IFCManager

Specialized tool for loading, analysis and visualization of AEC models from IFC files

---

IFCManager Version 0.1.0 MIT

Quick Start

```python
from ifc_manager.core import IFCManager

converter = IFCManager()
converter.load_ifc('sample_models/example.ifc')
structural_elements = converter.extract_structural_elements()
converter.visualize_model()
```

Description

IFCManager is a Python module designed to streamline the interaction with IFC (Industry Foundation Classes) files for structural engineering and BIM workflows. Built on top of ifcopenshell, it abstracts low-level complexity and provides a minimal, intuitive interface for loading, inspecting, analyzing, and visualizing structural components within IFC models.

This module is suitable for structural engineers, BIM technicians, and researchers who wish to integrate BIM data into custom workflows—particularly in Python—without requiring advanced knowledge of the IFC schema or programming.

```
IFC_MANAGER/
├── src/
│   └── ifc_manager/
│       ├── core/
│       │   ├── __init__.py
│       │   ├── ifc_parser.py
│       │   └── visualization.py
│       ├── utils/
│       │   ├── __init__.py
│       │   └── geometry_utils.py
│       └── __init__.py
```

API Reference

IFCManager Class

Constructor

`py converter = IFCManager()` → IFCManager

Initializes the IFC converter. The converter handles loading IFC files, extracting structural elements, and providing access to element data through the structural_elements attribute.

Core Methods

`py converter.load_ifc(ifc_file_path)` → None

Loads the IFC model file into memory using the ifcopenshell backend. This method must be called before any extraction operations can be performed.

`py converter.extract_structural_elements()` → dict

Parses and extracts structural elements (beams, columns, walls, slabs, footings, piles, members) from the loaded IFC file. Returns a dictionary of structural elements and populates the converter.structural_elements attribute. Extracted types include: IfcBeam, IfcColumn, IfcSlab, IfcWall, IfcFooting, IfcPile, IfcMember.

`py converter.visualize_model()` → plot

Generates a 3D visualization of the parsed IFC structural model. Useful for debugging geometry and structure placement. Must be called after extract_structural_elements().

Data Access

`py converter.structural_elements` → dict

Dictionary containing all extracted structural elements keyed by their GlobalId. This attribute is populated after calling extract_structural_elements() and contains complete element data including materials, profiles, and properties.

Dependencies

The module requires the following Python packages:

Core dependencies for IFC processing: *ifcopenshell*[1] provides tools for parsing IFC files, accessing geometry, property sets, and element placements within BIM models. *numpy*[2] handles numerical operations and array manipulations for geometric calculations.

Data analysis and visualization: *pandas*[3] for structured data manipulation and Excel export capabilities. *matplotlib*[4] provides plotting functionality for charts and graphs. *plotly*[5] enables interactive 3D model visualization.

[1] https://pypi.org/project/ifcopenshell
[2] https://pypi.org/project/numpy
[3] https://pypi.org/project/pandas
[4] https://pypi.org/project/matplotlib
[5] https://pypi.org/project/plotly
[6] https://pypi.org/project/logging
[7] https://pypi.org/project/math

Built-in Python modules: *logging*[6] for debugging and monitoring operations, *math*[7] for mathematical functions, and *os*[8] for file system operations.

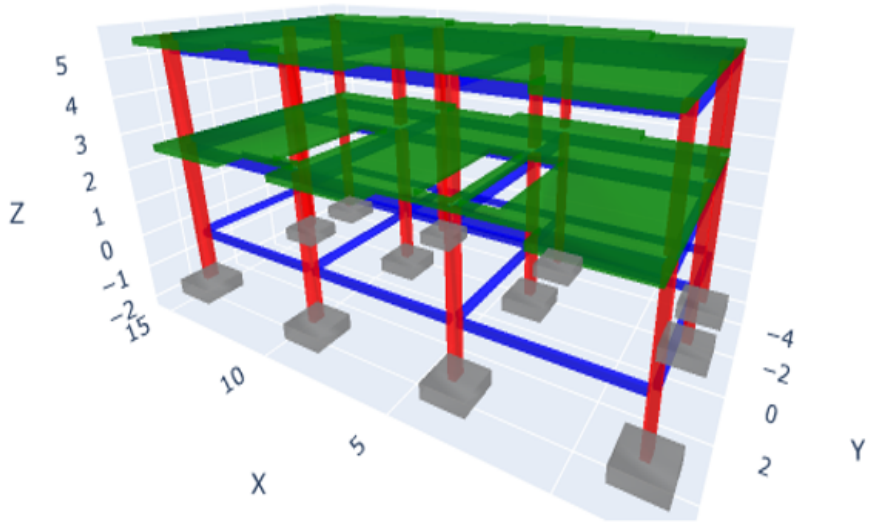
Examples

Basic Element Extraction and Visualization

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from ifc_manager.core import IFCManager

converter = IFCManager()
converter.load_ifc('sample_models/example.ifc')
structural_elements = converter.extract_structural_elements()
print(f"Extracted elements: {len(structural_elements)}")
print("Element types:", {elem['type'] for elem in structural_elements.values()})
converter.visualize_model()
```

Output:
Extracted elements: 89
Element types: {'IfcBeam', 'IfcColumn', 'IfcSlab', 'IfcWall', 'IfcFooting'}



Detailed Element Analysis

```
# Extract structural elements (this populates converter.structural_elements)
converter.extract_structural_elements()

# Access the extracted data
structural_elements_data = converter.structural_elements
```

[8]https://pypi.org/project/os

```
for element_id, data in structural_elements_data.items():
    print(f"--- Element ID: {element_id} ---")
    print(f"  Name: {data.get('name', 'N/A')}")
    print(f"  Type: {data.get('type', 'N/A')}")

    # Access material information
    if data.get('material'):
        print("  Materials:")
        for mat_name, mat_data in data['material'].items():
            print(f"    - {mat_name} (Type: {mat_data.get('type', 'N/A')})")
            if mat_data.get('thickness') is not None:
                print(f"      Thickness: {mat_data['thickness']}")
            if mat_data.get('profile'):
                print(f"      Profile: {mat_data['profile']}")

    # Access profile information for linear elements
    if data.get('profile'):
        print(f"  Profile Information: {data['profile']}")

    # Access general properties (Psets)
    if data.get('properties'):
        print("  Other Properties:")
        for prop_name, prop_value in data['properties'].items():
            print(f"    - {prop_name}: {prop_value}")
```

Data Export to Excel

```
# Extract structural elements
converter.extract_structural_elements()
structural_elements_data = converter.structural_elements

# Create list to store data
datos = []

for element_id, data in structural_elements_data.items():
    # Create dictionary for each element
    elemento = {
        'Element ID': element_id,
        'Name': data.get('name', 'N/A'),
        'Type': data.get('type', 'N/A')
    }

    # Add material information
    if data.get('material'):
        for mat_name, mat_data in data['material'].items():
            elemento[f'Material {mat_name}'] = mat_data.get('type', 'N/A')
            if mat_data.get('thickness') is not None:
                elemento[f'Material {mat_name} Thickness'] = mat_data['thickness']

    # Add profile information for linear elements
    if data.get('profile'):
        elemento['Profile Information'] = data['profile']

    # Add general properties (Psets)
    if data.get('properties'):
        for prop_name, prop_value in data['properties'].items():
```

```
            elemento[f'Property {prop_name}'] = prop_value

    datos.append(elemento)

# Create DataFrame and export to Excel
df = pd.DataFrame(datos)
df.to_excel('structural_elements_data.xlsx', index=False)
```

Material Distribution Analysis

```
# Create dictionary to store element count by material
materiales = {}
for elemento in datos:
    material = elemento.get('Material Concrete, Cast-in-Place gray', 'Otro')
    if material not in materiales:
        materiales[material] = 0
    materiales[material] += 1

# Create pie chart for material distribution
plt.pie(materiales.values(), labels=materiales.keys(), autopct='%1.1f%%')
plt.title('Distribución de materiales')
plt.show()
```

Volume Analysis by Type and Material

```
# Create dictionary to store quantities by type and material
tipos_materiales = {}
for elemento in datos:
    tipo = elemento['Type']
    material = elemento.get('Material Concrete, Cast-in-Place gray', 'Otro')
    if tipo not in tipos_materiales:
        tipos_materiales[tipo] = {}
    if material not in tipos_materiales[tipo]:
        tipos_materiales[tipo][material] = 0

    # Add volumes from different quantity sets
    volume_properties = [
        'Property Qto_BeamBaseQuantities.NetVolume',
        'Property Qto_ColumnBaseQuantities.NetVolume',
        'Property Qto_SlabBaseQuantities.NetVolume',
        'Property Qto_FootingBaseQuantities.NetVolume',
        'Property Qto_WallBaseQuantities.NetVolume'
    ]

    for vol_prop in volume_properties:
        if vol_prop in elemento:
            tipos_materiales[tipo][material] += elemento[vol_prop]

# Create stacked bar chart
tipos = list(tipos_materiales.keys())
materiales = list(set([material for tipo in tipos_materiales.values()
                       for material in tipo.keys()]))
x = np.arange(len(tipos))
ancho = 0.8 / len(materiales)
```

```
for i, material in enumerate(materiales):
    cantidades = [tipos_materiales[tipo].get(material, 0) for tipo in tipos]
    plt.bar(x + i * ancho, cantidades, width=ancho, label=material)

plt.xlabel('Tipo de elemento')
plt.ylabel('Volumen (m³)')
plt.title('Distribución de volúmenes por tipo y material')
plt.xticks(x + ancho * (len(materiales) - 1) / 2, tipos)
plt.legend()
plt.show()
```

Data Structure

Each extracted structural element in converter.structural_elements contains:

```
{
    'id': 'GlobalId_string',
    'name': 'Element name',
    'type': 'IfcBeam|IfcColumn|IfcSlab|etc',
    'geometry': {
        'type': 'line|mesh',
        'start': [x, y, z],           # For line type
        'end': [x, y, z],             # For line type
        'vertices': [[x,y,z], ...],   # For mesh type
        'triangles': [[i,j,k], ...]   # For mesh type
    },
    'material': {
        'material_name': {
            'name': 'Material name',
            'type': 'material|layer|profile',
            'thickness': float,           # For layers
            'profile': 'profile_name',    # For profiles
            'properties': {...}
        }
    },
    'profile': {                      # For linear elements
        'type': 'Profile IFC type',
        'shape': 'rectangular|circular|I-shape',
        'width': float,
        'height': float,
        'area': float
    },
    'properties': {
        'PropertySet.PropertyName': 'value',
        'Qto_BeamBaseQuantities.NetVolume': float,
        'Qto_ColumnBaseQuantities.NetVolume': float,
        # ... other quantity and property sets
    }
}
```

Features

Structural Element Extraction

The converter automatically identifies and extracts key structural elements including beams, columns, slabs, walls, footings, piles, and structural members. Each element is processed with complete metadata including names, types, unique identifiers, and IFC property sets.

Material Analysis

Comprehensive material extraction supports multiple IFC material representations including direct materials, material lists, layered materials, and profile-based materials. Material properties and quantities are automatically collected from property sets.

Quantity Takeoffs

Access to IFC quantity sets (Qto **BaseQuantities**) enables automated volume, area, and length calculations for cost estimation and material quantification workflows.

Data Export Capabilities

Built-in support for exporting extracted data to Excel format through pandas integration, enabling further analysis in spreadsheet applications and integration with existing workflows.

Visualization

Interactive 3D model visualization provides immediate visual feedback for model validation and geometry verification, helping identify potential issues in the IFC data.

Statistical Analysis

Integration with matplotlib and pandas enables comprehensive statistical analysis of building components, including material distribution charts, volume analysis, and comparative studies.

Property Sets

The module automatically extracts common IFC property sets including:

- **Base Quantities**: Qto_BeamBaseQuantities, Qto_ColumnBaseQuantities, Qto_SlabBaseQuantities, Qto_WallBaseQuantities, Qto_FootingBaseQuantities
- **Material Properties**: Physical and thermal properties from material definitions
- **Custom Property Sets**: User-defined property sets from the IFC model
- **Element Properties**: Standard IFC element properties and classifications

Copyright