHR11 = copy.deepcopy(self.inline_weld_heel)
12      self.weldHR12 = copy.deepcopy(self.inline_weld_toe)
```

Listing 4.4. Updated `__init__()` in `WeldedCAD.py` — accepts and stores separate weld objects for heel and toe flanges

- **Lines 1–3:** The constructor signature is updated to accept two distinct weld parameters: `inline_weld_toe` and `inline_weld_heel`.
- **Lines 5–6:** Both weld objects are stored as instance attributes for use during geometry creation.
- **Lines 8–12:** Deep copies of the heel and toe weld objects are assigned to the four horizontal weld segment attributes (`weldHL11`, `weldHL12`, `weldHR11`, `weldHR12`), ensuring that each flange side carries its correct weld length independently without shared references.

Function: **`createWeldGeometry()`**

```
1   # Left end     BOTTOM flange (TOE, shorter)
2   weldHL12OriginL = numpy.array([
3       -self.plate_intercept + self.inline_weld_toe.L,
4       0.0,
5       -self.opline_weld.L / 2
6   ])
7
8   # Right end     TOP flange (HEEL, longer)
9   weldHR11OriginL = numpy.array([
10      -self.plate_intercept + self.member.L -
            self.inline_weld_heel.L,
11      0.0,
12      self.opline_weld.L / 2
13  ])
```

Listing 4.5. Updated `createWeldGeometry()` in `WeldedCAD.py` — uses independent heel and toe weld lengths for weld origin placement

- **Lines 2–6:** The origin of the toe-side weld segment (`weldHL12`) is computed using

`inline_weld_toe.L`, positioning it correctly relative to the plate intercept on the bottom flange.

- **Lines 9–13:** The origin of the heel-side weld segment (`weldHR11`) is computed using `inline_weld_heel.L`, offset from the far end of the member on the top flange.
- Both origins are now calculated independently, removing the earlier assumption of symmetric weld placement and allowing the final CAD geometry to accurately reflect centroid-based asymmetric weld distribution.

4.5 Documentation

The updated implementation successfully extends centroid-based asymmetric weld distribution to welded compression members connected to end gusset plates. Separate heel-side and toe-side flange weld lengths are now preserved throughout the complete workflow, starting from weld demand calculation and continuing through CAD object creation, weld length assignment, and final geometry generation.

The compression member CAD workflow now supports independent inline weld objects for heel and toe flange welds, allowing different weld lengths to be placed and generated correctly in the final model. The updated weld placement logic ensures that weld geometry reflects the actual centroid-based weld demand instead of producing equal flange welds on both sides.

Additional validation checks were also introduced in the weld distribution stage to detect cases where excessive web weld demand could lead to non-positive heel-side or toe-side flange weld lengths. This helped preserve design stability while maintaining compatibility with the existing weld strength and total weld length calculations.

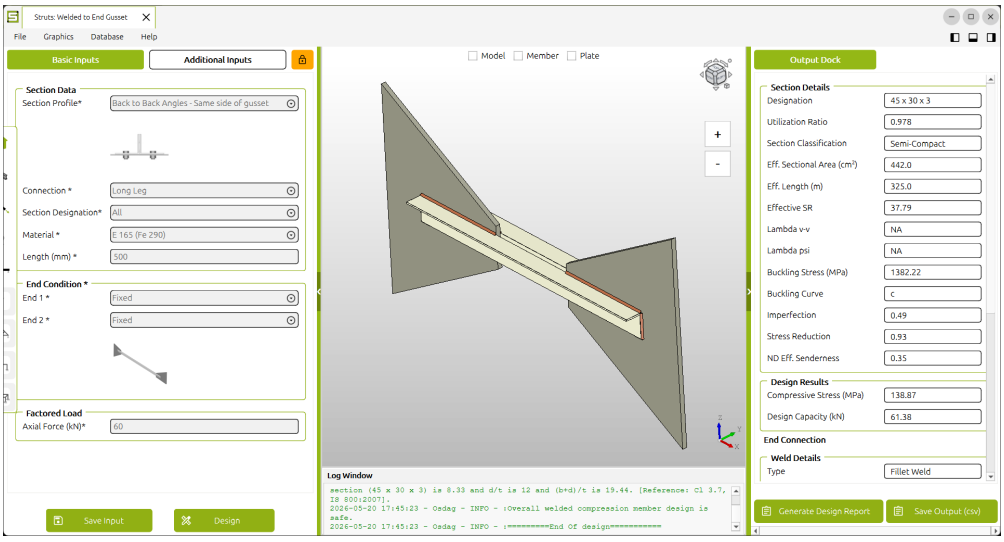

Figure 4.1. Final CAD output showing updated asymmetric weld geometry for a compression member.

The final result is a consistent asymmetric weld workflow across both welded tension and compression member connections, ensuring that the generated CAD geometry now accurately represents the underlying centroid-based weld calculation logic for all supported section profiles.

Note

The changes span four files. Modifications should be applied in the order shown in the process flow table (Section 4.1), starting with `component.py` and ending with `WeldedCAD.py`, to ensure that each dependent stage receives the correct updated values.

CHAPTER 5

Conclusions

5.1 Tasks Accomplished

During the internship, key weld design and CAD workflow components were developed and integrated into the structural steel design framework, improving centroid-based weld representation and CAD generation capabilities.

- ***Asymmetric Weld Logic for Tension Members:*** *Implemented centroid-based asymmetric weld distribution for welded tension members connected to end gusset plates by updating weld calculation, design workflow, CAD object handling, and geometry generation logic.*
- ***Asymmetric Weld Logic for Compression Members:*** *Extended the asymmetric weld workflow to welded compression members, enabling separate heel-side and toe-side flange weld generation while preserving existing weld design checks and CAD integration.*

5.2 Skills Developed

- **Programming and Development:** *Strengthened Python development skills through modification of multi-file design and CAD workflows, object handling, and geometry generation logic within a structural steel design software framework.*
- **Structural Design and CAD Modeling:** *Developed understanding of centroid-based weld distribution, welded connection behavior, and parametric CAD generation for structural steel members and gusset plate connections.*
- **Software Engineering:** *Applied dependency tracing, modular code integration, backward compatibility handling, and staged workflow implementation across design and CAD generation layers.*
- **Problem-Solving and Analysis:** *Enhanced the ability to trace calculation workflows, identify logic mismatches between design and CAD generation, and implement solutions that preserved both engineering behavior and software stability.*
- **Technical Writing:** *Improved technical documentation and reporting skills through structured workflow analysis, implementation documentation, and detailed explana-*

tions.

Bibliography

[1] S. Ghosh et al., "Osdag: A software for structural steel design using is 800:2007," in *Advances in Structural Technologies*, S. Adhikari, A. Dutta, and S. Choudhury, Eds., ser. Lecture Notes in Civil Engineering, vol. 81, Singapore: Springer Singapore, 2021, pp. 219–231, ISBN: 9789811552342. DOI: 10.1007/978-981-15-5235-9_17 [Online]. Available: http://link.springer.com/10.1007/978-981-15-5235-9_17

[2] FOSSEE Project, *Fossee news - january 2018, vol 1 issue 3*, https://static.fossee.in/fossee/newsletters/Newsletter-Jan2018.pdf, Accessed: 2024-12-05.

[3] FOSSEE Project, *Osdag website*, https://osdag.fossee.in/, Accessed: 2024-12-05.

APPENDIX A

Appendix

A.1 Work Reports

Date	Day	Task	Hours
16-Feb-2026	Monday	Internship onboarding, project introduction, and development setup instructions	4
17-Feb-2026	Tuesday	Installed required software, IDEs, libraries, and configured development environment	5
18-Feb-2026	Wednesday	Meeting with mentors regarding task allocation and asymmetric weld logic workflow	3
19-Feb-2026	Thursday	Explored OSDAG repository structure and CAD generation workflow	5
20-Feb-2026	Friday	Studied weld design modules and object interaction flow	5
21-Feb-2026	Saturday	Reviewed existing welded connection implementations and dependencies	4
22-Feb-2026	Sunday	LEAVE	0
23-Feb-2026	Monday	Analyzed design workflow for welded tension and compression members	5
24-Feb-2026	Tuesday	Meeting with mentors regarding weld logic understanding and workflow tracing	3
25-Feb-2026	Wednesday	Studied CAD geometry generation and weld placement workflow	5
26-Feb-2026	Thursday	Traced weld length calculations through dependent design files	5
27-Feb-2026	Friday	Investigated relationship between centroid values and weld geometry	5
28-Feb-2026	Saturday	Meeting with mentors to discuss understanding of existing architecture	3
01-Mar-2026	Sunday	LEAVE	0
02-Mar-2026	Monday	Planned initial geometry-based asymmetric weld implementation approach	5

(Continued from previous page)

Date	Day	Task	Hours
03-Mar-2026	Tuesday	Explored weld object creation and CAD instantiation logic	5
04-Mar-2026	Wednesday	Meeting with mentors regarding geometric approach and implementation direction	3
05-Mar-2026	Thursday	Traced object flow from design layer to CAD geometry generation	5
06-Mar-2026	Friday	Identified core functions responsible for weld distribution workflow	5
07-Mar-2026	Saturday	Prepared revised implementation direction focusing on core weld logic	4
08-Mar-2026	Sunday	LEAVE	0
09-Mar-2026	Monday	MID-SEMESTER EXAMINATION LEAVE	0
10-Mar-2026	Tuesday	MID-SEMESTER EXAMINATION LEAVE	0
11-Mar-2026	Wednesday	MID-SEMESTER EXAMINATION LEAVE	0
12-Mar-2026	Thursday	MID-SEMESTER EXAMINATION LEAVE	0
13-Mar-2026	Friday	MID-SEMESTER EXAMINATION LEAVE	0
14-Mar-2026	Saturday	MID-SEMESTER EXAMINATION LEAVE	0
15-Mar-2026	Sunday	LEAVE	0
16-Mar-2026	Monday	MID-SEMESTER EXAMINATION LEAVE	0
17-Mar-2026	Tuesday	MID-SEMESTER EXAMINATION LEAVE	0
18-Mar-2026	Wednesday	MID-SEMESTER EXAMINATION LEAVE	0
19-Mar-2026	Thursday	MID-SEMESTER EXAMINATION LEAVE	0
20-Mar-2026	Friday	MID-SEMESTER EXAMINATION LEAVE	0
21-Mar-2026	Saturday	Reviewed pending implementation notes after mid-semester examinations	3
22-Mar-2026	Sunday	LEAVE	0
23-Mar-2026	Monday	Finalized centroid-based asymmetric weld implementation plan	5
24-Mar-2026	Tuesday	Meeting with mentors to review implementation workflow and dependencies	3
25-Mar-2026	Wednesday	Implemented centroid-based weld distribution logic in weld calculation functions	5
26-Mar-2026	Thursday	Updated CAD object creation workflow for heel and toe weld handling	5
27-Mar-2026	Friday	Modified weld geometry placement logic for asymmetric weld representation	5

(Continued from previous page)

Date	Day	Task	Hours
28-Mar-2026	Saturday	Mentor review meeting regarding implemented asymmetric weld workflow	3
29-Mar-2026	Sunday	LEAVE	0
30-Mar-2026	Monday	Performed iterative testing and debugging on welded tension member workflow	5
31-Mar-2026	Tuesday	Reviewed geometry alignment and weld distribution across section profiles	5
01-Apr-2026	Wednesday	Meeting with mentors regarding edge cases and design compatibility	3
02-Apr-2026	Thursday	Refined weld distribution logic and validated CAD outputs	5
03-Apr-2026	Friday	Raised pull request for asymmetric weld logic in tension members	4
04-Apr-2026	Saturday	Started extending asymmetric weld workflow to compression members	4
05-Apr-2026	Sunday	LEAVE	0
06-Apr-2026	Monday	Updated compression member weld distribution and CAD workflow	5
07-Apr-2026	Tuesday	Meeting with mentors regarding compression member implementation progress	3
08-Apr-2026	Wednesday	Implemented asymmetric weld placement for compression member CAD geometry	5
09-Apr-2026	Thursday	Tested compression member weld geometry across supported profiles	5
10-Apr-2026	Friday	Raised pull request for compression member asymmetric weld implementation	4
11-Apr-2026	Saturday	Reviewed pull request feedback and identified weld logic inconsistency	3
12-Apr-2026	Sunday	LEAVE	0
13-Apr-2026	Monday	Resolved weld logic inconsistency and updated dependent weld calculations	5
14-Apr-2026	Tuesday	Validated updated weld distribution workflow and verified CAD outputs	4
15-Apr-2026	Wednesday	TERM-END EXAMINATION LEAVE	0
16-Apr-2026	Thursday	TERM-END EXAMINATION LEAVE	0
17-Apr-2026	Friday	TERM-END EXAMINATION LEAVE	0
18-Apr-2026	Saturday	TERM-END EXAMINATION LEAVE	0

(Continued from previous page)

Date	Day	Task	Hours
19-Apr-2026	Sunday	LEAVE	0
20-Apr-2026	Monday	TERM-END EXAMINATION LEAVE	0
21-Apr-2026	Tuesday	TERM-END EXAMINATION LEAVE	0
22-Apr-2026	Wednesday	TERM-END EXAMINATION LEAVE	0
23-Apr-2026	Thursday	TERM-END EXAMINATION LEAVE	0
24-Apr-2026	Friday	TERM-END EXAMINATION LEAVE	0
25-Apr-2026	Saturday	TERM-END EXAMINATION LEAVE	0
26-Apr-2026	Sunday	LEAVE	0
27-Apr-2026	Monday	TERM-END EXAMINATION LEAVE	0
28-Apr-2026	Tuesday	TERM-END EXAMINATION LEAVE	0
29-Apr-2026	Wednesday	TERM-END EXAMINATION LEAVE	0
30-Apr-2026	Thursday	TERM-END EXAMINATION LEAVE	0
01-May-2026	Friday	Started internship documentation and report preparation	5
02-May-2026	Saturday	Compiled implementation notes and organized CAD output references	4
03-May-2026	Sunday	LEAVE	0
04-May-2026	Monday	Started internship documentation and report preparation	5
05-May-2026	Tuesday	Meeting with mentors regarding report structure and technical documentation	3
06-May-2026	Wednesday	Documented asymmetric weld workflow and implementation methodology	5
07-May-2026	Thursday	Prepared code explanations and CAD output documentation	5
08-May-2026	Friday	Compiled implementation screenshots and workflow diagrams	4
09-May-2026	Saturday	Reviewed documentation progress and finalized remaining sections	3
10-May-2026	Sunday	LEAVE	0
11-May-2026	Monday	Finalized internship report and technical documentation	5
12-May-2026	Tuesday	Meeting with mentors for final review and feedback incorporation	3
13-May-2026	Wednesday	Performed final formatting, proofreading, and corrections	4

(Continued from previous page)

Date	Day	Task	Hours
14-May-2026	Thursday	Prepared final report submission and consolidated project artifacts	4
15-May-2026	Friday	Internship wrap-up and final submission of documentation	3