Cascade Circuit Analyser Design 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 document outlines the design approach used to solve the terminal inputs and outputs to a generalised two-port cascade circuit by method of ABCD matrix analysis. The program is designed to be modular, flexible and robust, with a thorough testing process embedded.**

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

CONTENTS

I Analysis of the Problem and Sub-tasks Identification 1
I-A **Parsing the input ".net" file** 1
I-B **Error handling for malformed inputs** 1
I-C **Matrix representation of components** 1
I-D **ABCD matrix calculations** 1
I-E **Generating output files** 1

II Module realisation and design decisions 2
II-A **Data Structures** 2
II-B **Functions and methods overview** . . . 2

III Testing Methodology 4
III-A **Python Doctests** 4
III-B **Regex and Error Handling Testing** . 4
III-C **Overall Program Testing** 4

IV Conclusion 4

I. Analysis of the Problem and Sub-tasks Identification

The assignment's goal is to create a comprehensive program for analyzing electrical circuits using the ABCD matrix method. This involves a detailed breakdown into manageable sub-tasks, each addressing a specific aspect of the overarching problem:

A. *Parsing the input ".net" file*

This step is foundational to the program's operation. By accurately extracting circuit components and parameters from the input file, it lays the groundwork for all subsequent analysis. Effective parsing requires a robust understanding of the expected file structure and the ability to handle variations and potential errors gracefully, ensuring that the program can adapt to real-world usage scenarios with varying input formats.

B. *Error handling for malformed inputs*

This sub-task is crucial for the program's reliability and user experience. It involves detecting and responding to errors in the input file, such as missing information, incorrect formats, or logically impossible configurations. By implementing comprehensive error handling, the program not only becomes more robust but also aids users in diagnosing and correcting their input files, thereby enhancing usability.

C. *Matrix representation of components*

The transition from a textual description of a circuit to a mathematical matrix representation is key to enabling the subsequent analysis. This process requires translate circuit configurations into their equivalent ABCD matrices consistently. The accuracy and efficiency of this representation directly impact the program's ability to provide precise analysis results in a reasonable time.

D. *ABCD matrix calculations*

This sub-task is at the heart of the program's functionality. Performing these calculations correctly is essential for solving the circuit and obtaining the desired outputs. It demonstrates the application of complex mathematical concepts and numerical methods, underscoring the program's core analytical capabilities. Optimizing these calculations for accuracy and performance is vital for handling a wide range of circuit configurations.

E. *Generating output files*

The final sub-task focuses on presenting the analysis results in a clear and accessible manner. This involves formatting the data into a specified CSV format, ensuring that the outputs are not only accurate but also matching the spec requested in the input file.

Each sub-task is intricately linked to the program's overall goal, showcasing a methodical and well-considered approach to solving the given problem. This detailed analysis and logical decomposition into sub-tasks highlight the program's thorough understanding and effective tackling of the circuit analysis challenge.

II. Module realisation and design decisions

This section breaks down the program into modules, classes and functions, and explains the design decisions made for each. The program is divided into four modules, each with a specific role:

- `net_parser.py`: Parses the input file and extracts circuit information.
- `circuit.py`: Contains classes and functions for circuit analysis.
- `csv_writer.py`: Manages the creation and formatting of the output file with analysis results.
- `main.py`: The main driver script that utilizes the above modules.

From a broad view, due to the varied nature of the input file, the program is designed to be modular and flexible in kind. The circuit module is designed as a class and subclass structure, with the main Circuit class containing the component array, the source and load terminations around the ABCD matrix, the matrix itself, and the output array. The subclasses are instances of the components, terminations, and output variables which are then appended to their respective arrays to keep track of their order. The main class has the methods to add to and sort these arrays, and to recusively reduce and invert the component matricies to obtain the ABCD termination solutions.

The ordering of output array is then used to generate the output CSV, and it is formatted to match the provided example output files.

A. Data Structures

This section outlines the primary data structures utilized within the main module (`main.py`), the circuit analysis module (`circuit.py`), and other modules such as the net file parser (`net_parser.py`) and the CSV writer (`csv_writer.py`).

Main Module *(`main.py`):* The main module orchestrates the program's execution flow, leveraging the following data structures:

- **Circuit Object:** An instance of the `Circuit` class from `circuit.py`, encapsulating the entire circuit's structure, including components and terminations.
- **Results array:** A list for the circuit's subclass of solved terminations for CSV output.

Circuit Module *(`circuit.py`):* Encompasses the entire model of the circuit and its methods. The module's primary data structures include:

- **Circuit Class:** Contains the circuit's comprehensive data, with arrays for instances of the component, termination, and output subclasses.
  - **Component List:** A list of component objects (resistors, capacitors, etc.), detailing type, value, and connections (e.g. {"n1": 1, "n2": 2, "value": 100, "type": "R"}).
  - **Terminations Dictionary:** Stores the source and load terminations parameters (e.g., {"VT": 5, "RS": 50}).
  - **Output Requirements List:** Lists the output variables as defined in the input file's order (e.g. {"name": Vin, "unit": V, "magnitude": m, "is_db": true}).
- **NumPy Arrays:** Employed for constructing and manipulating ABCD matrices, facilitating the numerical analysis of the electrical circuits. These arrays provide efficient handling of complex numbers and matrix operations essential for ABCD matrix calculations.

Net File Parser Module *(`net_parser.py`):* Responsible for interpreting the circuit definition file, it utilizes:

- **Regular Expressions (Regex):** For identifying component definitions and extracting pertinent information. This is especially useful as it can allow for optional fields,flexible use of whitespace and has extremely robust pattern matching, this also lets us define the correct form of a line for each section in the input file, playing nicely into exception raising when a line doesn't get a match.
- **Lists and Dictionaries:** To organize components, terminations, and output specifications for subsequent processing.

CSV Writer Module *(`csv_writer.py`):* Manages output file creation and formatting, employing:

- **Lists:** To compile analysis results for output variables across specified frequencies.
- **File Handler:** A standard object for writing formatted results to a CSV file/tempfile.
- **Magnitude Lookup:** A dictionary of magnitude multipliers for converting results to the desired units (e.g., "m": 1e-3, "u": 1e-6).

B. Functions and methods overview

This section provides an overview of the primary functions and methods within the program's modules, detailing their purpose and functionality.

Input File Parsing

The parsing of the input file is a critical initial step for the circuit analysis program, involving several key tasks to correctly interpret or sanatise the provided data for circuit components, terminations, and desired outputs. The process is outlined as follows:

1) **Reading the File:** Open and read the contents of the '.net' file line-by-line, paying special attention to lines starting with the hash symbol (#) as comments and thus skipping them during processing.
2) **Identifying Blocks:** Systematically identify the three main blocks within the file: CIRCUIT, TERMS, and OUTPUT, each of which provides essential data for the circuit analysis.
3) **Extracting Component Data:** Within the CIRCUIT block, extract details of each component, including node

connections and component values (resistance, inductance, capacitance, or conductance).
4) **Source and Load Termination:** From the TERMS block, determine the terminal connections provided for the source and load terminations, along with their respective values if available.
5) **Output Requirements:** In the OUTPUT block, capture the required output variables and formats, ensuring the program knows what results to calculate and how to format them.
6) **Error Handling:** Implement robust error handling to manage potential issues in the input file, such as missing/duplicate blocks, incorrect formatting, or unsupported component types/values.
7) **Pattern Matching:** Utilize regular expressions to identify and extract the required data from the input file, to help with resilience and flexibility in handling different input file formats.
   This requires that the cases where the output is in decibels or inputs/outputs with exponent prefixes are handled and stored for the extension tasks.
8) **Storing Data:** Efficiently store the extracted information in the Circuit class module, to the respective component, termination, and output sub-classes.

This structured approach ensures that the program can flexibly interpret and process the wide range of circuit definitions it may encounter.

Circuit Realization and Solving

To realize and solve the electrical circuit defined in the input file, a systematic approach towards constructing and analyzing the circuit using the ABCD matrix method is as follows:

1) **Circuit Representation:** Construct an internal representation of the circuit from the parsed data, an array of component objects, and the source and load terminations. This can then be sorted based on the node connections to facilitate the ABCD matrix construction.
2) **Matrix Construction:** Implement functions to calculate the ABCD matrices for individual circuit elements (resistors, inductors, capacitors, and any series or parallel combinations thereof).
3) **Cascade Handling:** A method to create a new array of 2x2 sub-ABCD matricies from the components object array, allowing for the multiplication reduction down to an individual ABCD matrix to represent the entire circuit's behavior.
4) **Termination Analysis:** Apply the source and load terminations as defined in the TERMS block to the overall circuit matrix to determine input/output impedance and thus Vin, Iin, Vout, and Iout. This requires that there are different methods for if the source is of Thevenin or Norton type to determine Vin/Iin.
5) **Solving the Circuit:** Utilize the inversion of the ABCD matrix of the whole circuit to solve for the desired output variables such as input/output impedance, voltage gain, and power gain.
6) **Error and Exception Handling:** Implement comprehensive error checking and exception handling to print an empty output file with an error message if the circuit analysis fails at any stage.
7) **Result storage:** Repeat the above and store the results for each frequency iteration in an array, to be passed to the CSV writer for output file generation.

Output File Generation

The final step in the program is to generate the output file with the calculated results. This involves the following steps:

1) **CSV File Creation:** Create a new CSV file with the specified output filename and write the header row with the required output variables names and units.
   This also requires that the cases where the output is in decibels or with exponent prefixes are handled, and that the order of the headers matches the output array order.
2) **Data Writing:** Write the calculated results to the CSV file, ensuring the order (as specified in the OUTPUT block and stored in the output array), and the formatting (decibels, exponent prefixes) in scientific notation, are correctly applied.
3) **File formatting:** Improve the readability of the file by padding the columns to be equal widths and aligning the values to one side, matching the provided example output files.
   This function should be able to operate on any CSV file.
4) **Error Handling:** Implement error handling to write an empty output file with an error message if the output file generation fails at any stage.

Extension Tasks

All of the extension tasks rely on the input file parsing being versatile enough to encompass the additional features, the approach to if the output is in dB or has additional exponent prefixes will be to store that to the output array and then convert the results to dB + Phase in the CSV writer and/or apply the exponent prefix.

Similar to determining if the source is of Thevenin or Norton type, the program will need to differentiate between if the frequency sweep is linear or logarithmic, which can also be done with a simple check of which sufficient values were provided by the input.

As for input values, the exponent will be converted on the spot with the input parser, this means that internally, all calculations are standardised to the base unit for consistency.

For all the above tasks, the input parser will pick them all up with regular expressions, these parameters being optional and the program will be able to handle them if they are present or not.

III. Testing Methodology

Effective testing is pivotal to software development, ensuring functional integrity and reliability. Our approach integrates Python doctests for precise function validation and manual testing to address complex error handling and regex patterns, targeting the core functionalities of the electrical circuit analysis program.

A. *Python Doctests*

Doctests offer an integrated testing framework within Python's docstrings, facilitating immediate verification of critical functions, such as mathematical computations and data manipulation. This method is applied extensively within the `circuit.py` and `net_parser.py` modules to guarantee the accuracy of:

1) ABCD matrix calculations, confirming correct mathematical operations across various circuit components.
2) Component parsing logic, ensuring reliable extraction of component data from input strings.

Automatically executed during documentation review, doctests provide a streamlined, effective means of ensuring code correctness.

B. *Regex and Error Handling Testing*

Acknowledging the diversity and complexity of potential inputs, our manual testing regime is meticulously designed to challenge the program's regex patterns and error handling capabilities. Through tailored example input files, this phase aims to:

1) Verify the fidelity of regex patterns in accurately parsing component information under variable conditions.
2) Assess the program's ability to elegantly manage errors, such as unsupported component types or formatting discrepancies, enhancing user experience through informative feedback.

This deliberate, manual inspection is indispensable for validating the robustness of input processing and error management mechanisms within the program.

C. *Overall Program Testing*

Comprehensive program validation is conducted via `AutoTest.py`, comparing actual outputs against expected results using standard test files. This ensures not only the functional coherence of the program but also the correctness and format of its outputs. Exception handling is rigorously implemented to manage anomalies, such as invalid inputs, by writing an empty CSV file as a fail-safe measure.

IV. Conclusion

By marrying doctests with targeted manual evaluations, our testing methodology offers a holistic verification framework, addressing both micro-level functionality and macro-level program integrity. This balanced approach not only assures the reliability and accuracy of the electrical circuit analysis tool but also underscores its modularity, flexibility, and robustness. The result is a user-centric, dependable application poised to support complex circuit analysis tasks.