Cascade Circuit Solver - Design Document

Eugene Levinson (el769)

March 2024

Table of Contents

Project Overview and Definition of Done

The goal of this cascade circuit solver is run a simulation of a circuit and output the results. The program must read the input file as specificed in the project definition file. Correctly interprite the circuit to be analaysed from the input file, and other meta data. Then correctly calculate the results of the circuit and output the results to the output file.

The project is considered done when all of the following are met:

1. The program can be launched from the command line in the following manner: `python app.py <input file> <output file>`
2. The program can read and create correct outputs for all test cases provided. This includes circuits of varying complexity and size.
3. The program produces and empty output if the input file is not valid.
4. All parts of the program are covered by unit tests.

Design Overview

The program is designed to be modular, with each part of the program being implemented as a separate class (where it makes sense). This will allow for easy testing and debugging. The program will be implemented in a test-driven manner, with unit tests being written before the implementation of the program.

The overall program flow will be as follows:

1. The program is launched from the command line with 2 arguments - input file and output file.
2. The command line arguments are parsed. This should be implemented as a separate function, so that it can be easily tested. The cli parser should check if the input file exists and if the output file is writable. If not, the program should exit with an error message.
3. The CircuitSimulation class is the instantilised. This is the main interface of the program.
4. The input file is parsed using a `parse` method of the `CircuitSimulation` class. During parcing several atributes of the `CircuitSimulation` class are set. The **CIRCUIT**, **OUTPUT** and **TERMS** blocks are indentified. Each block is parsed seperately and the data is stored in the `CircuitSimulation` class. The **CIRCUIT** block is parsed into a graph data structure, which is used to represent the circuit, where each node is an instance of a `Component` class. The **OUTPUT** block is parsed into a list of dictionaries. The **TERMS** is parsed and `Source` and `Load` classes are instantiated.
5. The circuit is solved using the `solve` method of the `CircuitSimulation` class. The circuit is solved with matrix multiplication and linear algebra using the "Chain Matrix Analysis" method. Numpy is used to perform all maths operations to increase performance.
6. The solving step maybe be repeated several times for different frequnecy and paramattric sweeps. The results are stored in the `CircuitSimulation` class.
7. The results are formatted and written to the output file with the `write_output` method of the `CircuitSimulation` class.
8. The program exits.

Design Implementation

CLI arguments parsing

A function will be implemented to parse the command line arguments and validate them.

The function will take in a configuration that defines the expected command line arguments - `input_file` and `output_file`. The function will return the input and the output file paths. If one of the arguments is missing, the program will return `None` for the missing argument.

Logic downstream will check that both paths are present and raise an error if not.

`CircuitSimulation` class

The `CircuitSimulation` class is the main interface of the program and is responsible for the overall program flow. The constructor of the class will have the following signature:

```
def __init__(self, input_file: str = None):
    pass
```

The input file is optional, as the class can be instantiated without an input file. The input file can be passed in later to the `parse` method of the class. The output file can be passed in as an argument to the `write_output` method of the class.

The class will have the following methods main methods:

- `__init__` - constructor
- `parse` - parse the input file
- `identify_blocks` - identify the **CIRCUIT**, **OUTPUT** and **TERMS** blocks in the input file
- `parse_circuit` - parse the **CIRCUIT** block
- `parse_output` - parse the **OUTPUT** block
- `parse_terms` - parse the **TERMS** block
- `convert_units` - method to convert between different units.
- `solve` - solve the circuit
- `write_output` - write the results to the output file

Other methods will be implemented as needed to further abstract the program and make it more modular.

The class will have the following attributes:

- `input_file` - the input file path
- `circuit` - a graph data structure representing the circuit
- `required_outputs` - a list of dictionaries representing the output block
- `source` - an instance of the `Source` class
- `load` - an instance of the `Load` class

`Component` Class

The `Component` class represents each component in the circuit. The component class will be a base class for all other component classes. The class will mainly be responsible for storing the component's attributes and methods to convert between different units. For example, the `Resistor` class will inherit from the `Component` class and will have a method to convert resistance from ohms to kiloohms. It will also contain the matrix representation of the component.

The class will have the following methods:

- `__init__` - constructor
- `convert_units` - method to convert between different units.

The class will have the following attributes:

- `base_unit` - the base unit of the component
- `value` - the value of the component
- `abcd_matrix` - the matrix representation of the component

`Source` Class

The `Source` class represents the source in the circuit. The class will be a base class for `Thevenin` and `Norton` classes.

`Load` Class

The `Load` class represents the load in the circuit. It will store the information about the load.

Use of `Source` and `Load` classes

The classes will be used in calculating the input and output values of the circuit. Depending on the type of the source, slighly different calculations will be performed. The class interface will abstract the calculations the downstream logic work irrespective of the source type.

Testing

Two main strategies will be used for testing the program - unit tests and integration tests. All testing will be done using the `pytest` library.

Unit Tests

Unit tests will be written for each function and class in the program. The tests will define the expected behaviour of the function or class, including all edge cases. The implementation of the function or class will then be written to satisfy the tests. This test driven approach will ensure that the program is implemented correctly and that all edge cases are handled. The tests will check that a given input produces the expected output. This will insure that if the implementation of the function or class changes, there will be a method to verify that the new implementation will produce appropriate results and fit the expected behaviour.

Integration Tests

Integration tests will be written to test the program as a whole. The tests provided on moodle can be considered as integration tests and will be used as such.

Main focus of testing

There are three critical cases that the test will focus on:

- The input is the "ideal" case - this will test the functionality of the program when the input file is valid and the program is able to solve the circuit and produce the correct output.
- The input is valid but is a variation of the ideal case, for example the units are different or the circuit is more complex. This will test the robustness of the program and ensure that it can handle different inputs.
- The input is invalid, missing or otherwise - this will test that the program produces an empty output file when the input file is invalid.

Error Handling

The program will use a "fail fast" approach, where the program will exit as soon as an error is encountered. This is to prevent the program from continuing to run in an invalid state, which could lead to further errors and make debugging more difficult. In the cases of errors, an empty output file should be created to comply with the project definition.

Key unit tests

The following arethe key unit tests that will be written for the program. However, this is not an exhaustive list and more tests will be written as needed:

- Testing of the CLI interface
- Testing of the `parse` method of the `CircuitSimulation` class.
- Testing of the `identify_blocks` method of the `CircuitSimulation` class

- Testing of the `parse_circuit` method of the `CircuitSimulation` class
- Testing of the `parse_output` method of the `CircuitSimulation` class
- Testing of the `parse_terms` method of the `CircuitSimulation` class
- Testing of the `convert_units` method of the `CircuitSimulation` class
- Testing of the `solve` method of the `CircuitSimulation` class
- Testing of the `write_output` method of the `CircuitSimulation` class

Style and organisation

The program will be written in a modular and object-oriented manner. Each part of the program will be implemented as a separate class or function. This will allow for easy testing and debugging.

Classes and function will be documented with doc strings and type hints. The program will be written in a PEP8 compliant manner.

The program will be organised into several files and directories to make it more modular and easier to navigate. However, this will be adjusted for the final submission if the program is required to be in a single file.