**Code Complexity Estimation**

The Python program is designed to estimate the complexity of C program files by analysing their decision points, which are key indicators of code complexity. It operates by scanning all C source files located within a specified directory path provided by the user. To identify decision points such as if, else, switch, and loop constructs, the program utilizes powerful regular expressions (REGEX). These patterns help in parsing the code structure effectively. The extracted data is then organized and processed using the Pandas library, which allows for efficient data manipulation and tabular representation. Additionally, the program leverages the Radon library, a tool specifically built for analysing code complexity metrics. Radon provides insights such as cyclomatic complexity, which quantifies the number of independent paths through the code. By combining these tools, the program delivers a comprehensive estimation of how complex each C file is. This analysis can be useful for developers aiming to refactor or better understand their codebase.

## Software Utilized

* **PyCharm Community Edition 2024.3(IDE)**: - PyCharm is a powerful and widely used Integrated Development Environment (IDE) developed by JetBrains, specifically designed for Python programming. It offers intelligent code completion, real-time error detection, and robust debugging tools, making it ideal for both beginners and professional developers. Additionally, PyCharm supports web development frameworks, version control systems, and seamless integration with tools like Jupyter notebooks and virtual environments.
* **Python Software Foundation 3.13.0 for Windows: -** Python 3.13.0, released on October 7, 2024, is the latest major version of the Python programming language, introduced by the Python Software Foundation. It brings several enhancements, including a new interactive interpreter with color support, experimental free-threaded execution (disabling the Global Interpreter Lock), and an early-stage Just-In-Time (JIT) compiler for performance improvements
* **Required Libraries: -** Pip install pandas, pip install radon, pip install openpyxl.  
  These libraries should install in the PyCharm terminal by using **PIP**(Keyword).

1. pip install pandas installs the Pandas library, which is widely used for data manipulation and analysis using data structures like Data Frames.
2. pip install radon installs Radon, a tool for analysing Python code complexity, including metrics like cyclomatic complexity and maintainability index.
3. pip install openpyxl installs OpenPyXL, a library used for reading and writing Excel files (.xlsx) in Python.

## Implementation changes

1. Output\_path = r' C: \Users\23\*\*\*\*\*\Downloads\Line\_check.xlsx'

The path specifies the location where the Excel output file will be saved. However, this path is user-specific and should be updated to reflect the actual username and directory structure of the system where the script is being run. This ensures the file is saved correctly without causing permission or path errors.

1. Directory\_path = r' C; \Users\23\*\*\*\*\Downloads\File\_path\_name'

This path refers to the directory on the user's system where the folder containing the C program files is located. These C files are the ones that the Python script will analyze to estimate code complexity. The user should update this path to match the actual location of their C source files.

## Estimations

Once the Python script is executed using the specified Excel output path and the directory path containing the C program files, it begins analysing all the .c files present in that folder. The script processes each file to extract various metrics related to code complexity and structure. For every C file, it records details such as the **File Name**, **Programming language**, **Total lines of code**, and the **Number of Functions** defined. It also identifies the **Status of each function**, **Lists their names**, and calculates the **Number of lines per function**. Additionally, it computes the **Cyclomatic complexity** for each function, which helps estimate the difficulty of testing and maintaining the code. Finally, the script estimates the required **Estimation(Hours)** and **Man-days**, and saves all this information neatly into an Excel file for easy review and reporting.

## CODE

import os  
import re  
import pandas as pd  
from radon.complexity import cc\_visit  
  
def clean\_function\_content(content, language):  
 if language == 'python':  
 content = re.sub(r'#.\*', '', content)  
 content = re.sub(r'""".\*?"""', '', content, flags=re.DOTALL)  
 content = re.sub(r"'''.\*?'''", '', content, flags=re.DOTALL)  
 else:  
 content = re.sub(r'//.\*', '', content)  
 content = re.sub(r'/\\*[\s\S]\*?\\*/', '', content, flags=re.DOTALL)  
 content = '\n'.join(line for line in content.splitlines() if line.strip())  
 return content  
  
def extract\_function\_lines(content, language):  
 function\_lines = {}  
 if language == 'python':  
 functions = re.findall(r'def\s+(\w+)\s\*\(', content)  
 for func in functions:  
 func\_start = content.find(f'def {func}(')  
 func\_end = content.find('\ndef ', func\_start + 1)  
 if func\_end == -1:  
 func\_end = len(content)  
 function\_content = content[func\_start:func\_end]  
 cleaned\_function\_content = clean\_function\_content(function\_content, language)  
 lines = cleaned\_function\_content.split('\n')  
 function\_lines[func] = len(lines)  
 elif language in ['java', 'c', 'cpp']:  
 function\_pattern = r'^\s\*(?!else\s+if\s\*\()(?:(?:static\s+)?(?:[a-zA-Z\_]\w\*\s+)+[a-zA-Z\_]\w\*\s\*\([^)]\*\)\s\*\{|\bISR\s\*\(\s\*\w+\s\*\))'  
 functions = list(re.finditer(function\_pattern, content, re.MULTILINE))  
 for i, func in enumerate(functions):  
 func\_start = func.start()  
 if i + 1 < len(functions):  
 func\_end = functions[i + 1].start()  
 else:  
 func\_end = len(content)  
 function\_content = content[func\_start:func\_end]  
 cleaned\_function\_content = clean\_function\_content(function\_content, language)  
 lines = cleaned\_function\_content.split('\n')  
 func\_signature = content[func\_start:content.find('{', func\_start)].strip()  
 func\_name\_match = re.match(r'^\s\*(?:static\s+)?(?:[\w\\*]+\s+)+([\w\\*]+)\s\*\(', func\_signature)  
 if not func\_name\_match:  
 func\_name\_match = re.match(r'\b(?:[a-zA-Z\_]\w\*\s+)+([a-zA-Z\_]\w\*)\s\*\(|\bISR\s\*\(\s\*([a-zA-Z\_]\w\*)\s\*\)', func\_signature)  
 if func\_name\_match:  
 func\_name = func\_name\_match.group(1) or func\_name\_match.group(2)  
 function\_lines[func\_name] = len(lines)  
 else:  
 print(f"Warning: Could not extract function name from signature: {func\_signature}")  
 return function\_lines  
  
def remove\_comments(content):  
 content = re.sub(r'//.\*', '', content)  
 content = re.sub(r'/\\*[\s\S]\*?\\*/', '', content, flags=re.DOTALL)  
 return content  
  
def count\_decision\_points(content):  
 decision\_patterns = [  
 r'\bif\b', r'\belse\s\*if\b', r'\bfor\b', r'\bwhile\b', r'\bcase\b', r'\bswitch\b',  
 r'&&', r'\|\|', r'\?\:', r'\bdo\b', r'\bgoto\b'  
 ]  
  
 decision\_points = sum(len(re.findall(pattern, content)) for pattern in decision\_patterns)  
 decision\_points += len(re.findall(r'\[.\*?for.\*?in.\*?\]', content))  
  
 return decision\_points  
  
def count\_function\_calls(content):  
 # Improved regex pattern to find function calls  
 function\_calls = re.findall(r'\b\w+\s\*\([^)]\*\)\s\*;', content)  
 return len(function\_calls)  
  
def count\_function\_calls\_per\_function(content, language):  
 function\_calls\_per\_function = {}  
 if language == 'python':  
 functions = re.findall(r'def\s+(\w+)\s\*\(', content)  
 for func in functions:  
 func\_start = content.find(f'def {func}(')  
 func\_end = content.find('\ndef ', func\_start + 1)  
 if func\_end == -1:  
 func\_end = len(content)  
 function\_content = content[func\_start:func\_end]  
 cleaned\_function\_content = clean\_function\_content(function\_content, language)  
 function\_calls\_per\_function[func] = count\_function\_calls(cleaned\_function\_content)  
 elif language in ['java', 'c', 'cpp']:  
 function\_pattern = r'^\s\*(?!else\s+if\s\*\()(?:(?:static\s+)?(?:[a-zA-Z\_]\w\*\s+)+[a-zA-Z\_]\w\*\s\*\([^)]\*\)\s\*\{|\bISR\s\*\(\s\*\w+\s\*\))'  
 functions = list(re.finditer(function\_pattern, content, re.MULTILINE))  
 for i, func in enumerate(functions):  
 func\_start = func.start()  
 if i + 1 < len(functions):  
 func\_end = functions[i + 1].start()  
 else:  
 func\_end = len(content)  
 function\_content = content[func\_start:func\_end]  
 cleaned\_function\_content = clean\_function\_content(function\_content, language)  
 func\_signature = content[func\_start:content.find('{', func\_start)].strip()  
 func\_name\_match = re.match(r'^\s\*(?:static\s+)?(?:[\w\\*]+\s+)+([\w\\*]+)\s\*\(', func\_signature)  
 if not func\_name\_match:  
 func\_name\_match = re.match(r'\b(?:[a-zA-Z\_]\w\*\s+)+([a-zA-Z\_]\w\*)\s\*\(|\bISR\s\*\(\s\*([a-zA-Z\_]\w\*)\s\*\)', func\_signature)  
 if func\_name\_match:  
 func\_name = func\_name\_match.group(1) or func\_name\_match.group(2)  
 function\_calls\_per\_function[func\_name] = count\_function\_calls(cleaned\_function\_content)  
 return function\_calls\_per\_function  
  
def calculate\_cyclomatic\_complexity(content, language):  
 cyclomatic\_complexities = {}  
  
 if language == 'python':  
 blocks = cc\_visit(content)  
 for block in blocks:  
 cyclomatic\_complexities[block.name] = block.complexity  
 else:  
 if language in ['java', 'c', 'cpp']:  
 function\_pattern = r'^\s\*(?!else\s+if\s\*\()(?:(?:static\s+)?(?:[a-zA-Z\_]\w\*\s+)+[a-zA-Z\_]\w\*\s\*\([^)]\*\)\s\*\{|\bISR\s\*\(\s\*\w+\s\*\))'  
 functions = list(re.finditer(function\_pattern, content, re.MULTILINE))  
  
 for i, func in enumerate(functions):  
 func\_start = func.start()  
 if i + 1 < len(functions):  
 func\_end = functions[i + 1].start()  
 else:  
 func\_end = len(content)  
 func\_content = content[func\_start:func\_end]  
 cleaned\_function\_content = clean\_function\_content(func\_content, language)  
  
 decision\_points = count\_decision\_points(cleaned\_function\_content)  
 cyclomatic\_complexity = decision\_points + 1  
 func\_signature = content[func\_start:content.find('{', func\_start)].strip()  
 func\_name\_match = re.match(r'^\s\*(?:static\s+)?(?:[\w\\*]+\s+)+([\w\\*]+)\s\*\(', func\_signature)  
 if not func\_name\_match:  
 func\_name\_match = re.match(r'\b(?:[a-zA-Z\_]\w\*\s+)+([a-zA-Z\_]\w\*)\s\*\(|\bISR\s\*\(\s\*([a-zA-Z\_]\w\*)\s\*\)', func\_signature)  
 if func\_name\_match:  
 func\_name = func\_name\_match.group(1) or func\_name\_match.group(2)  
 cyclomatic\_complexities[func\_name] = cyclomatic\_complexity  
  
 return cyclomatic\_complexities  
  
def estimate\_hours(cyclomatic\_complexity):  
 if cyclomatic\_complexity <= 9:  
 return 2  
 elif 10 <= cyclomatic\_complexity <= 20:  
 return 10  
 elif 21 <= cyclomatic\_complexity <= 30:  
 return 16  
 elif 31 <= cyclomatic\_complexity <= 50:  
 return "22 - 35"  
 else:  
 if cyclomatic\_complexity > 50:  
 return "40 - 54"  
  
def analyze\_file(filepath, language):  
 try:  
 with open(filepath, 'r', encoding='utf-8', errors='ignore') as file:  
 content = file.read()  
 except Exception as e:  
 print(f"Error reading file {filepath}: {e}")  
 return  
  
 # Remove comments before processing the content  
 content\_no\_comments = remove\_comments(content)  
  
 num\_lines = len(content.splitlines())  
 function\_lines = extract\_function\_lines(content\_no\_comments, language)  
 cyclomatic\_complexities = calculate\_cyclomatic\_complexity(content\_no\_comments, language)  
 function\_calls\_per\_function = count\_function\_calls\_per\_function(content\_no\_comments, language)  
 functions = list(function\_lines.keys())  
  
 print(f'File: {filepath}')  
 print(f'Lines: {num\_lines}')  
 print(f'Functions: {functions}')  
 print(f'Function Lines: {function\_lines}')  
 print(f'Cyclomatic Complexities: {cyclomatic\_complexities}')  
 print(f'Function Calls per Function: {function\_calls\_per\_function}')  
  
 return num\_lines, functions, cyclomatic\_complexities, function\_lines, function\_calls\_per\_function  
  
def get\_c\_files\_from\_directory(directory):  
 c\_files = []  
 for root, \_, files in os.walk(directory):  
 for file in files:  
 if file.endswith('.c'):  
 c\_files.append(os.path.join(root, file))  
 return c\_files  
  
def main(directory):  
 *"""Main function to analyze all C files in a directory and save results to an Excel file."""* filepaths = get\_c\_files\_from\_directory(directory)  
 processed\_files = {}  
  
 data\_general = {  
 'File': [],  
 'Language': [],  
 'Lines of Code': [],  
 'Number of Functions': [],  
 }  
  
 function\_data = []  
  
 for filepath in filepaths:  
 filename = os.path.basename(filepath)  
 if filename in processed\_files:  
 continue  
  
 language = 'c'  
  
 results = analyze\_file(filepath, language)  
 if not results:  
 continue  
  
 num\_lines, functions, cyclomatic\_complexities, function\_lines, function\_calls\_per\_function = results  
  
 data\_general['File'].append(filename)  
 data\_general['Language'].append(language.capitalize())  
 data\_general['Lines of Code'].append(num\_lines)  
 data\_general['Number of Functions'].append(len(functions))  
  
 for func in functions:  
 complexity = cyclomatic\_complexities.get(func, 'N/A')  
 function\_calls = function\_calls\_per\_function.get(func, 'N/A')  
 estimation = estimate\_hours(complexity) if complexity != 'N/A' else 'N/A'  
 if isinstance(estimation, str):  
 hours = list(map(int, re.findall(r'\d+', estimation)))  
 if len(hours) == 2:  
 man\_days = [hour / 8 for hour in hours]  
 man\_days\_str = f"{man\_days[0]} - {man\_days[1]}"  
 else:  
 man\_days\_str = 'N/A'  
 else:  
 man\_days = estimation / 9 if isinstance(estimation, (int, float)) else 'N/A'  
 man\_days\_str = f"{man\_days}" if isinstance(man\_days, (int, float)) else man\_days  
 function\_data.append({  
 'File': filename,  
 'Language': language.capitalize(),  
 'Function Names': func,  
 'Function Lines': function\_lines.get(func, 'N/A'),  
 'Cyclomatic Complexity': complexity,  
 'Estimation (hours)': estimation,  
 'Man Days': man\_days\_str,  
 'Function Calls': function\_calls,  
 })  
  
 processed\_files[filename] = True  
  
 df\_general = pd.DataFrame(data\_general)  
 df\_functions = pd.DataFrame(function\_data)  
  
 output\_path = r'C:\Users\2318641\Downloads\Line\_check.xlsx'  
  
 try:  
 with open(output\_path, 'w') as f:  
 pass  
 except PermissionError:  
 output\_path = r'C:\Users\23\*\*\*\*\Downloads\Line\_check\_alternate.xlsx'  
  
 with pd.ExcelWriter(output\_path, engine='xlsxwriter') as writer:  
 df\_general.to\_excel(writer, sheet\_name='General\_Data', index=False)  
 df\_functions.to\_excel(writer, sheet\_name='Function\_Details', index=False)  
 print(f'Data has been written to {output\_path}')  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 directory\_path = r"C:\Users\23\*\*\*\*\Downloads\mall"  
 main(directory\_path)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |