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 of approach

The program largely suck to the design specification, moulding and deviating slightly for improvements in performance or consiceness. The program is split into four main modules.

A. main.py

This module is the entry point for the program. It handles the command line arguments and calls the main function within a try-except block to catch any exceptions that may be raised further down the stack. In the event of any exceptions, the program will print the error message, write an empty CSV file with the output file argument and exit with a non-zero exit code.

B. net_parser.py

This module contains one main function to identify what section each line belongs to, and three helper functions for line processing. The `parse_net_file_to_circuit` function reads the netlist file line by line, stripping them of whitespace, then ignoring empty lines and comments. It sets the current section by matching the line to the delimiters, erroring if the preceeding section is not correct. Lines that are not delimiters are then matched to their respective section and processed by the helper functions. The helper functions process the line with a regex pattern specific to each section, returning the relevant data as dictionaries and adding them to the circuit object for components, terminations and outputs.

C. circuit.py

This module contains the Circuit class which is responsible for storing the circuit data in a structured manner. This is achieved with component and output nested sub-classes which contain the relevant data for each component and output. The Circuit class then contains a respective list for each of these classes so that their order is preserved for iteration and a dictionary for the termination values. The circuit class has methods to help with setting and adding components, outputs and terminations.

The Component sub-class contains a method to calculate the ABCD matrix for the component at a given frequency.

The core method of this class is the `solve` method which first sorts the component list by the nodes of the components, opens the output file and writes the header row based off the output list. It then iterates through the frequency list, generating a list of ABCD matrices from each component and multiplying them together to reduce the circuit to a single ABCD matrix. This matrix is then used to calculate the termination values at that frequency, where they are subsequently stored in the termination dictionary and a row is written to the output file. Once all frequencies have been iterated through, the output file is closed by the context manager.

D. csv_writer.py

This module contains three simple functions to write the header rows, data rows and an empty CSV in the event of an error. The first column is always the frequency, which is padded to 10 characters with whitespace to the left, all other columns are then padded to 11 characters. Each row is stored as a buffer before writing, then data is formatted in scientific notation to 3 decimal places and appended to the buffer. The buffer is then written to the file.

The order of the columns is determined by the order of the output sub-class list in the circuit class, the header changes based on if the output is in decibels or not. Data is scaled by the magnitude and differentiates between values that use 10 or $20 * Log_{10}$, all to match specification.