FOSSEE, IIT Bombay
OpenFOAM Case Study Project
May 15, 2025

# Development of Parser for PyVnt and Visual Representation of PyVnt Node in Blender

Yash Suthar
B.E - Computer Science and Engineering
Dayananda Sagar College of Engineering,bangalore Karnataka

# Abstract

OpenFOAM [1]. is a free and open-source C++ toolbox designed for solving continuum mechanics problems. While powerful, it requires users to manually write and configure case files in a structured text format. This process can be cumbersome and intimidating, especially for beginners or those unfamiliar with OpenFOAM's internal file structure. A Graphical User Interface (GUI) can significantly improve the user experience by making the software more intuitive and accessible.

The OpenFOAM GUI Project, under the FOSSEE initiative at IIT Bombay, aims to streamline and simplify the case setup process through the development of GUI tools. PyVNT (Python Venturial Node Trees), was developed to act as an intermediary layer between OpenFOAM and a VenturialGUI. This module is capable of both generating OpenFOAM case files from GUI-provided input and parsing existing case files (both traditional OpenFOAM dictionaries and YAML formats) to visualize their structure as a node tree. As part of this project, a robust parser within PyVNT was developed to interpret OpenFOAM configuration files and extract their hierarchical structure. This parsed data serves as the foundation for graphical representation within a node-based interface, visualized using a custom node system in Blender.

# Acknowledgement

I would like to extend my deepest and most sincere gratitude to **Prof. Chandan Bose**, **Mr. Diptangshu Dey**, and **Mr. Rajdeep Adak** for their exceptional mentorship, insightful guidance, and unwavering support throughout the duration of my semester-long internship under the *OpenFOAM GUI Project* at **FOSSEE, IIT Bombay**. Their extensive expertise in *computational fluid dynamics* as well as their dedication to academic and professional excellence, have been immensely influential in shaping the direction and quality of my work during this internship.

Throughout this period, I greatly benefited from their thoughtful feedback, patient explanations, and their willingness to engage in detailed technical discussions. Each interaction provided valuable learning opportunities that not only broadened my knowledge base, but also challenged me to approach complex problems with analytical rigor and precision.

I am particularly grateful for the environment they cultivated, one that encourages curiosity, critical thinking, and independent exploration. Their mentorship has not only improved my technical competencies, but also significantly contributed to my growth as a learner and aspiring professional in the field.

In addition, I would like to acknowledge the contributions and support of the entire team at **FOSSEE, IIT Bombay**. Their welcoming attitude, consistent support, and collaborative spirit created a productive and inspiring work environment. It was truly an honor and a privilege to be part of such a dynamic and talented group of individuals who are dedicated to open source development and educational empowerment.

This internship experience has been profoundly enriching, both technically and personally, and I remain deeply appreciative of the opportunity to contribute to a project of such significance and impact.

# Contents

# 1 Introduction

The OpenFOAM GUI Project, under the FOSSEE initiative at IIT Bombay, aims to streamline and simplify the case setup process through the development of GUI tools. Most existing third-party GUI solutions for OpenFOAM either offer limited capabilities or fail to cover the complete solution workflow required for CFD simulations. To overcome this, a custom Python-based module, **PyVNT (Python Venturial Node Trees)** was developed to act as an intermediary layer between OpenFOAM and a VenturialGUI. This module is capable of both generating OpenFOAM case files from GUI-provided input and parsing existing case files (both traditional OpenFOAM dictionaries and YAML formats) to visualize their structure as a node tree.

As part of this project, a robust parser within PyVNT was developed to interpret OpenFOAM configuration files and extract their hierarchical structure. This parser reads keyword entries, nested dictionaries, and lists from OpenFOAM's text-based input files and YAML files, converting them into a structured, tree-like format defined by PyVNT classes. This parsed data serves as the foundation for graphical representation within a node-based interface.

To visualize this data interactively, a custom tree-node system was designed and implemented within Blender using its built-in scripting capabilities through the bpy Python module. Blender, a widely used open-source 3D creation suite, offers a powerful node-based environment ideal for representing hierarchical data. Tree-Nodes allow users to construct, edit, and understand OpenFOAM simulation setups in a modular and intuitive way, effectively bridging the gap between raw configuration files and a user-friendly graphical interface. PyVNT serves as the backend data model for these Tree Nodes.

# 2 System Design

## 2.1 Architecture Overview

The system developed during this internship is designed to bridge the gap between OpenFOAM's text-based case files and a graphical based editing, interactive representation in Blender. The architecture is modular, consisting of three primary components:

1. **Python-Based Parser (within PyVNT)**
   This module, part of the PyVNT library, is responsible for reading and interpreting OpenFOAM dictionary files (e.g., `fvSolution`, `controlDict`, `blockMeshDict`) as well as OpenFOAM-like YAML configuration files. It recursively parses nested key-word-entries structures and transforms them into a hierarchical data structure (tree) in memory using PyVNT's defined classes.

2. **Data Abstraction Layer (PyVNT Tree Node Structure)**
   After parsing, the extracted data is organized into a custom tree structure defined by PyVNT. Key classes include `Node_C` (for dictionaries/blocks), `Key_C` (for key-word entries), and various `Value_P` derivatives (for data types like strings, integers, floats, and lists). Each node in this structure corresponds to an OpenFOAM dictionary entry. This abstraction layer allows the system to manipulate and represent data generically.

3. **Blender Addon (Tree-Node)**
   A custom Blender Addon built with Python using Blender's `bpy` API. It reads the PyVNT tree structure and dynamically generates node graphs inside Blender's Node Editor. Each PyVNT node is represented by a Venturial Node, allowing users to interact with these nodes ,modify values, connecting entries, or exporting the final configuration back to OpenFOAM dictionary or YAML format using PyVNT's serialization capabilities.

The parser and data structures within PyVNT have been thoughtfully designed with reusability and scalability in mind. Its modular structure allows it to be easily integrated into other front-end systems beyond Blender. This makes PyVNT a core component of the broader Venturial GUI framework, enabling future expansion and adaptation to various user interfaces or platforms that require graphical interaction with OpenFOAM's case files.

## 2.2 Data Flow Pipeline: From Text Files to Blender Nodes

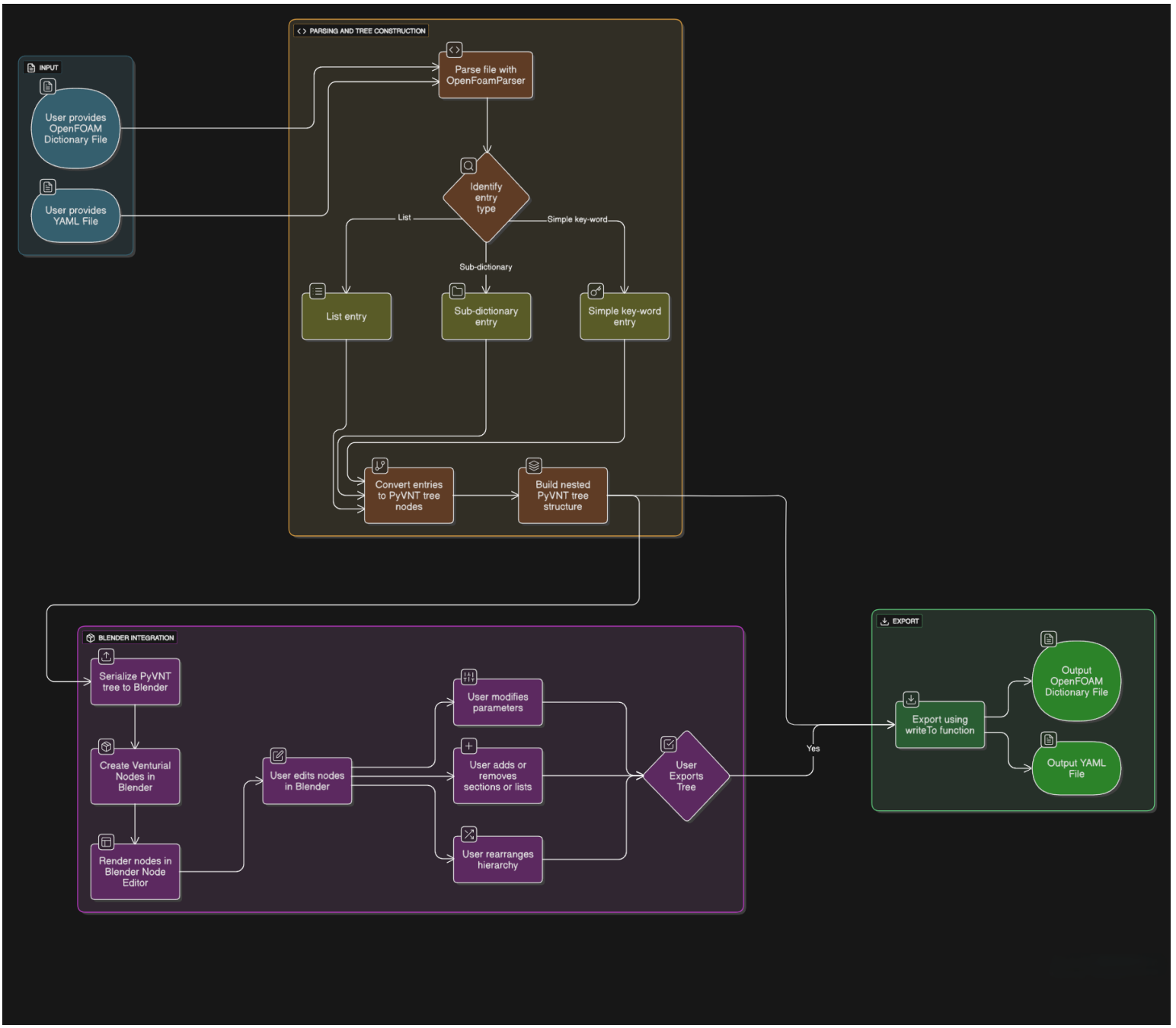


Figure 1: Data flow pipeline.

1. **Input: OpenFOAM Dictionary File or YAML File**
   The user provides a case file (e.g., `blockMeshDict` or `blockMeshDict.yaml`) written in OpenFOAM's custom text format or a compatible YAML structure. This file contains a hierarchy of dictionaries, parameters, and lists.

2. **Parsing the File with PyVNT's `OpenFoamParser`**
   The `OpenFoamParser` module within PyVNT processes the file. It identifies:

   - key-word entries (e.g., `solver PBiCG;`)
   - nested-dictionaries (e.g., `SIMPLE { ... }`)
   - Lists (e.g., `vertices ((0 1 0) (0 1 0.5 ));` or YAML sequences)

   Each entry is converted into PyVNT's internal tree node structure.

3. **Building PyVNT Tree Nodes**
   The parser organizes the content into a nested tree format using PyVNT classes like `Node_C`, `Key_C`, and `Value_P` derivatives. Each dictionary or parameter becomes a node, and its contents are added as children. This tree preserves the original file's logical structure.

4. **Serializing Tree to Blender**
   The PyVNT tree is passed to the Blender Addon, which interprets each PyVNT node and creates a corresponding Venturial Node in the Blender Node Editor.

5. **Rendering in Node Editor**
   Inside Blender, each dictionary or parameter appears as a node block. Nodes can be connected visually to reflect the original structure. Users can modify values directly from the UI.

6. **Export using PyVNT's `writeTo` function**
   After the user edits the setup, the modified PyVNT tree structure can be converted back to OpenFOAM dictionary syntax (`.txt`) or YAML format (`.yaml`) using PyVNT's `writeTo` function, ready for use in simulations.

# 3 Parser Implementation within PyVNT

## 3.1 Design Goals

OpenFOAM files, such as `fvSolution`, `controlDict`, `blockMeshDict` etc., use a custom nested syntax that makes manual editing error-prone especially for new users. The goal was to extract their structure and convert them into a **tree-based data representation using PyVNT**, which could then be used in any GUI framework to simplify the process of configuring OpenFOAM simulations.

The parser, as part of PyVNT, was expected to:

- Read and understand the nested syntax of OpenFOAM dictionaries and equivalent YAML structures.

- Build a PyVNT tree structure that accurately represents the hierarchy of dictionaries, lists, and key-value pairs.
- Interface with a GUI layer (like Blender's Tree-Node) to allow visualization and editing.

A key aspect of this work involved building a custom Tree-Nodes system inside Blender. This system utilizes the **PyVNT** node tree abstraction as its backend. These Tree-Nodes offer a visual representation of the structure generated by the parser, allowing users to interact with OpenFOAM files in a modular and graphical manner directly within Blender.

Thus, the final deliverables included:

- A robust parser that translates OpenFOAM and YAML files into PyVNT tree structures.
- A Tree-Node system integrated in Venturial, to display and interact with these trees.

## 3.2 Parsing Strategy and Structure

The parser, `OpenFoamParser` in PyVNT, is written in Python. For traditional OpenFOAM dictionary files, it follows a recursive descent parsing approach. It reads files line by line, identifying structural elements based on symbols like `{}`, `;`, and whitespace. For YAML files, it leverages a standard YAML parsing library and then maps the resulting Python data structures (dictionaries and lists) to the PyVNT class structure (such as `Node_C`, `Key_C`, and various `Value_P` derivatives).

The parsing process classifies content into:

```
#solver PBiCG;
def p_statement(self,p):
 '''statement : WORD anylist SEMICOLON'''
```

Listing 1: Parser Statement Function

```
'''
SIMPLE
{
    nNonOrthogonalCorrectors 1;
}
'''
def p_dictionary(self,p):
     '''dictionary : WORD LBRACE blocks RBRACE'''
```

Listing 2: Parser Dictionary Function

```
'''
vertices
(
    (0 0 0)(1 0 0.1)(1 1 0.1)(0 1 0.1)
);
edges
```

```
(
    arc 0 1 (0.5 0.1 0)
    spline 4 5
    (
        (4.1 4.2 4.3)
        (4.5 4.6 4.7)
        (4.9 5.0 5.1)
    )
    polyLine 6 7
    (
        (6.1 6.2 6.3)
        (6.5 6.6 6.7)
    )
);
'''
def p_listblock(self,p):    #isNode=False
     '''listblock : WORD LPAREN blocks RPAREN SEMICOLON'''
```

Listing 3: Parser Listblock Function

```
'''
boundary
(
    movingWall
    {
        type wall;
        faces
        (
            (3 7 6 2)
        );
    }
    fixedWalls
    {
        type wall;
        faces
        (
            (0 4 7 3)
            (2 6 5 1)
            (1 5 4 0)
        );
    }
    frontAndBack
    {
        type empty;
        faces ((0 3 2 1)(4 5 6 7));
    }
);
'''
def p_listblock(self,p):    #isNode=True
     '''listblock : WORD LPAREN blocks RPAREN SEMICOLON'''
```

Listing 4: Parser Listblock(isNode=True) Function

The parsing logic for dictionary files tokenizes the input into a structured hierarchy. For each block or entry, the parser creates corresponding PyVNT objects (`Node_C`, `Key_C`, etc.). This hierarchical representation allows the system to accurately mirror the original file's organization. Each PyVNT object stores information such as its name, associated value(s), its type (implicitly defined by the PyVNT class), and any children for nested structures. These PyVNT objects form the tree structure that serves as the backend for GUI components or can be manipulated directly in Python scripts.

## 3.3 Controlling Element Order for File Output

For OpenFOAM dictionary files (`.txt` format), the order of entries can be significant. PyVNT's `Node_C` objects maintain an ordered list of their elements (child `Key_C`s and `Node_C`s). This order is respected by display utilities like `show_tree` and, critically, by the `writeTo` function when generating `.txt` files.

- `set_order(names_list: list)`: This `Node_C` method allows users to explicitly define the order of its direct children (keys and nested nodes) by providing a list of their names. See Figure **??** for an example.
- `get_ordered_items() -> list`: This `Node_C` method returns items in the order they will be written out for `.txt` files.

```
from pyvnt import Node_C, Key_C, Enm_P, List_CP, writeTo, show_tree
import os
import shutil

# Create a directory for output
output_dir = "pyvnt_output_example"

if os.path.exists(output_dir):
        shutil.rmtree(output_dir) # Clean previous run
os.makedirs(output_dir)


root_node = Node_C("myConfig") # Filename will be myConfig.txt

key_c = Key_C("C_setting", Enm_P("val_c", {"val_c"}, default="val_c")
    )
key_a = Key_C("A_setting", Enm_P("val_a", {"val_a"}, default="val_a")
    )
child_b_node = Node_C("B_child_node",parent=root_node)
key_d = Key_C("D_setting", Enm_P("val_d", {"val_d"}, default="val_d")
    )

list_key = Key_C("E_list_key")
list_cp_val = List_CP("my_list_data", elems=[[Enm_P("item1", {"item1"
    },default="item1")],[Enm_P("item2", {"item2"}, default="item2")]])

list_key.append_val(list_cp_val._Value_P__name,list_cp_val)
# Add in an order different from desired final output
```

```
root_node.add_data(key_c)
root_node.add_data(key_a)
root_node.add_data(key_d)
root_node.add_data(list_key)

# --- Write to file BEFORE setting a specific order ---
print(f"--- Writing to {root_node.name}.txt (default order) ---")
writeTo(root_node, output_dir, fileType='txt')
show_tree(root_node)

# --- Set a specific order for root_node's items ---
desired_output_order = ["A_setting", "B_child_node", "C_setting", "E_list_key", "D_setting"]
root_node.set_order(desired_output_order)

print(f"\\n--- Writing to {root_node.name}_ordered.txt (custom order) ---")
# To avoid overwriting, let's change the root_node name for the new file
original_name = root_node.name
root_node.name = original_name + "_ordered"
show_tree(root_node)
writeTo(root_node, output_dir, fileType='txt')
```

Listing 5: `set_order` function to arrange connected children or data in a specific order

The output files generated before and after using `set_order` would differ in the arrangement of their content, demonstrating the control PyVNT provides over file layout.

**Code Reordering Example**

**Before**

```
B_child_node
{
}
C_setting       val_c;
A_setting       val_a;
D_setting       val_d;
E_list_key
(
    item1
    item2
);
```

**After**

```
A_setting       val_a;
B_child_node
{
}
C_setting       val_c;
E_list_key
(
    item1
    item2
);
D_setting       val_d;
```

## 3.4 Serializing PyVNT Trees to Files (`writeTo`)

PyVNT provides a `writeTo(root_node, path, fileType='txt')` function to serialize a node tree back into a file.

- `root_node`: The top-level `Node_C` object. Its name is used for the output filename (e.g., `controlDict.txt`).

- `path`: The directory to save the file.
- `fileType`: Output format.
  - `'txt'` (default): Traditional OpenFOAM dictionary format. Uses `get_ordered_items()`, respecting any custom order.
  - `'yaml'`: YAML representation. YAML library dict key order (often preserved from Python 3.7+ dicts) and `List_CP` order are maintained.

```
writeTo(NodeHead, path=r"\output", fileType='txt')
# OR
writeTo(NodeHead, path=r"\output", fileType='yaml')
```

Listing 6: `writeTo` function to export data to `txt` or `yaml` format

## 3.5 Tree Representation of Dictionaries

After parsing, the configuration file is transformed into a tree-like structure using PyVNT's classes (`Node_C`, `Key_C`, `Value_P` derivatives). This structure consists of nodes and branches where:

- Each `Node_C` represents a dictionary block, and each `Key_C` represents a key with its value(s).
- Leaf nodes within the conceptual tree are typically the `Value_P` instances (strings, numbers, simple lists).
- Branches denote hierarchy, with `Node_C` objects containing other `Node_C`s and `Key_C`s.

For example, `blockMeshDict` is transformed into a tree like:

```
blockMeshDict_from_yaml
{
   convertToMeters : 0.1
   vertices : ((0, 0, 0), (1, 0, 0), (1, 1, 0), (0, 1, 0), (0, 0, 0.1),
       (1, 0, 0.1), (1, 1, 0.1), (0, 1, 0.1))
   blocks : ('hex', (0, 1, 2, 3, 4, 5, 6, 7), (20, 20, 1), '
      simpleGrading', (1, 1, 1))
   edges : ('arc', 0, 1, (0.5, 0.1, 0), 'spline', 4, 5, ('(4.1', 4.2, '
      4.3)', '(4.5', 4.6, '4.7)', '(4.9', 5.0, '5.1)'), 'polyLine', 6,
      7, ('(6.1', 6.2, '6.3)', '(6.5', 6.6, '6.7)'))
   mergePatchPairs : (('cyclicPair1A', 'cyclicPair1B'), ('anotherPatchA
      ', 'anotherPatchB'))
}
|-- FoamFile
|   {
|       format : ascii
|       class : dictionary
|       object : blockMeshDict
|   }
```

```
\-- boundary
    |-- movingWall
    |   {
    |      type : wall
    |      faces : ((3, 7, 6, 2),)
    |   }
    |-- fixedWalls
    |   {
    |      type : wall
    |      faces : ((0, 4, 7, 3), (2, 6, 5, 1), (1, 5, 4, 0))
    |   }
    \-- frontAndBack
        {
           type : empty
           faces : ((0, 3, 2, 1), (4, 5, 6, 7))
        }
```

This abstract tree makes it easy to manipulate the structure in code or pass it to a visual frontend like Blender.

## 3.6 Parser Grammar (for OpenFOAM Dictionary Format)

```
file : blocks

blocks : blocks block | block

block : dictionary | listblock | statement | hexEdge_items | coordlists
    | empty

dictionary : WORD LBRACE blocks RBRACE

listblock : WORD LPAREN blocks RPAREN SEMICOLON

hexEdge_items : hexEdge_items hexEdge_item | hexEdge_item

hexEdge_item : hex_item | edge_item

hex_item : WORD LPAREN NUMBER NUMBER NUMBER NUMBER NUMBER NUMBER NUMBER
    NUMBER RPAREN LPAREN NUMBER NUMBER NUMBER RPAREN word gradlist

edge_item : WORD number number gradlist

gradlist : coodlist | LPAREN coordlists RPAREN

coordlists : coordlists coodlist | coodlist

coodlist : LPAREN anylist RPAREN

statement : WORD anylist SEMICOLON

anylist : anylist sitem | sitem

sitem : word | number | dimension | vector | empty
```

```
30
31 vector : LPAREN NUMBER NUMBER NUMBER RPAREN
32
33 dimension : LSQUABRAC NUMBER NUMBER NUMBER NUMBER NUMBER NUMBER NUMBER
       RSQUABRAC
```

Listing 7: Parser Grammer

## 3.7 YAML Configuration Syntax

This section explains the structure and syntax of YAML files used to represent OpenFOAM dictionaries in a structured, readable format. YAML supports scalars, dictionaries, and lists.

### 3.7.1 Key-word entry

```
1 key: "entry1 entry2 ..."
2 OR
3 key: singleEntry
```

Listing 8: Key-word entries

This corresponds to the following OpenFOAM syntax:

```
1 format: "ascii"
2 format ascii;
```

Listing 9: Key-word entry Equivalent OpenFOAM Syntax

### 3.7.2 Dictionary

```
1 FoamFile:
2   format: "ascii"
3   class: "dictionary"
4   object: "blockMeshDict"
```

Listing 10: Dictionary

This corresponds to the following OpenFOAM syntax:

```
1 FoamFile
2 {
3     format ascii;
4     class dictionary;
5     object blockMeshDict;
6 }
```

Listing 11: Dictionary Equivalent OpenFOAM Syntax

### 3.7.3 List (Sequence)

Lists are defined using dashes '-':

```
vertices:
  - [0, 0, 0]
  - [1, 0, 0]
  - [1, 1, 0]
```

Listing 12: YAML List Example

This corresponds to the following OpenFOAM syntax:

```
vertices
(
    (0 0 0)
    (1 0 0)
    (1 1 0)
);
```

Listing 13: Equivalent OpenFOAM List

### 3.7.4 Mixed-Type List

Certain OpenFOAM entries (e.g., blocks) are represented by ordered sequences in YAML:

```
blocks:
  - hex
  - [0, 1, 2, 3, 4, 5, 6, 7]
  - [20, 20, 1]
  - simpleGrading
  - [1, 1, 1]
```

Listing 14: Mixed-Type List in YAML

### 3.7.5 Complete YAML Example for blockMeshDict

The following is a full YAML representation of an OpenFOAM `blockMeshDict` file:

```
FoamFile:
  format: "ascii"
  class: "dictionary"
  object: "blockMeshDict"

convertToMeters: 0.1

vertices:
  - [0, 0, 0]
  - [1, 0, 0]
  - [1, 1, 0]
  - [0, 1, 0]
  - [0, 0, 0.1]
  - [1, 0, 0.1]
  - [1, 1, 0.1]
```

```yaml
  - [0, 1, 0.1]

blocks:
  - hex
  - [0, 1, 2, 3, 4, 5, 6, 7]
  - [20, 20, 1]
  - simpleGrading
  - [1, 1, 1]

edges:
  - arc
  - 0
  - 1
  - [0.5, 0.1, 0]
  - spline
  - 4
  - 5
  - [[4.1, 4.2, 4.3], [4.5, 4.6, 4.7], [4.9, 5.0, 5.1]]
  - polyLine
  - 6
  - 7
  - [[6.1, 6.2, 6.3], [6.5, 6.6, 6.7]]

boundary:
  - movingWall:
      type: "wall"
      faces:
        - [3, 7, 6, 2]

  - fixedWalls:
      type: "wall"
      faces:
        - [0, 4, 7, 3]
        - [2, 6, 5, 1]
        - [1, 5, 4, 0]

  - frontAndBack:
      type: "empty"
      faces:
        - [0, 3, 2, 1]
        - [4, 5, 6, 7]

mergePatchPairs: []
```

Listing 15: Complete YAML Configuration for blockMeshDict

## 3.8 `OpenFoamParser.parse_file` Function

**Functionality**

The `parse_file(text: str = None, fileType: str = 'txt', path: str = None)` method of the `OpenFoamParser` class is the primary entry point for parsing single OpenFOAM dictionary files or YAML configuration files into a PyVNT node tree. It accepts

either a direct text string or a file path and delegates the parsing task to internal text or YAML parsers accordingly.

This function supports multiple input formats (`.txt` for traditional OpenFOAM dictionaries, `.yaml` for YAML format) and chooses the appropriate internal parser based on file extension (if `path` is given) or the explicitly provided `fileType` (if `text` is given). The parsed result is returned as a PyVNT `Node_C` object.

**Parameters**

Table 1: Parameters for `OpenFoamParser.parse_file`

| Name | Type | Description |
|---|---|---|
| `text` | `str` | Optional. Raw string containing the file content to be parsed. |
| `fileType` | `str` | Optional. Type of the input ('txt' or 'yaml'). Required if `text` is provided and type cannot be inferred. |
| `path` | `str` | Optional. Path to the OpenFOAM dictionary or YAML file. If provided, content is read from this file. |

**Output**

- Returns a PyVNT `Node_C` object representing the root of the parsed structure.
- Returns `None` if parsing fails (e.g., file not found, unsupported format, syntax errors).

**Behavior Summary**

- If a `path` is provided:
  - Reads the file from disk.
  - Determines the parser based on file extension (`.txt`, no extension for dictionary, or `.yaml`).
  - Parses the content. The root node of the PyVNT tree is typically named after the file (e.g., "blockMeshDict").
- If `text` is provided:
  - Uses `fileType` to choose the parser ('txt' or 'yaml').
  - Parses the given string directly. The root node might be named 'root' by default, which can be renamed by the user.
- Internal parsers handle the specifics of dictionary syntax or YAML structure conversion to PyVNT objects.

**Example Use Case (Parsing YAML from text)** This example demonstrates parsing YAML content directly and then displaying the tree structure. A similar approach works for dictionary files or parsing from file paths. The expected output would be a tree structure printed to the console:

```

# parse_yaml_example.py
from pyvnt import OpenFoamParser, Node_C, show_tree  # Assuming these are top-level imports

# Sample YAML content (can also be read from a file)
yaml_content = "YAML EXAMPLE LISTING 15"

# Initialize the parser
parser = OpenFoamParser()

# Parse the YAML string
# For a file: tree = parser.parse_file(path="path/to/your/blockMeshDict.yaml")
tree = parser.parse_file(text=yaml_content, fileType='yaml')

# If parsed from text, the root node might be named 'root' by default.
# You can rename it for clarity:
if tree and tree.name == "root":  # Default name from parser for text input
    tree.name = "blockMeshDict_from_yaml"

# Display the generated tree
if tree:
    show_tree(tree)
else:
    print("Failed to parse YAML content.")
```

Listing 16: Parsing YAML content using `OpenFoamParser`

## 3.9 `OpenFoamParser.parse_case` Function

**Functionality**

The `parse_case(path: str)` method of the `OpenFoamParser` class recursively parses an entire OpenFOAM case directory structure. It walks through folders (e.g., `system/`, `constant/`, `0/`), attempting to parse recognized files (`.txt`, dictionary files without extension, `.yaml`) and building a hierarchical PyVNT tree. Each folder becomes a parent `Node_C`, and each successfully parsed file becomes a child `Node_C`. This creates a full node-based representation of the case directory. (Note: The README mentions this is in beta as it might try to parse any file; this behavior should be considered.)

**Parameters**

Table 2: Parameters for `OpenFoamParser.parse_case`

| Name | Type | Description |
|---|---|---|
| `path` | `str` | The path to the top-level OpenFOAM case directory or a subdirectory. |

**Output**

- Returns a PyVNT `Node_C` object, representing the root of the parsed case directory, with children representing successfully parsed files and subdirectories.

**Behavior Summary**

- Creates a master `Node_C` named after the directory specified in `path`.
- Iterates through each item (file or subfolder) in the directory:
  - If it's a **directory**, calls `parse_case()` recursively for that subdirectory.
  - If it's a **file** that the parser recognizes (based on extension or heuristics), calls `parse_file()` to parse its contents into a PyVNT tree.
- Attaches each successfully parsed file tree or subdirectory tree as a child to the master `Node_C`.

**Example Use Case**

```
case_tree = parser.parse_case("OpenFOAM/cavity/")
```

Listing 17: `parse_case` function to parse an OpenFOAM case directory

This would return a root node named `"your_OpenFOAM_case_directory_name"` with children like `"system"`, `"0"`, and `"constant"`—each containing their respective parsed content as further PyVNT sub-trees.

**Key Advantages**

- Enables full-case visualization and manipulation within a single PyVNT tree structure.
- Allows for comprehensive case loading into tools like the Venturial Node system.

## 3.10 `OpenFoamParser.get_value` Utility Function

**Functionality**

The `get_value(node: Node_C, *keys)` method (typically part of the `OpenFoamParser` instance or a utility associated with PyVNT) is used to query a specific element from a parsed PyVNT tree by providing a sequence of keys representing the path to the desired element. It performs a traversal of the tree.

**Parameters**

Table 3: Parameters for `get_value`

| Name | Type | Description |
|---|---|---|
| `node` | `Node_C` | The root PyVNT `Node_C` from which to start the traversal. |
| `*keys` | `str` | A sequence of one or more strings representing the path (names of nested nodes or keys) to the desired value. |

**Output**

- Returns the corresponding PyVNT object (`Node_C`, `Key_C`, or a `Value_P` derivative like `List_CP`) if the path exists.
- Returns `None` if any key in the path is not found.

**Behavior Summary**

- Starts at the provided `node`.
- For each key in `*keys`, it searches within the current node's children (`Key_C`s or nested `Node_C`s) for an item with that name.
- If found, it moves to that item for the next key in the sequence.
- If a key is not found at any step, the search stops and returns `None`.

**Features**

- Simplifies accessing deeply nested data within a PyVNT tree.
- Useful for scripting, data validation, or extracting specific configuration details.

**Limitations**

- Assumes keys in the sequence exist in the expected order.
- Does not handle advanced OpenFOAM syntax like macro expansion (`#include`) during traversal; it operates on the already parsed static tree.

**Example Use Case**

```
# Get the 'tolerance' value under 'solvers'
tolerance_node = parser.get_value(case_tree, 'solvers', 'p', 'tolerance')
```

Listing 18: `get_value` usage to access a nested key from the OpenFOAM case tree

## 3.11 Challenges and Solutions

During the development of the parser, several technical challenges were encountered due to the non-standard structure of OpenFOAM configuration files and the need to support YAML as well. Each challenge was addressed with targeted solutions:

- **Custom Syntax Handling (OpenFOAM Dictionaries):** OpenFOAM files don't follow standard formats like JSON or XML, requiring a custom tokenizer and brace-matching mechanism.
  **Solution:** A custom tokenizer and brace-matching mechanism were implemented. A recursive parsing strategy was developed to handle nested structures, ensuring accurate interpretation of OpenFOAM dictionaries into PyVNT objects. (The report mentions PLY; PyVNT's dictionary parser might use similar principles or a different custom approach).

- **YAML Mapping:** YAML files, while structured, need their native Python dictionary/list representation mapped to PyVNT's specific class structure (`Node_C`, `Key_C`, etc.).
  **Solution:** After standard YAML parsing, a conversion layer iterates through the Python data structures and instantiates the appropriate PyVNT objects, recursively building the PyVNT tree.

- **Comment and Whitespace Skipping:** Files often include comments (`//`, `/* */`) and irregular spacing.
  **Solution:** For dictionary files, a preprocessing pass can be used to remove comments and normalize whitespace. YAML parsers typically handle comments inherently.

- **Deep Nesting:** Some files contain deeply nested structures.
  **Solution:** Recursive parsing (for dictionaries) and recursive conversion (for YAML) handle these structures effectively within PyVNT.

- **Value Ambiguity / Type Inference:** Scalar values, dictionaries, and lists need to be correctly identified and typed within the PyVNT model. OpenFOAM often requires type inference (e.g. is "0" an int or a string for an enum? Is a list of lists representing vectors or something else?).
  **Solution:** The parser employs context-aware analysis and type inference rules. For YAML, types are more explicit but still need mapping to PyVNT's property types (e.g., `IntProperty`, `Enm_P`). PyVNT aims to create generic representations, with `Enm_P` often being used for string values that could have restricted options.

- **Consistency Across Frontend:** The data format needed to work seamlessly with the PyVNT backend and Blender frontend.
  **Solution:** The standardized PyVNT class interface ensures that the parser output (a PyVNT tree) is directly usable by downstream systems like the Tree-Nodes in Blender.

# 4 Venturial Node System in Blender

## 4.1 Venturial Node Structure

The Venturial Node addon follows Blender's modular addon architecture. The structure is organized as:

```
venturial_nodes/
    __init__.py
    ButtonDraw_UI_Header.py
    DataConvertore.py
    Examples/
        Collection_Property.py
        custom_nodes_template.py
    Nodes/
        Dict_Node.py
        Dim_Set_Node.py
        Enm_Node.py
```

```
        Flt_Node.py
        Int_Node.py
        Key_Node.py
        List_Node.py
        MultiValue_Node.py
        Node.py
        Output_Node.py
        Str_Node.py
        Tensor_Node.py
        Vector_Node.py
    Operator/
        List_Operators.py
        Node_Links_Swapper.py
    utils/
        Ven_Export.py
        Ven_Import.py
```

Listing 19: Python Greet Function

The node editor is integrated into Blender's UI via a custom node tree called Venturial Nodes.

- A new node editor window can be opened from the Editor Type menu.
- Within this editor, users can select the "Venturial Nodes" type to start working.

The node tree inherits from `bpy.types.NodeTree` and provides customized behaviors like linking validation and update callbacks. The underlying data for these nodes is managed by a PyVNT tree.

## 4.2 Node Design

Each node in the Venturial system visually represents a component of an OpenFOAM dictionary, corresponding to a PyVNT object (`Node_C`, `Key_C`, or `Value_P` derivative).

**Features:**

- Input/Output sockets allow visual construction of the PyVNT tree hierarchy.
- Custom drawing methods in Blender for dynamic field input.
- Property update callbacks synchronize Venturial Node UI with the backend PyVNT data structures.

# 5 Results and Output

## 5.1 Python Tree Output (PyVNT `show_tree`)

A parsed OpenFOAM file, like `blockMeshDict`, results in a PyVNT tree. The `show_tree` utility can visualize this textually:

This textual tree directly reflects the structure held by PyVNT objects and is the data source for the Blender GUI.

```
blockMeshDict
{
   convertToMeters : 0.1
   vertices : ((0, 0, 0), (1, 0, 0), (1, 1, 0), (0, 1, 0), (0, 0, 0.1), (1, 0, 0.1), (1, 1, 0.1), (0, 1, 0.
1))
   blocks : ('hex', (0, 1, 2, 3, 4, 5, 6, 7), (20, 20, 1), 'simpleGrading', (1, 1, 1))
   edges : ('arc', 0, 1, (0.5, 0.1, 0), 'spline', 4, 5, ((4.1, 4.2, 4.3), (4.5, 4.6, 4.7), (4.9, 5.0, 5.1))
, 'polyLine', 6, 7, ((6.1, 6.2, 6.3), (6.5, 6.6, 6.7)))
   mergePatchPairs : (('cyclicPair1A', 'cyclicPair1B'), ('anotherPatchA', 'anotherPatchB'))
}
├─ FoamFile
│  {
│     format : ascii
│     class : dictionary
│     object : blockMeshDict
│  }
└─ boundary
   ├─ movingWall
   │  {
   │     type : wall
   │     faces : ((3, 7, 6, 2),)
   │  }
   ├─ fixedWalls
   │  {
   │     type : wall
   │     faces : ((0, 4, 7, 3), (2, 6, 5, 1), (1, 5, 4, 0))
   │  }
   └─ frontAndBack
      {
         type : empty
         faces : ((0, 3, 2, 1), (4, 5, 6, 7))
      }
```

Figure 2: Example PyVNT `show_tree` output for a parsed `blockMeshDict`.

Table 4: Venturial Node Types and their corresponding PyVNT elements and Design Aspects

| **Venturial Node Type** | **Corresponding PyVNT Element(s)** | **Design Elements in Blender** |
|---|---|---|
| `Dict_Node.py` | `Node_C` | - String field for dictionary name<br>- Input and output sockets for nesting<br>- Side panel for renaming |
| `Output_node.py` | Root `Node_C` (conceptual) | - Single input socket for root `Node_C`<br>- "Export" button<br>- Shows status. Also acts as head/input node. |
| `List_Node.py` | `List_CP` (holding `Node_C`s or other `Value_P`s) | - Input socket for items.<br>- Toggle for list type (nodes vs. simple data). |
| `Key_Node.py` | `Key_C` | - String property for key name<br>- Input socket for its value (`Value_P` derivative). |
| `Int_Node.py` | `IntProperty` (or `Int_P`) | - Integer input field. |
| `Flt_Node.py` | `FloatProperty` (or `Flt_P`) | - Float input field. |
| `Str_Node.py` | `Enm_P` (or generic string) | - String input field. May include dropdown for enum options. |
| `Vector_Node.py` | `List_CP` of 3 numbers (often from `FloatProperty`) | - Three separate FloatProperty fields, combines into tuple/list. |
| `Dim_Set_Node.py` | `List_CP` of 7 numbers | - Seven FloatProperty fields. Exports as a list. |
| `MultiValue_Node.py` | `List_CP` of a single type (e.g., multiple `IntProperty`) | - UI for adding/removing subfields of a single data type. |

## 5.2 Blender GUI (Venturial Nodes)

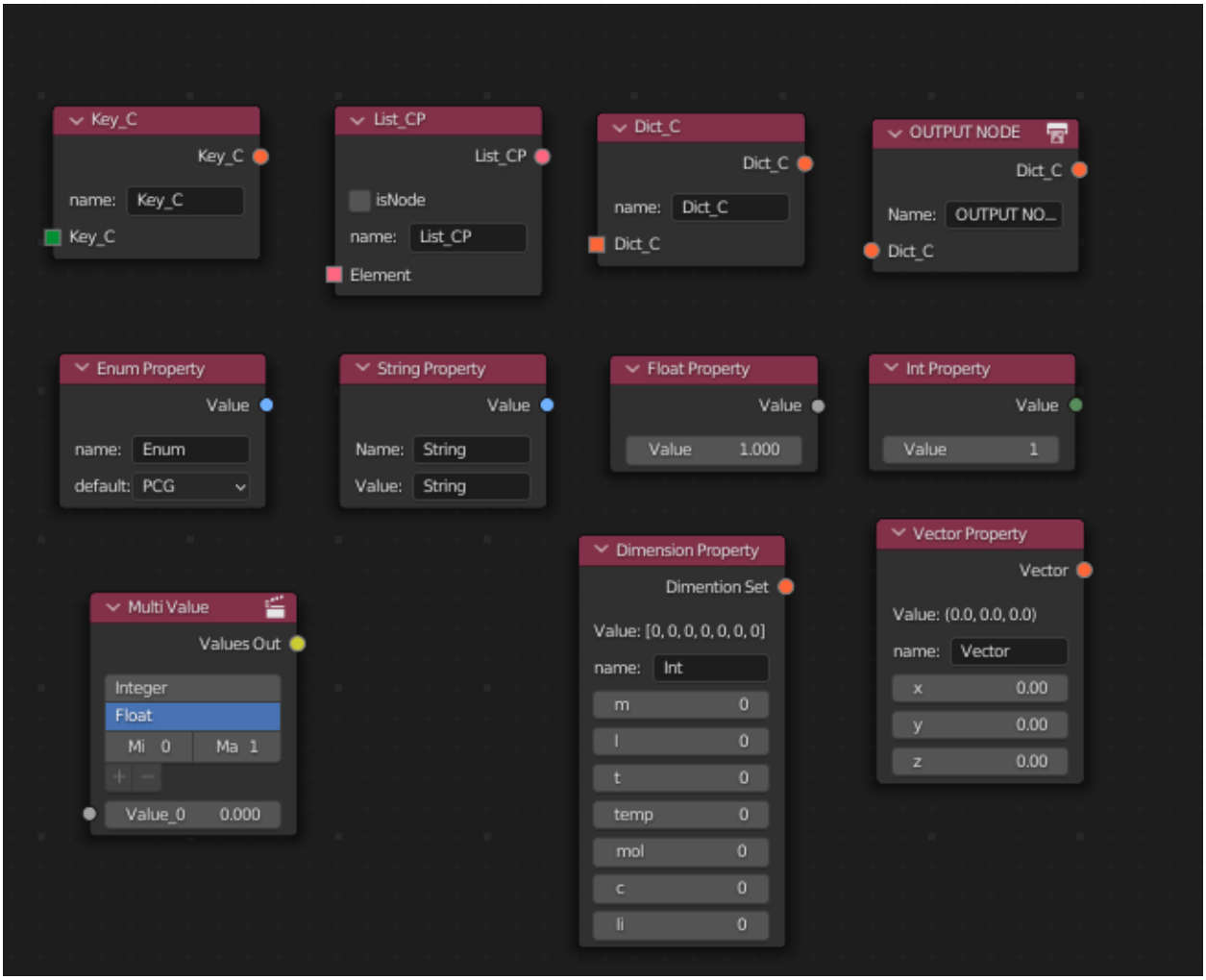


Figure 3: Venturial Node interface in Blender, displaying a part of an OpenFOAM case.

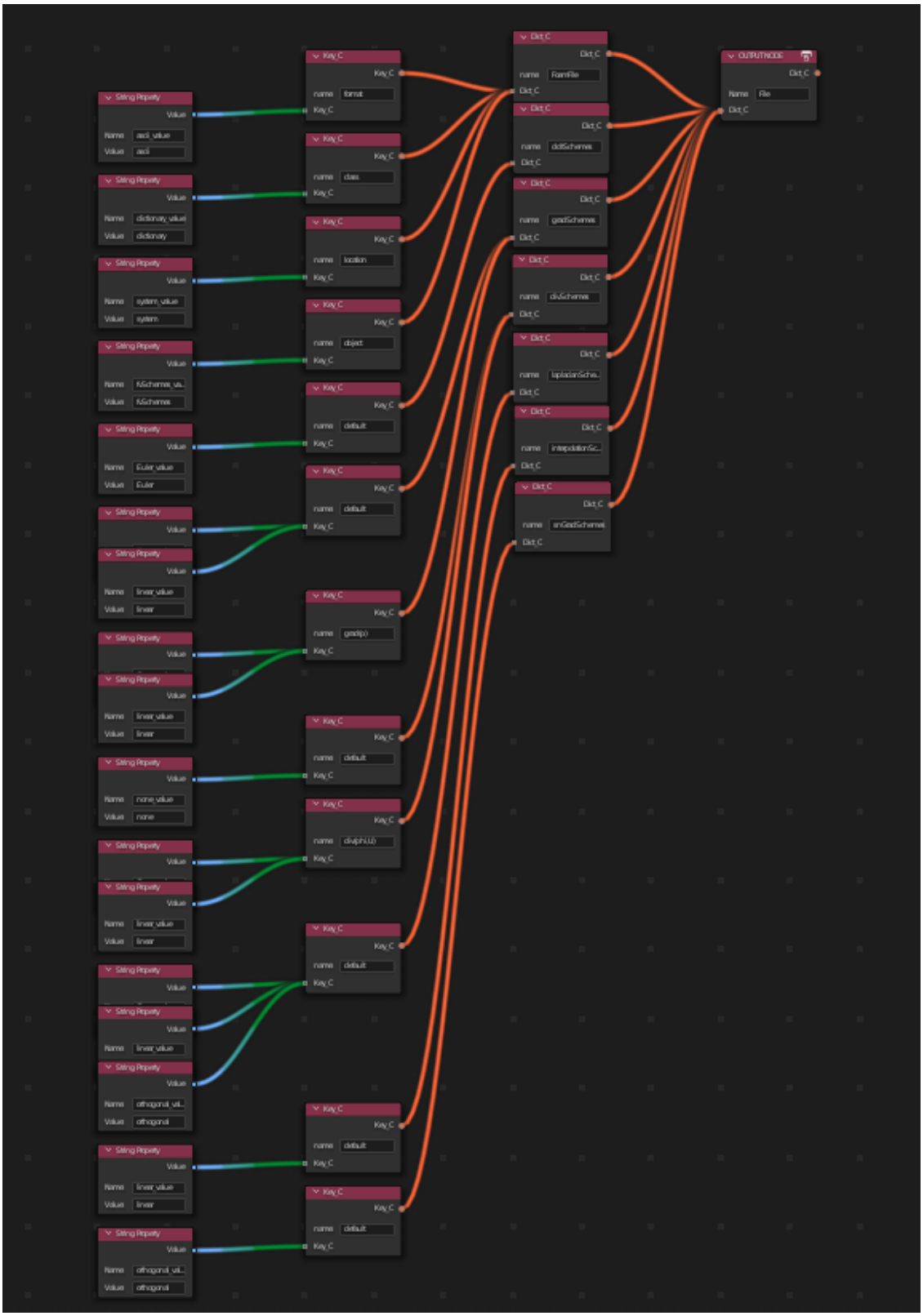

Figure 4: Detailed view of interconnected Tree-Nodes representing Cavity/system/fvSchemes.

# 6 Conclusion and Future Work

## 6.1 Summary of Work

During this internship, a parser was developed, capable of interpreting both traditional OpenFOAM dictionary files and compatible YAML configuration files. This parser was designed as a core component of the PyVNT (Python Venturial Node Trees) library, facilitating interaction with OpenFOAM case structures.

The developed `OpenFoamParser` within PyVNT successfully translates the complex, nested syntax of these files into a structured, hierarchical tree representation using PyVNT's defined data classes (e.g., `Node_C`, `Key_C`, `Value_P`). Key functionalities of PyVNT that were established around this parser include serializing these trees back to file formats using the `writeTo` method, and mechanisms for controlling element order in dictionary output. The parser supports both standalone file parsing and full OpenFOAM case directory traversal.

While the central objective was the parser's development and its integration into the PyVNT data model, its utility was further demonstrated by enabling the integration of PyVNT trees with a visual frontend. A node-based system, **Venturial Nodes**, was created inside Blender, which uses the PyVNT tree (generated by the parser) as its backend data model. This allowed for interactive editing of OpenFOAM case files, showcasing the parser's role in a complete workflow from raw file input to a graphical environment.

## 6.2 Impact on OpenFOAM Usability

OpenFOAM is a powerful tool used in computational fluid dynamics (CFD) for simulating complex physical phenomena. However, the software primarily relies on manually edited dictionary files, which can be time-consuming and error-prone, especially for users who are just beginning to explore CFD workflows or OpenFOAM itself.

The parser developed as part of PyVNT, and the broader PyVNT library (encompassing data structures and utilities), significantly improve the user experience by providing:

- A fully automated parser in PyVNT that handles OpenFOAM dictionary syntax and YAML mapping, eliminating much of the need for manual syntax handling.
- A structured, object-oriented PyVNT tree-based representation of cases, making them easier to analyze, manipulate programmatically, and transform.
- A clear pathway for GUIs, like the Blender-integrated **Venturial Node System**, to consume and interact with OpenFOAM data by leveraging PyVNT's parsed output.

These tools, with the parser at their core, drastically improve the accessibility and user experience of OpenFOAM, especially for:

- New users unfamiliar with its syntax.
- Students and educators looking for a more visual learning experience (enabled by GUIs using PyVNT).
- Researchers and engineers aiming for quick, error-free, and programmatic case editing.

## 6.3 Scope for Enhancements (for PyVNT and Venturial System)

While PyVNT and the Venturial Node system lay a solid foundation, several enhancements can be made:

1. **Live File Synchronization**
   - Enable real-time file watching; if an OpenFOAM/YAML file linked to a PyVNT tree is changed externally, offer to reload/update the tree and GUI.
2. **Export/Import Enhancements for PyVNT**
   - PyVNT's `writeTo` function already supports `.txt` and `.yaml`. Ensure YAML output is optimized for readability and potential use as an intermediate format for Venturial or other tools.
   - Enhance `OpenFoamParser` to more robustly handle various OpenFOAM tutorial cases as templates, potentially with more sophisticated type inference for ambiguous entries.
3. **Enhanced GUI Features**
   - Add inline documentation or tooltips inside Blender for each node parameter, possibly sourcing information from PyVNT's structure (e.g., `Enm_P` options).
   - Allow users to search and filter nodes based on OpenFOAM categories or PyVNT object types.
4. **Enable Macros Support in PyVNT Parser**
   OpenFOAM configuration files often make use of **macros**, such as `#include`, `#calc`, and `#codeStream`. Currently, PyVNT's `OpenFoamParser` processes the static content of dictionary files and does not evaluate or resolve these macros.

   A future enhancement for PyVNT would involve extending the parser to:
   - Recognize macro directives.
   - Optionally resolve `#include` paths by parsing and merging the content of included files into the main PyVNT tree structure.
   - Represent unresolved macros or expressions in a way that Venturial Nodes can display them (e.g., as special read-only nodes or string values).

   This would improve PyVNT's ability to handle complex, real-world OpenFOAM cases.
5. **Advanced Type Handling and Tensor Support in PyVNT and Venturial Nodes**
   Support for more complex OpenFOAM data types, like vectors, symmetric tensors, and full tensors, needs robust handling in both Parser and the Venturial Nodes.
   - **Venturial Nodes**: Develop specialized Venturial Nodes for these tensor types, providing intuitive UI elements in Blender for viewing and editing their components (e.g., dedicated vector input nodes, matrix-like displays for tensors).

This ensures accurate parsing, editing, and re-exporting of tensor data while maintaining compatibility with OpenFOAM's syntax.

# References

[1] Hrvoje Jasak. "OpenFOAM: Open source CFD in research and industry". *International Journal of Naval Architecture and Ocean Engineering*, vol. 1, Dec. 2009, pp. 89–94. Available at: dictionary Class Reference | OpenFOAM Source Code Guide

[2] Blender Foundation. *Blender Python API Documentation*. Available at: https://docs.blender.org/api/current/

[3] PyFoam GitHub Repository. Available at: https://github.com/takaakiaoki/PyFoam

[4] Python Software Foundation. *PLY (Python Lex-Yacc) Documentation*. Available at: https://ply.readthedocs.io/en/latest/index.html