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***

This module houses the `Circuit` class, which acts as a container for all the data related to the electrical circuit being analyzed. It structures the data and provides methods for manipulating it effectively. Below are elaborations on each significant method within this class:

- **Initialization:**
  - The `__init__` method initializes empty lists for storing components, outputs, and a dictionary for terminations. This setup prepares the class to receive and organize data as it is parsed from the input file.
- **Adding Components:**
  - `add_component`: This method takes parameters that define a component, such as type (resistor, capacitor), connection nodes, and value. A new instance of the respective component class is created and appended to the component list. Each component type class has an `ABCD` method that calculates its ABCD matrix based on the component's value and the operational frequency, facilitating modular and dynamic circuit analysis.
- **Setting Terminations:**
  - `set_termination`: Inputs termination values like source and load impedance. This method populates the termination dictionary, which is later used in the matrix calculations to incorporate boundary conditions of the circuit.
- **Adding Outputs:**
  - `add_output`: Specifies the outputs required from the circuit analysis, such as voltage at specific nodes or overall power. This method ensures that these specifications are stored and later used to determine what results are calculated and written to the output file.
- **ABCD Matrix Calculation:**
  - `calculate_abcd_matrix`: Iterates over the component list, invoking the `ABCD` method of each component to compute its matrix at the given frequency. This method handles the matrix multiplication of individual component matrices to form a comprehensive ABCD matrix representing the entire circuit at that frequency.
- **Circuit Solution Method:**
  - `solve`: This method is the core of the circuit analysis. It iterates over a range of frequencies, calls `calculate_abcd_matrix` for each frequency, and uses the resulting ABCD matrix to compute the circuit's response based on the terminations. The computed values for each requested output are formatted and passed to `csv_writer.py` for file output.
  - The efficiency of this method is crucial, as it impacts the overall execution time of the program. It ensures that computations are done in a logical sequence and are based on accurate mathematical foundations of network theory.

These methods collectively ensure that the `Circuit` class can manage data effectively, compute necessary parameters accurately, and facilitate the output of these parameters in a user-defined format. Each method is designed to be modular, allowing for ease of maintenance and potential future enhancements.