Osdag Screening Task Report

Project Title

Generation of 2D and 3D Shear Force and Bending Moment Diagrams from Xarray Dataset using Python

Screening Task – Osdag (FOSSEE, IIT Bombay)

Submitted by:

Sneha Shaji

4th Year B.Tech Student, School of Computing

VIT Bhopal University

31 January 2026

Table of Contents

List of figures

Abstract

This project was carried out as part of the Osdag screening assignment to visualize internal force results obtained from a bridge grillage model stored in an Xarray dataset. The objective of the work is to generate two-dimensional shear force and bending moment diagrams for the central longitudinal girder and to generate three-dimensional shear force and bending moment diagrams for all longitudinal girders of the bridge.

The internal force components (Vy and Mz) are extracted directly from the provided NetCDF Xarray dataset without modifying the sign convention. The first task focuses on producing clear and continuous two-dimensional diagrams for the central girder using Matplotlib. The second task focuses on generating MIDAS-style three-dimensional visualizations for all girders using Plotly, where the force and moment values are shown as vertical extrusions over the bridge geometry.

The developed codes demonstrate correct usage of Xarray for data extraction, proper use of node coordinates and element connectivity, and visually understandable diagrams suitable for post-processing and interpretation.

1. Introduction

In bridge and structural analysis, shear force diagrams and bending moment diagrams are essential tools for understanding the internal behaviour of structural members. Modern structural analysis tools usually store results in structured numerical datasets rather than directly producing plots.

In this screening task, a NetCDF file containing internal force and moment results of a grillage bridge model is provided in Xarray format. The task is to process this dataset using Python and generate both two-dimensional and three-dimensional visualizations similar to professional post-processing tools such as MIDAS.

This work focuses on extracting shear force (Vy) and bending moment (Mz) components and using the provided node coordinates and element connectivity to create meaningful diagrams for the bridge girders.

2. Problem Statement

The main objectives of this project are:

1. To extract bending moment (Mz) and shear force (Vy) values from the given Xarray dataset for a specified sequence of elements forming the central longitudinal girder and generate continuous two-dimensional SFD and BMD plots.
2. To generate three-dimensional shear force and bending moment diagrams for all longitudinal girders of the bridge using the given node coordinates and element connectivity, following a visualization style similar to MIDAS.

The plots must use the sign convention directly as stored in the dataset and must be continuous and visually clear.

3. Methodology

The complete work is divided into two parts corresponding to Task-1 and Task-2.

3.1 Data and Input Files

The following input files are used in both tasks:

- A NetCDF file containing the internal forces and moments stored as an Xarray dataset.
- A `node.py` file containing node coordinates.
- An `element.py` file containing element connectivity.

The Xarray dataset stores the internal forces in a two-dimensional array named `forces`, indexed by:

- Element
- Component

The required components for this task are:

- `Mz_i`, `Mz_j` – bending moment at start and end of each element
- `Vy_i`, `Vy_j` – shear force at start and end of each element

4. Task-1: 2D Shear Force and Bending Moment Diagrams

4.1 Objective

The objective of Task-1 is to generate two-dimensional bending moment and shear force diagrams for the central longitudinal girder formed by the following elements:

[15, 24, 33, 42, 51, 60, 69, 78, 83]

4.2 Method Used

The code begins by loading the node coordinates and element connectivity from the provided Python files. After that, the NetCDF dataset is opened using Xarray.

Since the dataset stores all force components inside a single variable called forces, the data is accessed using both element number and component name.
For example, bending moment at the start of element 15 is accessed using:

forces(Element=15, Component='Mz_i')

The central girder element list is defined explicitly. The code verifies that the girder is continuous by checking that the end node of each element matches the start node of the next element. This verification ensures that the plotted diagram represents a single continuous member. For each element of the central girder:

- The start and end nodes are identified.
- Their coordinates are obtained from the node dictionary.
- The x-coordinate of the nodes is used as the position along the girder.
- The values of Mz_i, Mz_j, Vy_i and Vy_j are extracted from the Xarray dataset.

To maintain continuity of the diagram, only the first element contributes both its start and end values, while for all subsequent elements only the end value is appended.
This avoids duplication at common nodes. The extracted values are stored in arrays:

- position along the girder
- bending moment values
- shear force values

4.3 Plot Generation

Matplotlib is used to generate the plots. Two subplots are created:

- the upper plot shows the bending moment diagram
- the lower plot shows the shear force diagram

Both plots use:

- markers at each data point
- filled area under the curve
- grid lines
- proper axis labels and titles

An interactive cursor is added using the mplcursors library. This allows the user to hover over any point on the diagram and see the exact position and force or moment value. The sign of the data is not modified at any stage. The diagrams directly represent the sign convention stored in the dataset.

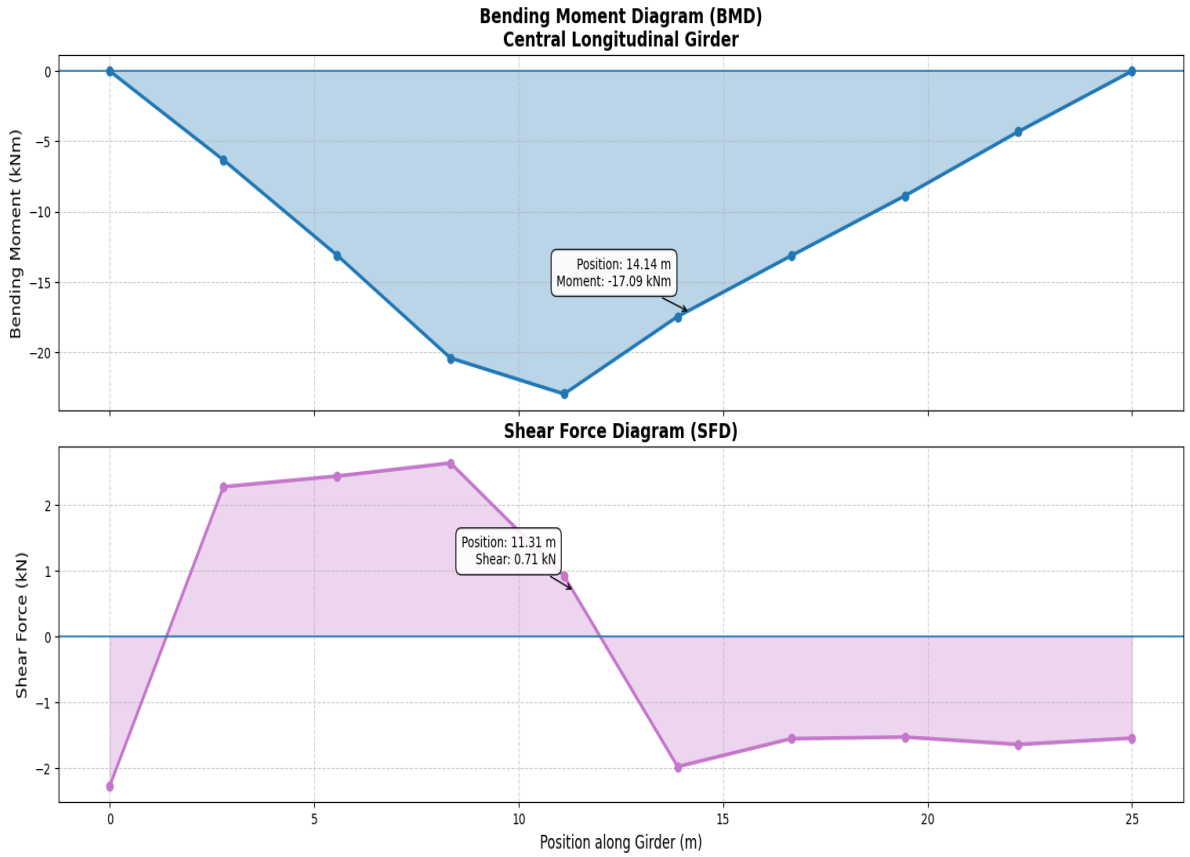

Figure 1 Shear Force and Bending Moment Diagram for the central longitudinal girder

CODE IMPLEMENTATION

1. Import required libraries

```
import matplotlib.pyplot as plt
import xarray as xr
import numpy as np
import plotly.graph_objects as go
from plotly.subplots import make_subplots
```

This imports all libraries needed for reading the dataset and plotting the diagrams.

2. Load node, element and result files

```
print("Loading input files...")

import sys
sys.path.append('.')

from node import nodes
from element import members

print(f"Nodes loaded   : {len(nodes)}")
print(f"Elements loaded: {len(members)}")

ds = xr.open_dataset(r'./screening_task.nc')

print("Xarray dataset loaded")
print("Available components:", ds['Component'].values)
```

This loads the node coordinates, element connectivity and the Xarray result file containing forces and moments.

3. Define the central longitudinal girder

```
central_elements = [15, 24, 33, 42, 51, 60, 69, 78, 83]

print("\nCentral girder elements:", central_elements)
```

This list defines the element numbers that form the central girder.

4. Check connectivity of the girder elements

```
for i, ele in enumerate(central_elements):
    ni, nj = members[ele]
    if i > 0:
        prev_nj = members[central_elements[i - 1]][1]
        if ni != prev_nj:
            print("Warning: element connectivity issue at element", ele)

print("Central girder connectivity verified")
```

This checks whether each element starts from the end node of the previous element.

5. Create storage for diagram data

These lists will store positions, bending moments and shear forces along the girder.

```
positions = []
bending_moments = []
shear_forces = []

print("\nExtracting Mz and Vy for central girder...")
```

6. Loop through each girder element and extract values

```
for idx, ele in enumerate(central_elements):

    node_i = members[ele][0]
    node_j = members[ele][1]

    coord_i = nodes[node_i]
    coord_j = nodes[node_j]

    # position along girder (x direction)
    pos_i = coord_i[0]
    pos_j = coord_j[0]

    try:
        mz_i = float(ds['forces'].sel(Element=ele,
Component='Mz_i').values)
        mz_j = float(ds['forces'].sel(Element=ele,
Component='Mz_j').values)
        vy_i = float(ds['forces'].sel(Element=ele,
Component='Vy_i').values)
        vy_j = float(ds['forces'].sel(Element=ele,
Component='Vy_j').values)
    except KeyError as e:
        print("Error reading element", ele)
        print("Available components:", ds['Component'].values)
        raise
```

For each element, the code reads its start and end node, x-positions and the corresponding Mz and Vy values from the dataset.

7. Build continuous arrays along the girder

```
if idx == 0:
        positions.extend([pos_i, pos_j])
        bending_moments.extend([mz_i, mz_j])
        shear_forces.extend([vy_i, vy_j])
    else:
        positions.append(pos_j)
        bending_moments.append(mz_j)
        shear_forces.append(vy_j)
```

This avoids duplicating joint points and creates a continuous profile along the girder.

8. Convert lists to NumPy arrays

```
positions = np.array(positions)
bending_moments = np.array(bending_moments)
shear_forces = np.array(shear_forces)

print("Extraction completed")
print("Number of points:", len(positions))
```

This converts all extracted values into arrays suitable for plotting.

9. Create the figure for SFD and BMD

```
import mplcursors

fig, axes = plt.subplots(2, 1, figsize=(14, 10), sharex=True)
```

This creates two vertically stacked plots sharing the same x-axis.

10. Plot the bending moment diagram

```
ax_bmd = axes[0]
line_bmd, = ax_bmd.plot(
    positions,
    bending_moments,
    marker='o',
    linewidth=2.5
)

ax_bmd.fill_between(positions, bending_moments, alpha=0.3)
ax_bmd.axhline(0, linewidth=1.2)
ax_bmd.grid(True, linestyle='--', alpha=0.6)

ax_bmd.set_ylabel("Bending Moment (kNm)", fontsize=12)
ax_bmd.set_title(
    "Bending Moment Diagram (BMD)\nCentral Longitudinal Girder",
    fontsize=14,
    fontweight='bold'
)
```

This draws the bending moment diagram of the central girder.

11. Plot the shear force diagram

```
ax_sfd = axes[1]
line_sfd, = ax_sfd.plot(
    positions,
    shear_forces,
    marker='o',
    linewidth=2.5,
    color="#C476CA"
)

ax_sfd.fill_between(positions, shear_forces, color="#C476CA", alpha=0.3)
ax_sfd.axhline(0, linewidth=1.2)
ax_sfd.grid(True, linestyle='--', alpha=0.6)

ax_sfd.set_xlabel("Position along Girder (m)", fontsize=12)
ax_sfd.set_ylabel("Shear Force (kN)", fontsize=12)
ax_sfd.set_title(
    "Shear Force Diagram (SFD)",
    fontsize=14,
    fontweight='bold'
)
```

This draws the shear force diagram using the same girder positions.

12. Enable interactive hover values

```
cursor_bmd = mplcursors.cursor(line_bmd, hover=True)
cursor_sfd = mplcursors.cursor(line_sfd, hover=True)

@cursor_bmd.connect("add")
def on_add_bmd(sel):
    x, y = sel.target
    sel.annotation.set_text(
        f"Position: {x:.2f} m\nMoment: {y:.2f} kNm"
    )
    sel.annotation.get_bbox_patch().set(fc="white", alpha=0.9)

@cursor_sfd.connect("add")
def on_add_sfd(sel):
    x, y = sel.target
    sel.annotation.set_text(
        f"Position: {x:.2f} m\nShear: {y:.2f} kN"
    )
    sel.annotation.get_bbox_patch().set(fc="white", alpha=0.9)
```

This shows exact position and force values when the user hovers over any point.

13. Display the plots

```
plt.tight_layout()

plt.show()
```

This final command displays the interactive bending moment and shear force diagrams.

5. Task-2: 3D Shear Force and Bending Moment Diagrams

5.1 Objective

The objective of Task-2 is to generate three-dimensional shear force and bending moment diagrams for all longitudinal girders of the bridge. The visualization should show the bridge framing and display the force or moment values as vertical extrusions in a style similar to MIDAS post-processing.

5.2 Girder Definition

Five girders are defined exactly according to the task specification. Each girder is defined using its element tags and node tags. A reverse mapping from element number to girder name is created so that each element can be associated with its respective girder during plotting.

5.3 Data Loading and Component Selection

The NetCDF dataset is loaded using Xarray. Depending on the selected result type:

- bending moment uses Mz_i and Mz_j
- shear force uses Vy_i and Vy_j

The code automatically switches between these components based on the selected diagram type.

5.4 Scaling of Results

To ensure that the vertical extrusions are clearly visible and comparable, the maximum absolute value of the selected result component is first calculated over all elements. A global scale factor is then computed based on this maximum value. For the shear force diagram, an additional scaling factor is applied to improve visibility. This ensures that the plotted results are visually balanced and readable without modifying the actual numerical values.

5.5 Geometry and Surface Construction

For every element in the model:

- the start and end node coordinates are obtained
- the corresponding force or moment values at the start and end are extracted

Each element is divided into several small segments.
For each segment, a linear interpolation is performed between the start and end node positions and between the start and end force values. A thin vertical surface is generated for every element using a triangular mesh. The base of the surface lies on the actual bridge geometry, and the top edge represents the force or moment magnitude extruded in the vertical direction.

Boundary edges are drawn at:

- the start of the element
- the end of the element
- the top connecting line between start and end values

This helps to clearly visualize the shape of the diagram, similar to professional structural software.

5.6 Structural Framing

Before drawing the force and moment surfaces, the centre lines of all elements are drawn using the actual node coordinates. This shows the bridge framing and provides a clear reference for the location of each girder.

5.7 Visual Styling and Interaction

Plotly is used to create the three-dimensional visualization. The following features are implemented:

- visible grids on all axes
- labelled axes
- pastel colour scheme for better visibility
- semi-transparent surfaces, lighting effects for depth perception
- colour bar showing the magnitude of the result

Each girder is grouped so that it can be controlled through interactive buttons. Buttons are added to:

- show all girders
- show individual girders

This allows the user to focus on specific girders during inspection. The final output is saved as interactive HTML files for both bending moment and shear force diagrams.

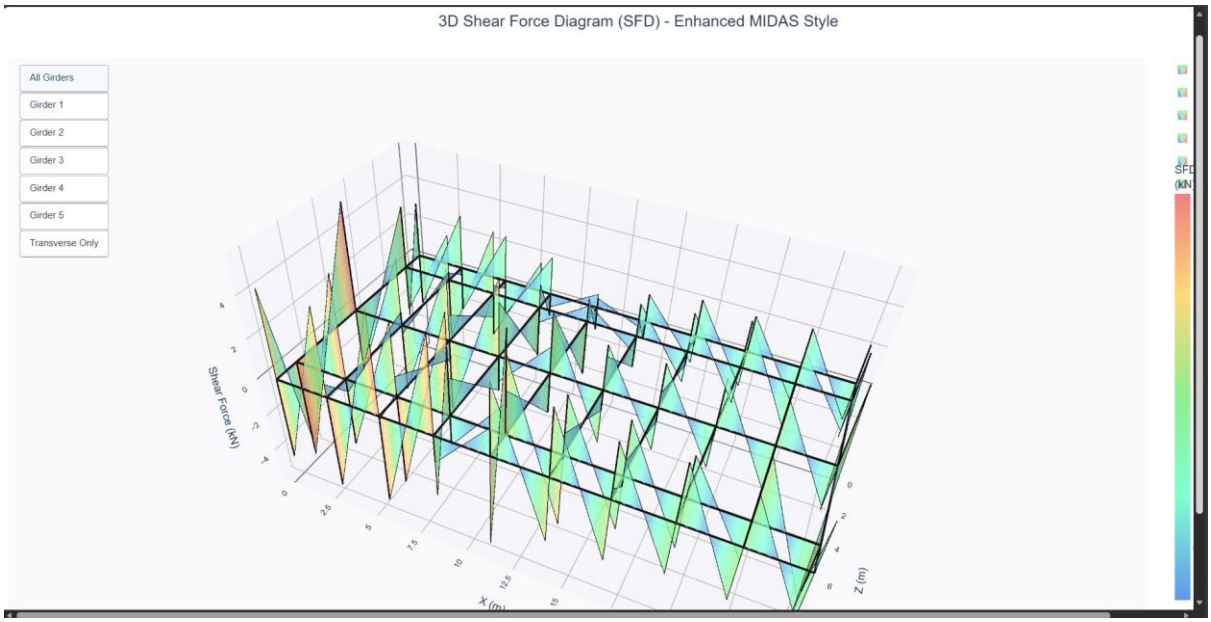

Figure 2 Shear Force Model 3d (for all girders). Each girder can be selected as needed from the selection boxes in the left side.

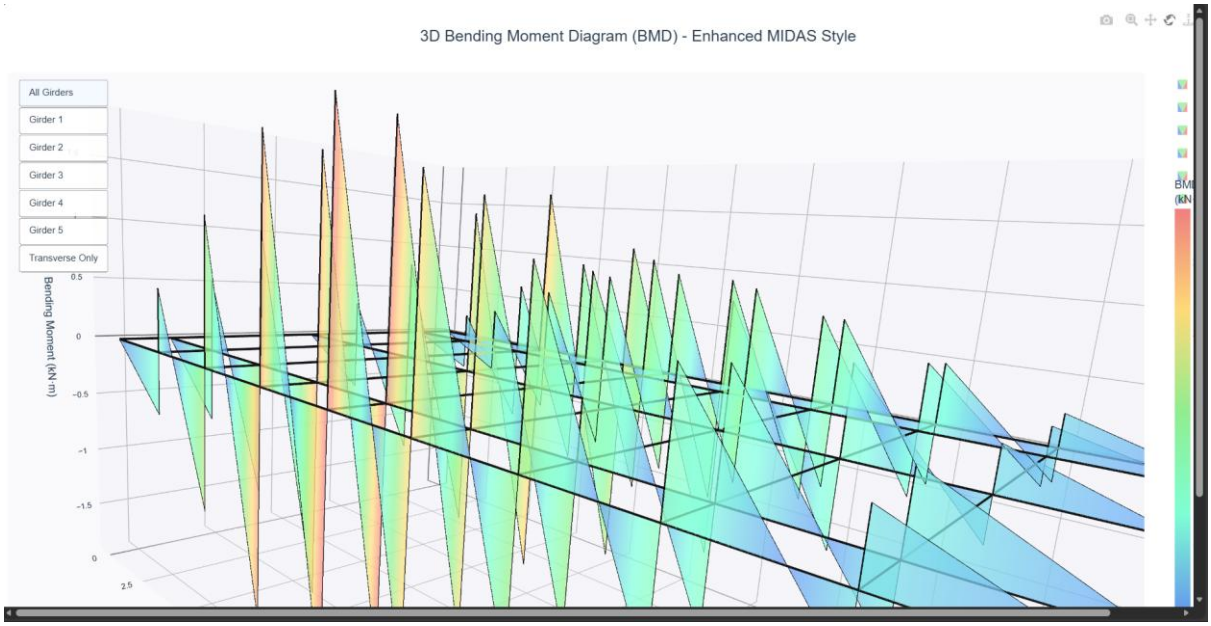

Figure 3 3D Bending Moment Diagram for each girder. Selection for each grider is done through selection of the needed girder.

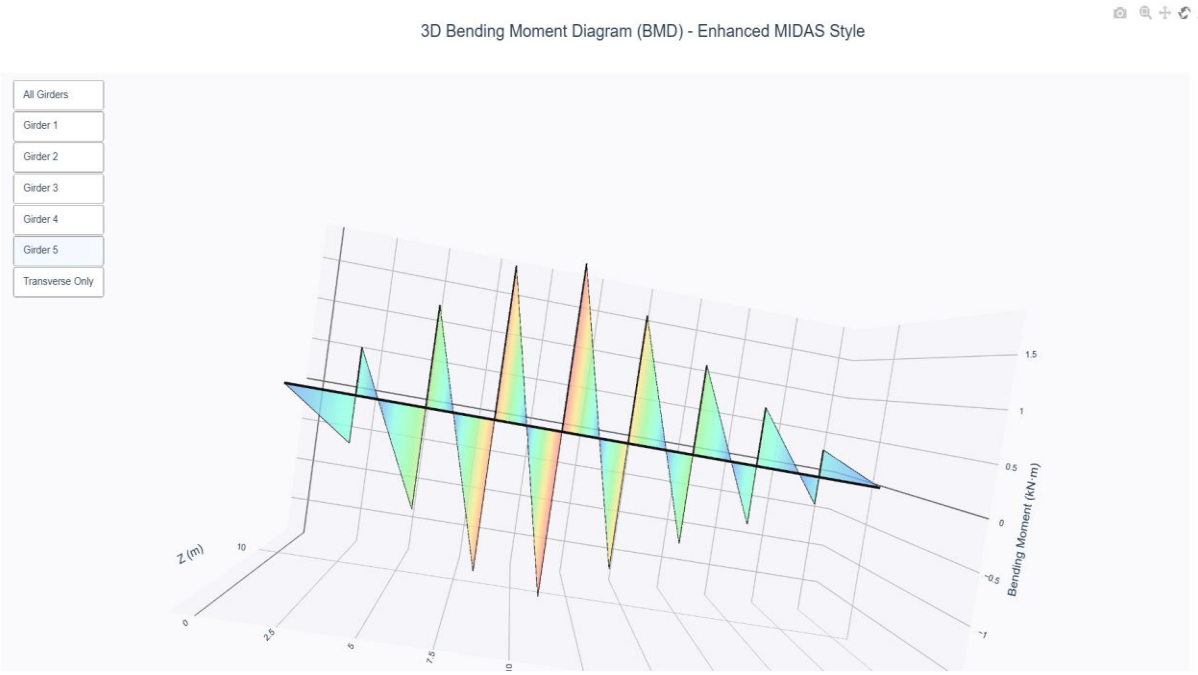

Figure 4 Each girder can be selected individually to view its modelling, enabling better and clear understanding

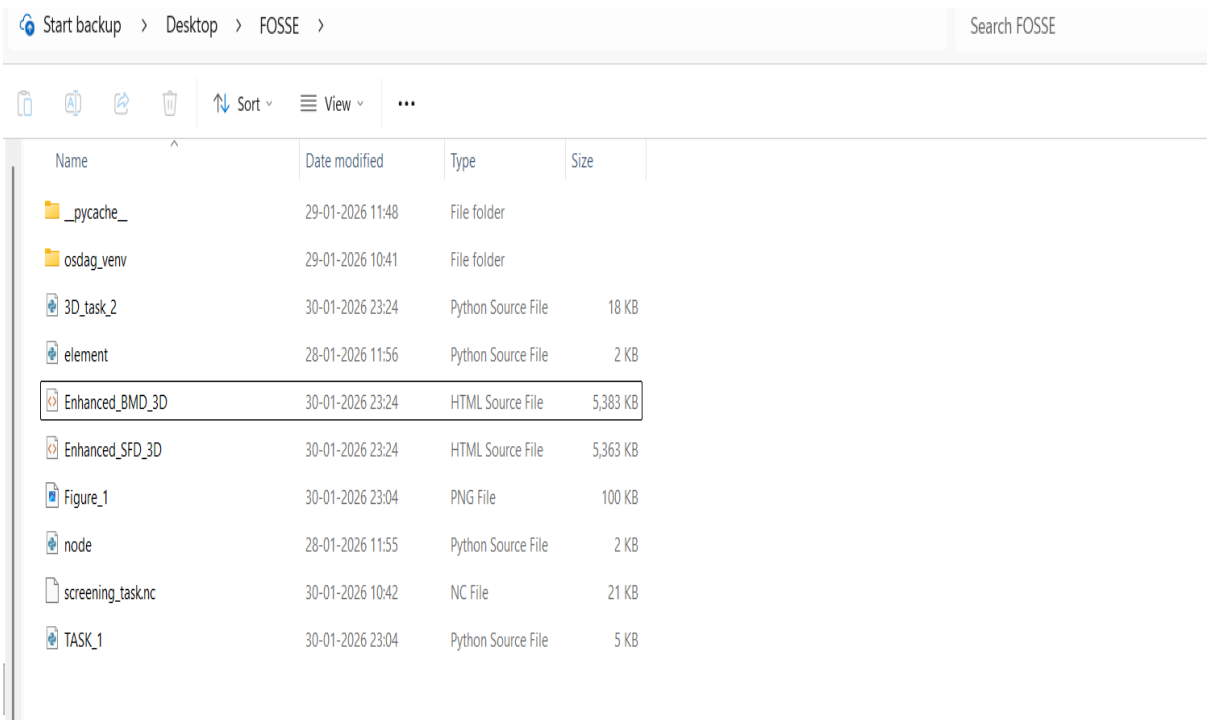

Figure 5 The files are kept in the same folder. An environment is created for better functionality and efficeincy. The html files are saved in the same folder which can be retrieved to interact with the 3D models.

CODE IMPLEMENTATION

1. File header and imports

```
"""
Enhanced Bridge 3D Shear Force and Bending Moment Diagram Generator
MIDAS-style visualization with improved grids, borders, and pastel colors

with Fixed Girder Definitions and Visibility Controls
"""

import numpy as np
import xarray as xr
import plotly.graph_objects as go
from plotly.subplots import make_subplots
from pathlib import Path
```

This block defines the purpose of the script and imports all required libraries.

2. Main 3D diagram generator function

```
def create_enhanced_3d_diagram(nc_file, node_file, element_file,
result_type='BMD', segments=50):
```

This function generates a full interactive 3D SFD or BMD model.

3. Function documentation

```
"""
    Create enhanced 3D diagram with:
    - CAD-style visible grids
    - Girder boundary edges
    - Pastel/muted colors for better visibility
    - Transparent surfaces
    - Proper girder visibility controls

    Parameters:
    -----------
    nc_file : str
        Path to NetCDF dataset
    node_file : str
        Path to node.py file
    element_file : str
        Path to element.py file
    result_type : str
        'BMD' or 'SFD'
    segments : int
        Number of segments per element (50 recommended) "”
```

This explains what the function produces and what each input means.

4. Basic console header

```
    print("\n" + "="*80)
    print(f"ENHANCED 3D {result_type} GENERATOR - Version 2.1")
    print("="*80 + "\n")
```

This prints a simple header for the current run.

5. Load NetCDF, node and element data

```
    print(" Loading data files...")
    ds = xr.open_dataset(nc_file)

    spec = {}
    with open(node_file, 'r') as f:
        exec(f.read(), spec)
    nodes = spec['nodes']

    spec = {}
    with open(element_file, 'r') as f:
        exec(f.read(), spec)
    members = spec['members']

    print(f"✓ Loaded: {len(nodes)} nodes, {len(members)} elements\n")
```

This loads the result file and reads the node and element dictionaries.

6. Exact girder definitions used in the task

```
girders = {
        'Girder 1': {
            'elements': [13, 22, 31, 40, 49, 58, 67, 76, 81],
            'nodes': [1, 11, 16, 21, 26, 31, 36, 41, 46, 6],
            'color': 'rgb(100, 149, 237)'  # Cornflower Blue
        },
        'Girder 2': {
            'elements': [14, 23, 32, 41, 50, 59, 68, 77, 82],
            'nodes': [2, 12, 17, 22, 27, 32, 37, 42, 47, 7],
            'color': 'rgb(127, 255, 212)'  # Aquamarine
        },
        'Girder 3': {
            'elements': [15, 24, 33, 42, 51, 60, 69, 78, 83],
            'nodes': [3, 13, 18, 23, 28, 33, 38, 43, 48, 8],
            'color': 'rgb(144, 238, 144)'  # Light Green
        },
        'Girder 4': {
            'elements': [16, 25, 34, 43, 52, 61, 70, 79, 84],
            'nodes': [4, 14, 19, 24, 29, 34, 39, 44, 49, 9],
            'color': 'rgb(255, 218, 120)'  # Pale Gold
        },
        'Girder 5': {
            'elements': [17, 26, 35, 44, 53, 62, 71, 80, 85],
            'nodes': [5, 15, 20, 25, 30, 35, 40, 45, 50, 10],
            'color': 'rgb(240, 128, 128)'  # Light Coral
        }
```

This defines all five longitudinal girders exactly as per the model.

7. Element-to-girder lookup table

```
element_to_girder = {}
    for girder_name, girder_info in girders.items():
        for eid in girder_info['elements']:
            element_to_girder[eid] = girder_name
```

This allows each element to be mapped to its corresponding girder.

8. Print girder summary

```
print(" Girder Definitions:")
    for gname, ginfo in girders.items():
        print(f"  {gname}: {len(ginfo['elements'])} elements")
    print()
```

This prints how many elements belong to each girder.

9. Select result components (BMD or SFD)

```
if result_type == 'BMD':
        comp_i, comp_j = 'Mz_i', 'Mz_j'
        unit = 'kN·m'
        title = '3D Bending Moment Diagram (BMD) - Enhanced MIDAS Style'
        ylabel = 'Bending Moment'
    else:
        comp_i, comp_j = 'Vy_i', 'Vy_j'
        unit = 'kN'
        title = '3D Shear Force Diagram (SFD) - Enhanced MIDAS Style'
        ylabel = 'Shear Force'
```

This selects which force or moment components will be visualised.

10. Compute scaling factor

```
    print("Calculating scale factor...")
    all_values = []
    for eid in members.keys():
        try:
            val_i = abs(float(ds['forces'].sel(Element=eid,
Component=comp_i).values))
            val_j = abs(float(ds['forces'].sel(Element=eid,
Component=comp_j).values))
            all_values.extend([val_i, val_j])
        except:
            continue

    max_val = max(all_values) if all_values else 1.0
    scale = 1.8 / max_val if max_val > 0 else 1.0

    if result_type == "SFD":
        scale *= 3.0
```

This automatically scales diagram height so the model is visually balanced.

11. Create the Plotly figure

```
fig = go.Figure()
```

This starts the 3D figure.

12. Pastel colour scale

```
colorscale = [
    [0.0, 'rgb(100, 149, 237)'],
    [0.25, 'rgb(127, 255, 212)'],
    [0.5, 'rgb(144, 238, 144)'],
    [0.75, 'rgb(255, 218, 120)'],
    [1.0, 'rgb(240, 128, 128)']
]
```

This defines a soft colour scale for better readability.

13. Storage for girder trace control

```
girder_traces = {gname: [] for gname in girders.keys()}
girder_traces['Other Elements'] = []
```

This keeps track of which traces belong to each girder.

14. Draw the structural centre lines

```
print("Drawing structure centerlines...")
for eid, (ni, nj) in members.items():
    x1, y1, z1 = nodes[ni]
    x2, y2, z2 = nodes[nj]

    girder_name = element_to_girder.get(eid, 'Other Elements')

    trace = go.Scatter3d(
        x=[x1, x2],
        y=[y1, y2],
        z=[z1, z2],
        mode='lines',
        line=dict(
            color='rgba(20, 20, 20, 1.0)',
            width=6
        ),
        showlegend=False,
        hoverinfo='skip',
        name=girder_name,
        legendgroup=girder_name
    )

    fig.add_trace(trace)
    girder_traces[girder_name].append(len(fig.data) - 1)
```

This draws the 3D skeleton of the bridge structure.

15. Create force or moment surfaces for all elements

```
    print(f" Creating {result_type} surfaces with boundary edges...")
    element_count = 0
    shown_in_legend = set()

    for eid, (ni, nj) in members.items():
        girder_name = element_to_girder.get(eid, 'Other Elements')
```

This loop generates surfaces for every element.

16. Geometry, values and mesh generation

```
        try:
            x1, y1, z1 = nodes[ni]
            x2, y2, z2 = nodes[nj]

            v_i = float(ds['forces'].sel(Element=eid,
Component=comp_i).values)
            v_j = float(ds['forces'].sel(Element=eid,
Component=comp_j).values)

            X, Y, Z, C = [], [], [], []

            for i in range(segments + 1):
                t = i / segments

                x = x1 + t * (x2 - x1)
                y0 = y1 + t * (y2 - y1)
                z = z1 + t * (z2 - z1)

                v = v_i + t * (v_j - v_i)

                X.extend([x, x])
                Y.extend([y0, y0 + v * scale])
                Z.extend([z, z])
                C.extend([abs(v), abs(v)])
```

This builds the 3D surface points along each element.

17. Triangular mesh connectivity

```
            I, J, K = [], [], []
            for i in range(segments):
                b = i * 2
                I.extend([b, b + 1])
                J.extend([b + 1, b + 3])
                K.extend([b + 2, b + 2])
```

This defines how the surface triangles are connected.

18. Add the surface mesh

```
            show_in_legend = girder_name not in shown_in_legend
            if show_in_legend:
                shown_in_legend.add(girder_name)

            mesh = go.Mesh3d(
                x=X, y=Y, z=Z,
                i=I, j=J, k=K,

                name=girder_name,
                legendgroup=girder_name,
                showlegend=show_in_legend,

                intensity=C,
                colorscale=colorscale,
                cmin=min(all_values),
                cmax=max(all_values),
                opacity=0.75,
                showscale=(element_count == 0),
                colorbar=dict(
                    title=f"{result_type}<br>({unit})",
                    thickness=20,
                    len=0.7,
                    x=1.02
                ) if element_count == 0 else None,
                hovertemplate=(
                    f"<b>{girder_name} - Element {eid}</b><br>" +
                    f"Nodes: {ni} → {nj}<br>" +
                    f"{comp_i}: {v_i:.3f} {unit}<br>" +
                    f"{comp_j}: {v_j:.3f} {unit}<br>" +
                    "<extra></extra>"
                ),
                flatshading=False,
                lighting=dict(
                    ambient=0.7,
                    diffuse=0.8,
                    specular=0.3,
                    roughness=0.5,
                    fresnel=0.2
                ),
                lightposition=dict(x=1000, y=1000, z=1000)
            )
```

This creates the coloured 3D surface for each element.

19. Add boundary edges for clarity

```
            fig.add_trace(mesh)
            girder_traces[girder_name].append(len(fig.data) - 1)

            edge = go.Scatter3d(
                x=[x1, x1],
                y=[y1, y1 + v_i * scale],
                z=[z1, z1],
                mode='lines',
                line=dict(
                    color='rgba(0, 0, 0, 0.9)',
                    width=4
                ),
                showlegend=False,
```

```
            hoverinfo='skip',
            name=girder_name,
```

```
            legendgroup=girder_name
        )
        fig.add_trace(edge)
```

This draws the vertical edge at the start of the element.

```
        edge = go.Scatter3d(
            x=[x2, x2],
            y=[y2, y2 + v_j * scale],
            z=[z2, z2],
            mode='lines',
            line=dict(
                color='rgba(0, 0, 0, 0.9)',
                width=2
            ),
            showlegend=False,
            hoverinfo='skip',
            name=girder_name,
            legendgroup=girder_name
        )
        fig.add_trace(edge)
```

This draws the vertical edge at the end of the element.

```
        edge = go.Scatter3d(
            x=[x1, x2],
            y=[y1 + v_i * scale, y2 + v_j * scale],
            z=[z1, z2],
            mode='lines',
            line=dict(
                color='rgba(0, 0, 0, 0.9)',
                width=1.5
            ),
            showlegend=False,
            hoverinfo='skip',
            name=girder_name,
            legendgroup=girder_name
        )
        fig.add_trace(edge)
```

This connects the top edges of the surface.

20. Finish surface generation loop

```
        element_count += 1

    except Exception as e:
        print(f" Warning: Could not process element {eid}: {e}")
        continue
```

This safely skips any problematic elements.

21. Apply CAD-style layout and grids

```
    fig.update_layout(
        title={
            'text': title,
            'x': 0.5,
            'xanchor': 'center',
            'font': {'size': 20, 'color': '#2c3e50', 'family': 'Arial,
sans-serif'}
        },
        scene=dict(
            xaxis=dict(
                title=dict(text='X (m)', font=dict(size=14,
color='#34495e')),
                gridcolor='rgb(180, 180, 180)',
                gridwidth=2,
                showgrid=True,
                zeroline=True,
                zerolinecolor='rgb(100, 100, 100)',
                zerolinewidth=3,
                showbackground=True,
                backgroundcolor='rgb(245, 245, 250)',
                showspikes=False,
                dtick=2.5,
                tickfont=dict(size=11)
            ),
            yaxis=dict(
                title=dict(text=f'{ylabel} ({unit})', font=dict(size=14,
color='#34495e')),
                gridcolor='rgb(180, 180, 180)',
                gridwidth=2,
                showgrid=True,
                zeroline=True,
                zerolinecolor='rgb(100, 100, 100)',
                zerolinewidth=3,
                showbackground=True,
                backgroundcolor='rgb(245, 245, 250)',
                showspikes=False,
                tickfont=dict(size=11)
            ),
            zaxis=dict(
                title=dict(text='Z (m)', font=dict(size=14,
color='#34495e')),
                gridcolor='rgb(180, 180, 180)',
                gridwidth=2,
                showgrid=True,
                zeroline=True,
                zerolinecolor='rgb(100, 100, 100)',
                zerolinewidth=3,
                showbackground=True,
                backgroundcolor='rgb(245, 245, 250)',
                showspikes=False,
                dtick=2.0,
                tickfont=dict(size=11)
            ),
            aspectratio=dict(x=2, y=1, z=1),
            camera=dict(
                eye=dict(x=1.8, y=1.5, z=1.2),
                center=dict(x=0, y=0, z=0),
                up=dict(x=0, y=1, z=0)
            ),
            bgcolor='rgb(250, 250, 252)'
```

```
        ),

        width=1600,
        height=900,
        margin=dict(l=0, r=100, b=0, t=80),
        paper_bgcolor='rgb(255, 255, 255)',
        font=dict(family='Arial, sans-serif', size=12, color='#2c3e50'),
        hovermode='closest'
    )
```

This applies the MIDAS-like visual styling and grid appearance.

22. Add girder visibility buttons

```
    buttons = []
    total_traces = len(fig.data)

    buttons.append(dict(
        label="All Girders",
        method="update",
        args=[{"visible": [True] * total_traces}]
    ))
```

This creates a button to show the full structure.

```
    for girder_name in ['Girder 1', 'Girder 2', 'Girder 3', 'Girder 4',
'Girder 5']:
        if girder_name in girder_traces:
            visibility = [False] * total_traces
            for idx in girder_traces[girder_name]:
                visibility[idx] = True

            buttons.append(dict(
                label=girder_name,
                method="update",
                args=[{"visible": visibility}]
            ))
```

This adds one button per girder.

```
    if 'Other Elements' in girder_traces and girder_traces['Other
Elements']:
        visibility = [False] * total_traces
        for idx in girder_traces['Other Elements']:
            visibility[idx] = True

        buttons.append(dict(
            label="Transverse Only",
            method="update",
            args=[{"visible": visibility}]
        ))
```

This adds a button to show only transverse members.

```
    fig.update_layout(
        updatemenus=[
            dict(
                type="buttons",
                direction="down",
                buttons=buttons,
                x=0.01,
                y=0.99,
                xanchor='left',
                yanchor='top',
                showactive=True,
                bgcolor='rgba(255, 255, 255, 0.9)',
                bordercolor='rgba(0, 0, 0, 0.3)',
                borderwidth=1
            )
        ]
    )
```

This places the dropdown control on the figure.

23. Finish the function

```
    print(f"ENHANCED 3D {result_type} COMPLETE")

    return fig
```

This returns the finished 3D figure.

24. Main function

```
def main():
    """Main execution"""
```

This controls the full workflow.

25. Header and input file check

```
    print("\n" + "="*80)
    print("ENHANCED BRIDGE 3D DIAGRAMS ")
    print("Improved Grids | Visible Borders | Pastel Colors | Girder
Controls")
    print("="*80 + "\n")

    nc_file = r'./screening_task.nc'
    node_file = r'./node.py'
    element_file = r'./element.py'

    for f in [nc_file, node_file, element_file]:
        if not Path(f).exists():
            print(f" Error: File not found: {f}")
            return

    print("✓ All input files found\n")
```

This checks that all required input files exist.

26. Generate and save the BMD model

```
    print("="*80)
    print("GENERATING BENDING MOMENT DIAGRAM (BMD)")
    print("="*80)
    bmd_fig = create_enhanced_3d_diagram(nc_file, node_file, element_file,
'BMD', segments=50)

    output_bmd = 'Enhanced_BMD_3D.html'
    bmd_fig.write_html(output_bmd)
    print(f"Saved: {output_bmd}")
    bmd_fig.show()
```

This generates and saves the 3D bending moment diagram.

27. Generate and save the SFD model

```
    print("\n" + "="*80)
    print("GENERATING SHEAR FORCE DIAGRAM (SFD)")
    print("="*80)
    sfd_fig = create_enhanced_3d_diagram(nc_file, node_file, element_file,
'SFD', segments=50)

    output_sfd = 'Enhanced_SFD_3D.html'
    sfd_fig.write_html(output_sfd)
    print(f" Saved: {output_sfd}")
    sfd_fig.show()

    print("ALL ENHANCED DIAGRAMS COMPLETE")
```

This generates and saves the 3D shear force diagram.

28. Script entry guard

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

This ensures the script runs only when executed directly.

6. Results

For Task-1, continuous two-dimensional bending moment and shear force diagrams are successfully generated for the central longitudinal girder.
The diagrams clearly show the variation of bending moment and shear force along the girder length and allow interactive inspection of values. For Task-2, three-dimensional bending moment and shear force diagrams are generated for all five longitudinal girders.
The bridge framing is visible, and the force and moment distributions are represented as vertical extruded surfaces along each girder. Interactive controls allow isolation of individual girders for better interpretation.

7. Discussion

The use of Xarray enables direct and reliable access to the internal force results stored in the NetCDF dataset. By extracting values using the element and component dimensions, the data structure is respected and no manual data rearrangement is required.

The approach used in Task-1 ensures continuity by carefully handling shared nodes between consecutive elements. This avoids artificial jumps in the diagrams.

In Task-2, the use of interpolation and surface generation allows the force and moment distribution to be visualized in a manner similar to commercial structural analysis software. The inclusion of girder-wise visibility controls significantly improves usability and clarity.

All plots strictly follow the sign convention present in the dataset and no sign modification is applied.

8. Conclusion

In this screening task, a complete workflow was developed to extract shear force and bending moment data from an Xarray dataset and visualize them in both two-dimensional and three-dimensional formats.

The first part of the work successfully produces continuous and interactive SFD and BMD plots for the central longitudinal girder. The second part generates MIDAS-style three-dimensional diagrams for all longitudinal girders using the actual bridge geometry and element connectivity.

The developed codes satisfy the requirements of correct Xarray usage, correct element selection, accurate geometric representation and visually clear plotting, and demonstrate a practical approach for post-processing structural analysis results using open-source Python tools.