DEVELOPMENT OF NODE-BASED OPENFOAM CASE GENERATOR IN PYQT FOR OPENFOAM GUI

Navya Sai Sadu[1], **Dr. Chandan Bose**[2], **Mr. Diptangshu Dey**[3]

[1] Undergraduate student, Department of Computer Science and Engineering, Ace Engineering College, Ghatkesar, Telangana, India
[2] Assistant Professor, University of Birmingham
[3] Research Assistant, OpenFOAM GUI, FOSSEE, IIT Bombay

Abstract

The generation of OpenFOAM cases traditionally requires the manual creation and editing of complex text-based configuration files, presenting significant barriers to new users and creating the potential for errors. This project presents the `PyVNT Node Editor`, a professional desktop application that transforms OpenFOAM case creation through an intuitive node-based visual programming interface.
Built using `PyQt6` and implementing a robust Model-View-Controller architecture, the application provides a comprehensive graphical environment where users create OpenFOAM configurations by connecting visual nodes representing parameters, containers, and output files. The system seamlessly integrates with the `PyVNT` library through lazy evaluation optimization, ensuring efficient memory usage and real-time validation of OpenFOAM configurations.
Key features include drag-and-drop node creation, intelligent connection validation, comprehensive case loading capabilities, and optimized file generation workflows. The application supports both individual file generation and complete OpenFOAM directory structure creation while maintaining full compatibility with existing OpenFOAM workflows.
The implementation demonstrates significant improvements in usability and accessibility for OpenFOAM case generation, successfully bridging the gap between complex CFD requirements and user-friendly interfaces. This work establishes a new standard for visual OpenFOAM case creation tools and provides a solid foundation for future computational fluid dynamics software development.

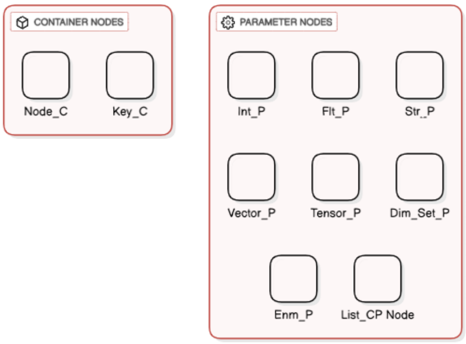
Keywords: OpenFOAM, PyQt6, Node Editor, Visual Programming, CFD, PyVNT

Contents

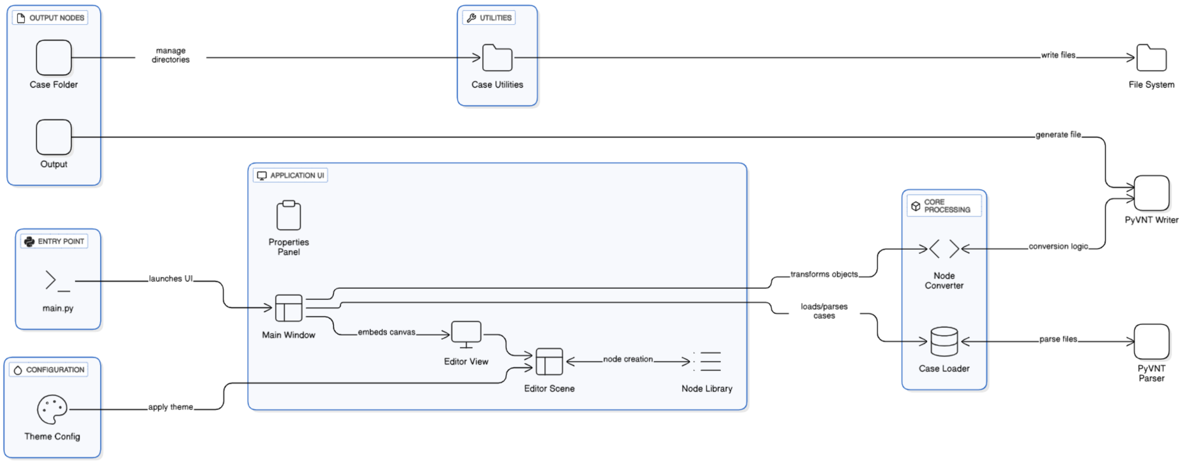
List of Figures

1. Introduction

OpenFOAM is an open-source computational fluid dynamics (CFD) toolbox used in academia and industry to solve continuum mechanics problems. It requires users to manually create and edit complex configuration files in text format. This manual process is time-consuming, error-prone, and presents a significant barrier for new users entering the CFD field.
The `PyVNT Node Editor` addresses these challenges by providing a professional, node-based graphical interface for creating, editing, and generating OpenFOAM case files with the help of `PyQt` Desktop Application. This application represents a significant advancement over previous prototype implementations, offering Visual Programming Interface, Professional Grade Features, Seamless Integration upon `PyVNT` API and Flexible Workflow.
The application follows modern software engineering principles, implementing a clean Model-View-Controller architecture that separates concerns and ensures maintainability. The node-based interface leverages visual programming concepts to make complex CFD configurations accessible to users of all skill levels.



(a) Node Categories and Types



(b) Application Architecture and Data Flow

Figure 1: System Architecture Overview

1.1 Model-View-Controller Design

The `PyVNT Node Editor` implements a sophisticated Model-View-Controller (MVC) architecture that provides clear separation of concerns:

Model Layer (PyVNT Objects):

- `PyVNT` library serves as the underlying data model
- Handles OpenFOAM file parsing and object representation
- Maintains configuration state and validation logic
- Provides API for file generation and case management

View Layer (Graphical Interface):

- `PyQt6`-based user interface components
- Custom graphical nodes for visual representation
- Interactive canvas with zoom, pan, and selection capabilities

- Real-time visual feedback for user operations

Controller Layer (Application Logic):

- Event handling and user interaction management
- Command pattern implementation for undo/redo operations
- Node connection and validation logic
- File I/O operations and case generation coordination

This architecture ensures that changes to the underlying data model automatically propagate to the visual interface, while user interactions are properly validated and processed through the controller layer.

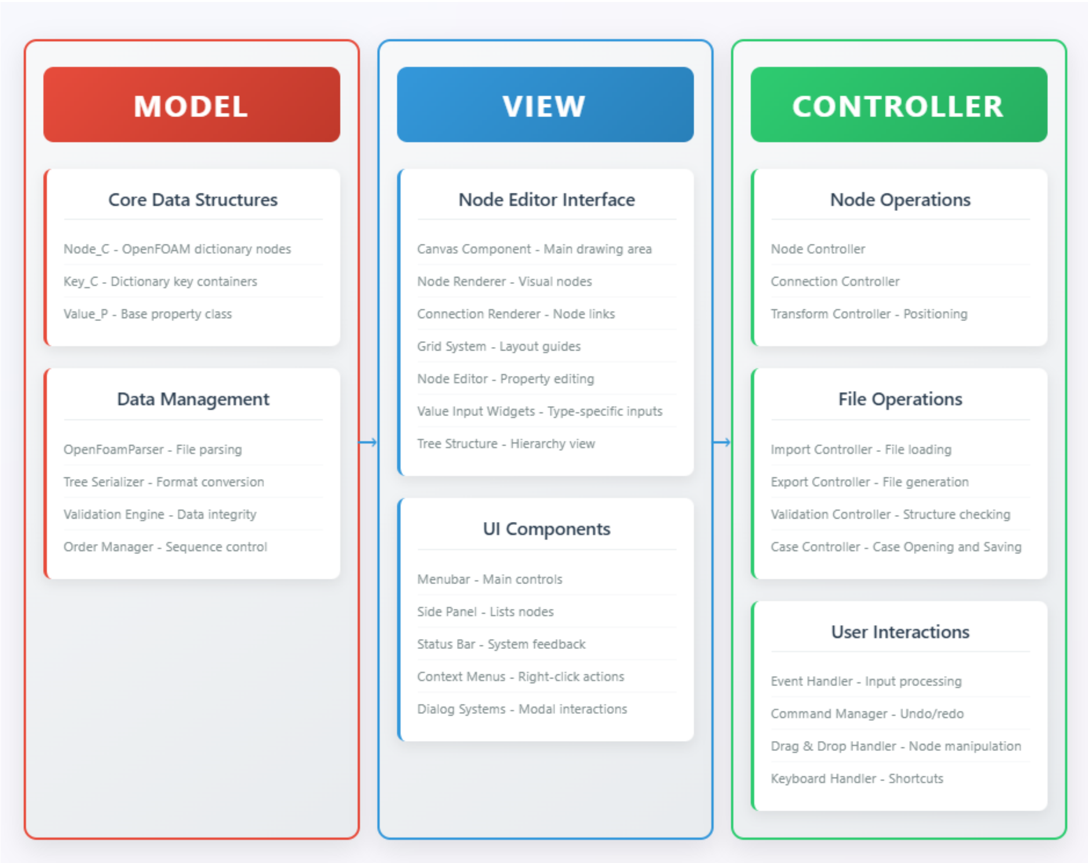


Figure 2: MVC Architecture

1.2 Node-Based Interface Architecture

The application employs a sophisticated node-based interface that translates OpenFOAM concepts into visual programming elements:

Node Categories:

- **Container Nodes**: Represent OpenFOAM dictionaries and structural elements
- **Parameter Nodes**: Handle specific data types (integers, floats, strings, vectors, tensors)
- **Output Nodes**: Manage file generation and case assembly

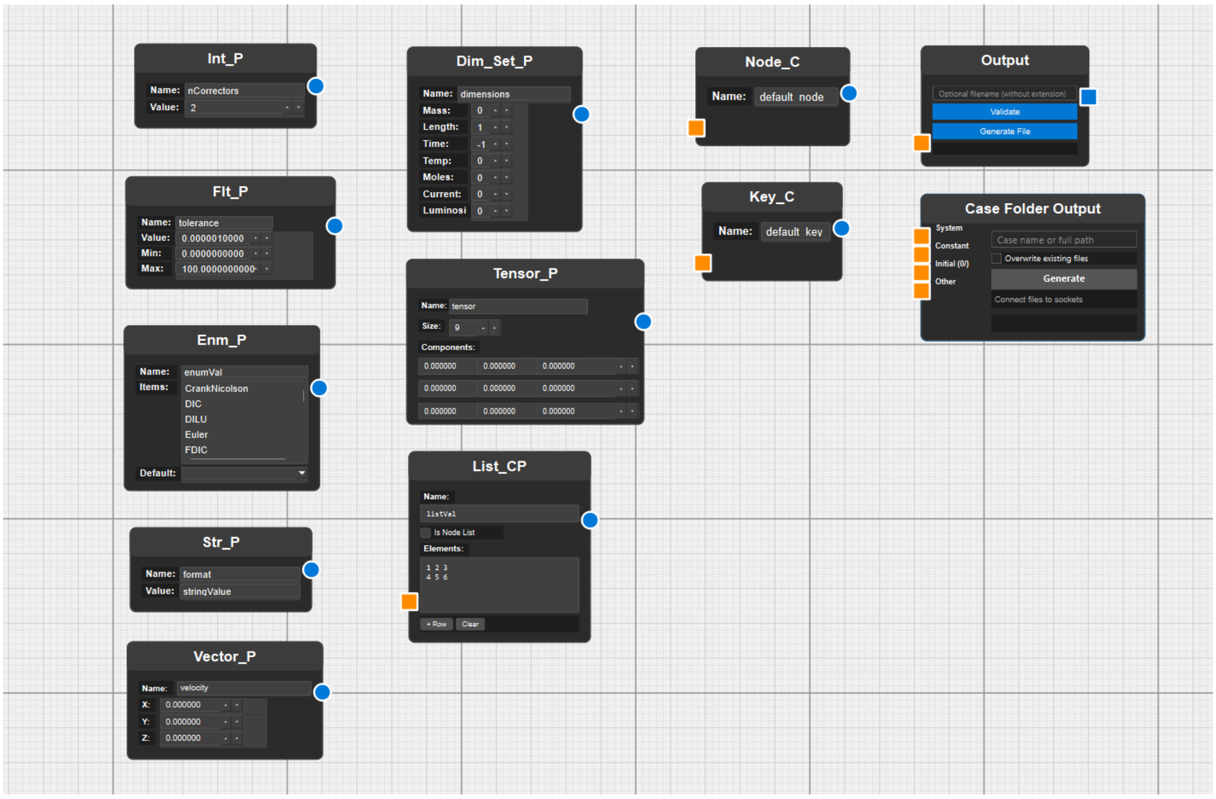


Figure 3: Nodes

Connection System:

- Socket-based communication between nodes
- Type-safe connections with validation
- Visual feedback for connection states
- Automatic routing algorithms for edge visualization

Data Flow:

- Unidirectional data flow from parameter nodes to container nodes to output nodes
- Real-time validation of node connections
- Efficient propagation of changes through the node graph

1.3 PyVNT Integration Layer

The application integrates seamlessly with the `PyVNT` library through a dedicated abstraction layer:
Parser Integration:

- Automatic conversion of OpenFOAM files to node structures
- Support for complex dictionary hierarchies
- Preservation of OpenFOAM syntax and formatting requirements

Object Mapping:

- Direct mapping between graphical nodes and `PyVNT` objects through `getPyVNTObject()` method
- Bidirectional synchronization of data with lazy evaluation optimization
- Validation of object integrity during editing
- On-demand object construction to minimize memory footprint

Generation Pipeline:

- Optimized workflow using lazy evaluation to eliminate duplicate file generation
- `PyVNT` object construction through `getPyVNTObject()` method calls
- Intelligent categorization of files into OpenFOAM directory structures
- Comprehensive error handling and status reporting
- Deferred object creation until actual file generation is required

2. Implementation Details

2.1 Core Application Framework

The application is built on a robust `PyQt6` framework with the following key components:
Main Window (`main_window.py`):

- Central application orchestration
- Menu system and toolbar management
- Status bar and progress indication
- Window state management and persistence

Editor Scene (`editor_scene.py`):

- Grid-based canvas for node placement
- Scene management and coordinate system

- Background rendering and visual feedback
- Node and edge lifecycle management

Editor View (`editor_view.py`):

- Interactive viewport with zoom and pan capabilities
- Mouse and keyboard event handling
- Selection and manipulation tools
- View state persistence and restoration

2.2 Node System Implementation

The node system is implemented through a hierarchical class structure based on the base graphical node class:

```python
class BaseGraphicalNode:
    def __init__(self):
        self.scene = None
        self.sockets = []
        self.input_sockets = []
        self.output_sockets = []
        self.edges = []

    def get_pyvnt_object(self):
        """Override in subclasses to return PyVNT object"""
        raise NotImplementedError("Subclasses must implement")

    def add_socket(self, socket_type, data_type):
        """Add input/output socket for connections"""
        socket = Socket(self, socket_type, data_type)
        self.sockets.append(socket)
        return socket

    def serialize(self):
        """Serialize node data for save/load"""
        return {
            'type': self.__class__.__name__,
            'position': [self.pos().x(), self.pos().y()],
            'properties': self.get_properties()
        }
```

Listing 1: Base Graphical Node Structure

Specialized Node Types:

- Container Nodes: `Node_C` (dictionary containers), `Key_C` (key-value pairs)
- Parameter Nodes: `Int_P`, `Flt_P`, `Str_P`, `Vector_P`, `Tensor_P`, `Dim_Set_P`, `Enm_P`, `List_CP`
- Output Nodes: `Output` (file generation), `Case Folder` (directory structure)

2.3 Socket and Edge System

The connection system uses sockets and edges to enable data flow between nodes:

```
class Socket:
    def __init__(self, node, socket_type, data_type, position=0):
        self.node = node
        self.socket_type = socket_type  # INPUT or OUTPUT
        self.data_type = data_type
        self.position = position
        self.edges = []

    def can_connect_to(self, other_socket):
        """Check if connection is valid"""
        if self.socket_type == other_socket.socket_type:
            return False
        return self.data_type.is_compatible(other_socket.data_type)

    def connect_to(self, other_socket):
        """Create edge connection"""
        if self.can_connect_to(other_socket):
            edge = Edge(self, other_socket)
            self.edges.append(edge)
            other_socket.edges.append(edge)
            return edge
        return None
```

Listing 2: Socket Implementation

```
class Edge:
    def __init__(self, start_socket, end_socket):
        self.start_socket = start_socket
        self.end_socket = end_socket
        self.scene = start_socket.node.scene

    def update_positions(self):
        """Update edge visual representation"""
        start_pos = self.start_socket.get_scene_position()
        end_pos = self.end_socket.get_scene_position()
        self.update_path(start_pos, end_pos)

    def remove(self):
        """Clean up edge connections"""
        self.start_socket.edges.remove(self)
        self.end_socket.edges.remove(self)
        if self.scene:
            self.scene.removeItem(self)
```

Listing 3: Edge Connection System

2.4 Case Loading and Parsing

The application provides comprehensive support for loading existing OpenFOAM cases:

```
class CaseLoader:
    def __init__(self, parser):
        self.parser = parser
        self.node_converter = NodeConverter()

```

```python
    def load_case_directory(self, case_path):
        """Load complete OpenFOAM case directory"""
        case_files = self._discover_case_files(case_path)
        nodes = []

        for file_path in case_files:
            try:
                # Parse using PyVNT
                pyvnt_tree = self.parser.parse_file(file_path)

                # Convert to visual nodes
                visual_nodes = self.node_converter.convert_tree(pyvnt_tree)
                nodes.extend(visual_nodes)

            except Exception as e:
                print(f"Failed to load {file_path}: {e}")

        return nodes

    def _discover_case_files(self, case_path):
        """Find OpenFOAM files in case directory"""
        foam_files = []
        for root, dirs, files in os.walk(case_path):
            for file in files:
                if self._is_openfoam_file(file):
                    foam_files.append(os.path.join(root, file))
        return foam_files
```

Listing 4: Case Loader Implementation

Loading Process:

1. File system scanning and OpenFOAM structure detection
2. Progressive parsing with status indication
3. Object graph construction and validation
4. Node creation and automatic layout
5. Connection establishment and verification

2.5 Output Generation System

The application implements an optimized output generation system:

```python
class OutputNode(BaseGraphicalNode):
    def __init__(self):
        super().__init__()
        self.output_path = ""
        self.add_input_socket("data", "PyVNTObject")

    def generate_files(self):
        """Generate OpenFOAM files from connected nodes"""
        try:
            # Validation phase
            connected_objects = self._get_connected_objects()

            # Object construction phase
```

```python
            pyvnt_objects = []
            for node in connected_objects:
                pyvnt_obj = node.get_pyvnt_object()
                if pyvnt_obj:
                    pyvnt_objects.append(pyvnt_obj)

            # File generation phase
            for obj in pyvnt_objects:
                file_path = os.path.join(self.output_path, obj.name)
                obj.write_to_file(file_path)

            return f"Generated {len(pyvnt_objects)} files successfully"

        except Exception as e:
            return f"Generation failed: {str(e)}"

    def _get_connected_objects(self):
        """Get all nodes connected to input sockets"""
        connected_nodes = []
        for socket in self.input_sockets:
            for edge in socket.edges:
                source_node = edge.start_socket.node
                connected_nodes.append(source_node)
        return connected_nodes
```

Listing 5: Output Node Generation

2.6 PyVNT Integration and Lazy Evaluation

The application implements sophisticated `PyVNT` integration with performance optimized lazy evaluation:

Lazy Evaluation System:

- On-Demand Object Creation: `PyVNT` objects are created only when needed through `getPyVNTObject()` method calls
- Memory Optimization: Reduces memory footprint by avoiding premature object instantiation
- Performance Enhancement: Minimizes computational overhead during interactive editing
- Dependency Tracking: Maintains dependency graphs to determine when objects need regeneration

```python
class BaseGraphicalNode:
    def __init__(self):
        self._pyvnt_object = None
        self._needs_rebuild = True
        self.dependencies = []

    def get_pyvnt_object(self):
        """Lazy evaluation of PyVNT objects"""
        if self._needs_rebuild or self._pyvnt_object is None:
            self._pyvnt_object = self._build_pyvnt_object()
            self._needs_rebuild = False
        return self._pyvnt_object
```

```python

    def _build_pyvnt_object(self):
        """Override in subclasses"""
        raise NotImplementedError("Subclasses must implement")

    def mark_dirty(self):
        """Mark object as needing rebuild"""
        self._needs_rebuild = True
        for dependent in self.dependents:
            dependent.mark_dirty()
```

Listing 6: Lazy Evaluation Implementation

Integration Benefits:

- Seamless Workflow: Users work with visual nodes while `PyVNT` handles OpenFOAM specifics
- Type Safety: `PyVNT` validation ensures generated files conform to OpenFOAM standards
- Performance: Lazy evaluation prevents unnecessary object creation during editing
- Consistency: All nodes use standardized `PyVNT` object interface

3. User Interaction and Workflow

3.1 Visual Programming Interface

The application provides an intuitive visual programming environment:
Node Creation:

- Drag nodes from the library panel to the canvas
- Automatic placement and alignment assistance
- Context-sensitive node suggestions
- Duplicate detection and prevention

Node Connection:

- Click and drag from output sockets to input sockets
- Visual connection preview during dragging
- Type compatibility validation
- Automatic connection routing and optimization

Node Configuration:

- In-place parameter editing
- Property panels for advanced configuration
- Real-time validation feedback
- Context-sensitive help and documentation

3.2 File Generation Process

The application implements a streamlined file generation workflow:
Phase 1: Node Graph Construction

- Users create and connect nodes to define case structure
- Real-time validation ensures correctness
- Visual feedback indicates connection status

Phase 2: Validation and Optimization

- System validates complete node graph through recursive `getPyVNTObject()` calls
- Identifies missing connections or invalid configurations using `PyVNT` validation
- Optimizes object hierarchy for efficient generation with lazy evaluation
- Caches validated `PyVNT` objects to avoid redundant computation

Phase 3: Output Generation

- Generates files based on validated `PyVNT` object structure
- Uses `PyVNT` serialization for proper OpenFOAM format compliance
- Provides progress indication and status updates through `PyVNT` callbacks
- Reports generation results and file locations with `PyVNT` object validation status

3.3 Case Management

The application supports comprehensive case management features including case loading, editing, and export capabilities with full `PyVNT` integration for maintaining OpenFOAM compatibility.

4. Results and Demonstrations

The application successfully generates standard OpenFOAM case files:

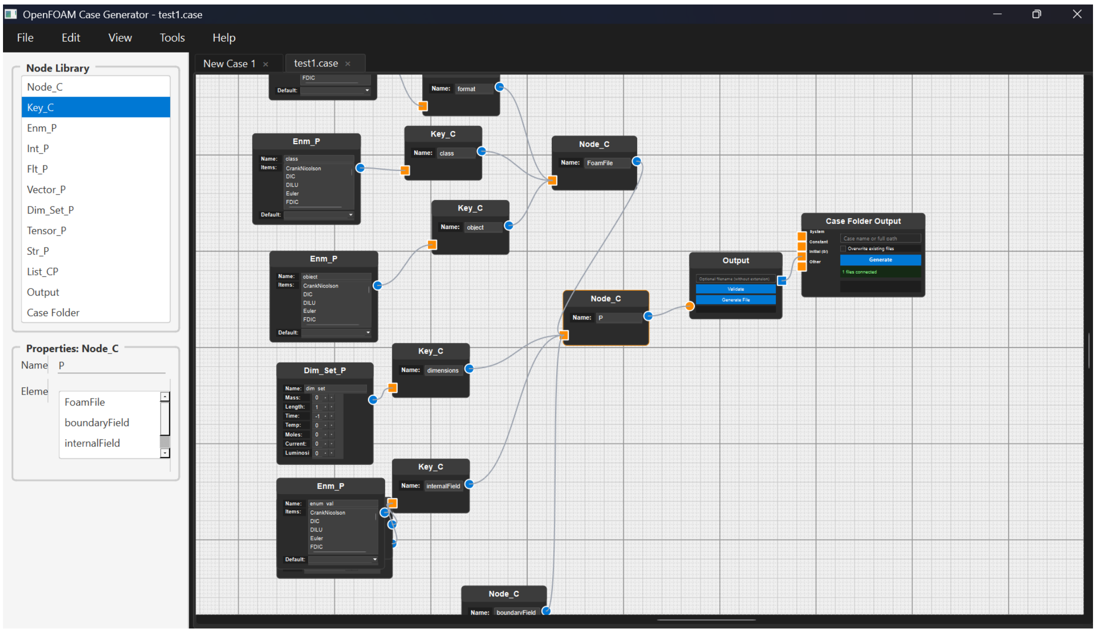


Figure 4: p file

The `PyVNT Node Editor` demonstrates:

- Successful parsing and visualization of OpenFOAM case files
- Intuitive node-based interface for case modification
- Reliable file generation with proper OpenFOAM formatting
- Efficient memory management through lazy evaluation
- Comprehensive validation and error reporting

```
FoamFile
{
   format          ascii;

   class           volScalarField;

   object          p;
}

dimensions      [0 1 -1 0 0 0 0];

internalField   uniform 1e-06;

boundaryField
{
   movingWall
   {
       type        zeroGradient;
   }

   fixedWalls
   {
       type        zeroGradient;
   }

   frontAndBack
   {
       type        empty;
   }
}
```

Listing 7: Generated p file

5. Conclusion

The `PyVNT Node Editor` represents a significant advancement in OpenFOAM case generation tools. By providing a professional, node-based interface, the application successfully addresses the primary challenges faced by OpenFOAM users.
It demonstrates that sophisticated graphical interfaces can significantly improve the usability of complex engineering software while maintaining full compatibility with existing workflows. This project establishes a standard for OpenFOAM case generation tools and provides a solid foundation for the computational fluid dynamics software.

Acknowledgements

I would like to express my sincere gratitude to **Dr. Chandan Bose**, **Mr. Diptangshu Dey**, and the entire **FOSSEE team at IIT Bombay** for their continuous support and guidance in the development of the `PyVNT Node Editor` project. Their expertise in software development and computational fluid dynamics has been instrumental in creating this professional tool for OpenFOAM case generation.

I also acknowledge the contributions of previous fellowship participants who laid the foundation for this project through their work on the initial **OpenFOAM GUI** prototypes and API development.

This project represents a significant evolution from the initial proof-of-concept implementations to a fully functional, professional-grade application that addresses the real-world needs of OpenFOAM users.

References

1. The OpenFOAM Foundation https://openfoam.org/
2. Riverbank Computing. *Reference Guide PyQt Documentation v6.5.1*. https://www.riverbankcomputing.com/static/Docs/PyQt6/
3. OpenFOAM v9 User Guide - 4.2 Basic input/output file format https://doc.cfd.direct/openfoam/user-guide-v9/basic-file-format
4. Node Editor in Python using PyQt5 https://gitlab.com/pavel.krupala/pyqt-node-editor
5. PyVNT Repository's used branch https://github.com/FOSSEE/pyvnt/tree/YashSuthar_sem_intern_2025