Cascade Circuit Analyser Final Report

*EE20084 - Structured Programming

Jake Stewart
Department of Electrical and Electronic Engineering
University of Bath
Bath, United Kingdom
email: js3910@bath.ac.uk

***Abstract*—This report details the design choices, implementation and testing of the Cascade Circuit Analyser program. embedded.**

***Index Terms*—circuit analysis, Python, pytest, regex, software testing**

CONTENTS

I. OVERVIEW

This section provides a brief overview of the Cascade Circuit Analyser program, outlining its purpose, functionality, and overall problem-solving structure.

A. `main.py`

This module is the entry point for the application. It is primarily responsible for:

- Handling command line arguments.
- Encapsulating the main execution logic within a try-except block to handle exceptions effectively. This ensures that any unhandled exceptions in the lower modules are caught and managed properly.
- On encountering an exception, it displays the error message, creates an empty CSV file using the specified output argument, and terminates the program with a non-zero exit code to indicate an error state.

B. `net_parser.py`

This module is crucial for interpreting the netlist file into a usable format:

- The `parse_net_file_to_circuit` function reads the input file line-by-line, trimming whitespace and ignoring comments and empty lines.
- It recognizes different sections in the input file (e.g., component definitions or configurations) by matching lines with predefined delimiters.
- It validates the sequence of these sections to prevent processing errors and uses regular expressions to parse relevant data from each line.
- Extracted data is then converted into dictionaries or objects and integrated into a circuit object, facilitating structured access and manipulation in later stages.

C. `circuit.py`

This module defines the `Circuit` class, organizing all circuit-related data:

- The `Circuit` class is designed to store detailed information about the components, terminations, and outputs of a circuit.
- It contains sub-classes for components and outputs, which help in maintaining structured and easily accessible data.
- Its `solve` method plays a pivotal role by sorting components, computing the ABCD matrix for each frequency, and then writing these calculations to an output file.
- This method ensures that the computations adhere to the engineering requirements specified in the output file format, handling data consistently and accurately across various simulations.

D. `csv_writer.py`

Responsible for all CSV file operations, this module:

- Implements functions to write header rows, data rows, and generate an empty CSV file in case of errors.
- Ensures data is formatted in scientific notation, with precise alignment and scaling, whether for displaying results in decibels or linear formats.

E. `tests`

This directory contains all test files, including unit tests and integration tests. The aim was to achieve a high level of code coverage and ensure that the program functions correctly under the typical scenarios proposed in the example netlist files - but also extend to edge cases and cover most failure modes that I could think of.

II. Implementation

This section provides a detailed explanation of the implementation of the Cascade Circuit Analyser program, focusing on the key modules and functions that contribute to its functionality.

A. `net_parser.py`

This module is pivotal for parsing the netlist file and converting its contents into a structured format that the program can manipulate and analyze. Here, we detail the key functions:

- `parse_net_file_to_circuit`: This function serves as the starting point for processing the input netlist files.
  - **Functionality**: It initializes an empty circuit instance and begins reading the input file line by line. Each line is stripped of whitespace and checked for comments or emptiness, which are ignored.
  - **Delimiters and Section Matching**: The function uses a match case to identify section delimiters (such as headers for components, outputs, etc.). It ensures that sections are processed in a valid sequence, raising an error if an unexpected section is encountered.
  - **Data Handling**: Lines that do not match section delimiters are considered as data. If data appears outside of its designated section, the function raises an error. Otherwise, it is passed to specialized line processing helper functions for parsing.
- **`process_*_line`**: These functions are tasked with processing specific types of data lines, depending on the section they belong to.
  - **Data Extraction**: Each line of data is matched against a tailored regex pattern that extracts necessary information and returns it in the form of a dictionary.
  - **Component Addition**: The extracted data dictionary is passed to the appropriate `add_*` method of the circuit instance. This method is responsible for creating and appending a new instance of the respective Circuit Component or Output class, or updating the termination dictionary.
  - **Use of Regex**: Regular expressions play a crucial role in ensuring reliable and well-defined data extraction, including handling optional fields like magnitude and decibel prefixes. Magnitudes are applied immediately to standardize data storage format, with multipliers stored in a dictionary for quick access and conversion.
  - **Error Handling**: If a regex match fails or matches incorrectly, an error is raised to prevent incorrect data handling and ensure the integrity of the circuit data.
  - **Regex Development**: Regex patterns were developed and refined using https://regex101.com/, a tool that allows for meticulous testing and adjustment of regex expressions.

This detailed implementation ensures that the `net_parser.py` module robustly handles the parsing of the netlist file, setting a solid foundation for the subsequent circuit analysis processes.

B. `circuit.py`

The `circuit.py` module encapsulates the `Circuit` class, which is the core data structure for organizing and manipulating the circuit elements and their interactions in the Cascade Circuit Analyser program. This class and its methods are designed to store circuit data, compute necessary parameters, and facilitate the overall solution of the circuit analysis. Below are detailed descriptions of the key methods within this class:

- **Initialization:**
  - **Functionality**: The constructor method, `__init__`, initializes the lists for storing component and output instances, and a dictionary for terminations. This setup is crucial for managing the diverse types of data that are typical in circuit analysis applications.
- **Adding Components:**
  - **`add_component`**: This method accepts parameters defining a component (e.g., type, nodes, value) and creates an instance of a component class (e.g., Resistor, Capacitor). Each component class contains a method to calculate its specific ABCD matrix, which is critical for the network analysis.
  - **Data Organization**: Added components are organized in a list that maintains the order of their insertion, which is important for the sequential processing and analysis of the circuit.
- **Setting Terminations:**
  - **`set_termination`**: This method inputs termination values, such as source and load impedances. It updates the terminations dictionary, which is later used in matrix calculations to apply boundary conditions essential for accurate circuit analysis.
- **Adding Outputs:**
  - **`add_output`**: Specifies which outputs need to be calculated and recorded, such as voltages at specific nodes or overall power dissipation. This method ensures that the output specifications are stored for later use in determining what results are generated and stored.
- **ABCD Matrix Calculation:**
  - **`calculate_abcd_matrix`**: This method is responsible for computing the ABCD matrices of each component at the specified frequencies. It performs matrix multiplications of individual component matrices to derive an overall ABCD matrix for the circuit at each frequency.
- **Circuit Solution Method:**
  - **`solve`**: Acts as the main computational engine of the module. It iterates over the frequency range,

computes the total ABCD matrix for each frequency using `calculate_abcd_matrix`, and calculates the circuit's response based on the terminations. The results for each requested output are formatted and passed to the `csv_writer.py` for output to a file.
  - **Efficiency and Accuracy**: The efficiency of this method directly influences the program's performance, ensuring that the calculations are done in a logical sequence based on the accurate mathematical foundations of network theory.

This structured approach within the `circuit.py` module ensures comprehensive management and manipulation of circuit data, facilitating detailed and accurate circuit analysis. The module is designed to be flexible and robust, allowing for future enhancements and adjustments to the methods as new requirements or improvements are identified.

C. ***csv_writer.py***

The `csv_writer.py` module is dedicated to managing the output of the circuit analysis results into a CSV file format. This module plays a crucial role in ensuring data is accurately and effectively written to facilitate easy review and further processing. Below, we detail the functions implemented within this module and their interactions:

- **Write Header:**
  - **Functionality**: This function writes the header row of the CSV file, which typically includes the names of the variables being recorded, such as frequency, voltage, current, etc.
  - **Dynamic Headers**: Depending on the outputs specified in the analysis, this function dynamically adjusts to include the appropriate headers, ensuring the output file matches the expected format for subsequent analysis.
- **Write Data Rows:**
  - **Functionality**: Following the header, this function is responsible for writing the data rows into the CSV file.
  -
  - **Data Formatting**: The data is scaled to match the output magnitude, and formatted in scientific notation to ensure precision, and each value is appropriately aligned and spaced to maintain a structured and readable format. This is particularly important for ensuring the data can be easily imported and manipulated in data analysis tools.
- **Write Empty CSV:**
  - **Error Handling**: In cases where the program encounters an error that prevents the completion of the circuit analysis, this function generates an empty CSV file.
  - **Indication of Failure**: The creation of an empty CSV file serves as an indication that the analysis did not proceed as expected, which can be a critical signal for automated systems or further batch processing that relies on the output data.
- **Scientific Notation and Scaling:**
  - **Precision Management**: The module is designed to handle numerical data in scientific notation, allowing for a compact representation of large or small numbers without loss of precision.
  - **Adaptive Scaling**: Depending on the data type (e.g., voltages, currents) and the specified units (e.g., decibels for logarithmic scale results), the module scales the data appropriately. This feature is crucial for ensuring the data's usability in various analytical contexts where specific units or scales are required.

This module's design ensures that the results of the circuit analysis are documented in a manner that is both precise and adaptable to various needs, facilitating easy data management and analysis post-simulation.