2024

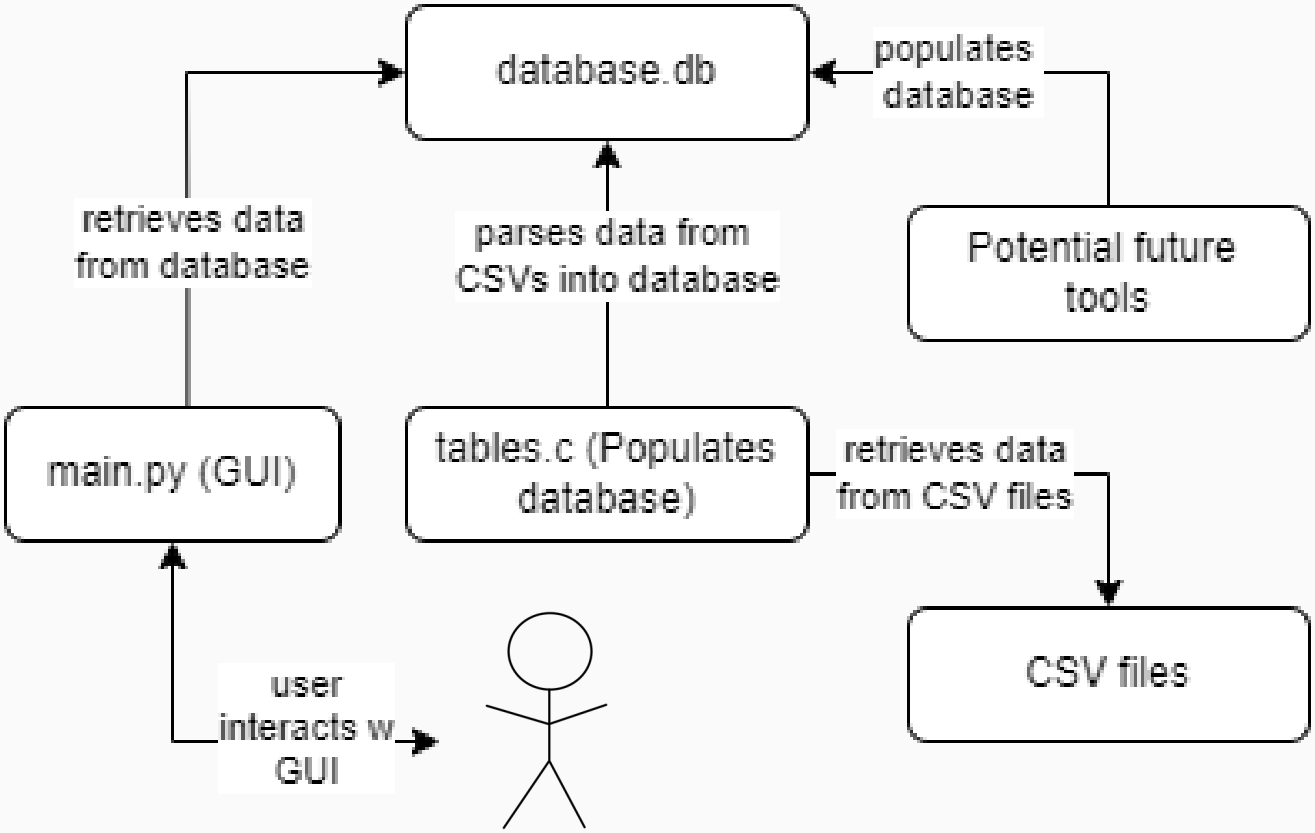
STEADY STATE SCREENING TOOL

Manitoba Hydro - Grid Infrastructure Planning

Created by Amy Ding

TABLE OF CONTENTS

1. INTRODUCTION



The user runs the GUI in Visual Studio 2022. The GUI retrieves data from the database file. This database file was previously populated using CSV files and the SQLite C interface.

2. SET UP

REQUIRED SOFTWARE

Python

Download Python here.
This app was created using Python 3.12

Visual Studio 2022

Download the Community version here.

SQLite

Download the precompiled binary here.

ADDITIONAL LIBRARIES

PyQt6 (documentation): Create the graphical user interface

To download, open command prompt and enter the following command:

```
pip install PyQt6
```

Official download instructions found here.
After installing PyQt6, users also need to manually install aqt[qt6] with the following command:

```
pip install aqt[qt6]
```

aqt should also automatically install the PyQt6.QtWebEngineWidgets module. If this was not automatically installed for whatever reason, it can be manually installed using the following command:

```
pip install PyQtWebEngine
```

Documentation is found here.

PyLaTeX (documentation): Create LaTeX formatted reports as PDF and .tex documents

To download, open command prompt and enter the following command:

```
pip install pylatex
```

Official download instructions found here.

pypandoc (documentation): Displays LaTeX document as HTML preview

To download, here are two different methods:

Method 1:
Open command prompt and enter the following command:
```
pip install pypandoc_binary
```
Then you must manually add pandoc to the path (instructions for this step can be found in method 2 below).
Official download instructions found here.

Method 2:
1. Download the pandoc installer here
2. Run the installer .msi file to start the installation process.
3. Verify Installation by opening command prompt and typing the following command:
   ```
   pandoc --version
   ```
   - If Pandoc is installed correctly, you should see the version number displayed, indicating that Pandoc is now available in your system's PATH.

Optional: Add Pandoc to PATH (if not automatically added):
- If pandoc --version doesn't work after installation, you may need to add Pandoc to your system's PATH manually:
  - Right-click on "This PC" or "Computer" on your desktop or in File Explorer.
  - Select "Properties" > "Advanced system settings" > "Environment Variables".
  - In the "System variables" section, find the "Path" variable and click "Edit".
  - Add the path to the Pandoc installation directory (usually C:\Program Files\Pandoc\ or C:\Program Files (x86)\Pandoc\ for 64-bit and 32-bit installations respectively) to the list of paths. Separate paths with a semicolon if adding multiple paths.

dirent.h (documentation): Allows directory functions in C

The dirent.h header file should already be included in the Visual Studio project.
If not, you can find the official download instructions here.

Overview of
REQUIREMENTS

The files must have the following columns in the following order, or else the code will not work. There will also be an exception column after each violate column (not pictured). Some column names have also been mapped to the database table column, for clarity.

	A	B	C	D	E	Bus Base	Low Voltage	High Voltage	I	J	K	Bus Voltage	M	N
1	bus_num	bus_name	bus_area	bus_zone	bus_owner	bus_basekv	criteria_nlo	criteria_nhi	criteria_elo	criteria_ehi	stat	bus_pu	bus_angle	violate

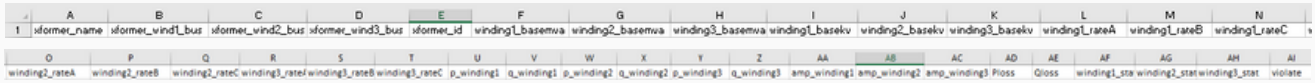
Voltage file format

	A	B	C	D	Voltage Class	Rating	G	H	I	J	K	L	M	N	Metered End	Other End	Q	R	S	T
1	branch_ni	branch_m	branch_ot	branch_id	branch_basekv	rateA	rateB	rateC	p_metere	p_other	q_metere	q_other	amp_angle_m	amp_angle_other	amp_metered	amp_other	ploss	qloss	stat	violate

Thermal branch file format

	A	B	C	D	MVA Base(W1&W2)	Winding 1 KV	Winding2 KV	Rating W1	I	J	Rating W2	L	M	N	O	P	Q	Loading W1	Loading W2	T	U	V	W
	xformer_name	xformer_wind1_bus	xformer_wind2_bus	xformer_id	xformer_basemva	winding1_basekv	winding2_basekv	winding1_rateA	winding1_rateB	winding1_rateC	winding2_rateA	winding2_rateB	winding2_rateC	p_winding1	q_winding1	p_winding2	q_winding2	amp_winding1	amp_winding2	Ploss	Qloss	stat	violate
	T9_3		667002	669718	9	120	230	13.80000019	401.6349792	401.6349792	401.6349792	6693.916504	6693.916504	6693.916504	-122.6933212	-48.31327057	123.036026	60.96306992	327.4661255	5594.212891	0.342692	12.64982		1

2-winding transformer file format



	A	B	C	D	E	F	G	H	I	J	K	L	M	N
1	xformer_name	xformer_wind1_bus	xformer_wind2_bus	xformer_wind3_bus	xformer_id	winding1_basemva	winding2_basemva	winding3_basemva	winding1_basekv	winding2_basekv	winding3_basekv	winding1_rateA	winding1_rateB	winding1_rateC

	O	P	Q	R	S	T	U	V	W	X	Y	Z	AA	AB	AC	AD	AE	AF	AG	AH	AI
	winding2_rateA	winding2_rateB	winding2_rateC	winding3_rateA	winding3_rateB	winding3_rateC	p_winding1	q_winding1	p_winding2	q_winding2	p_winding3	q_winding3	amp_winding1	amp_winding2	amp_winding3	Ploss	Qloss	winding1_sta	winding2_stat	winding3_stat	violate

3-winding transformer file format

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
1	facility_name	bus_num	gen_id	gen_owner	bus_basekv	criteria_nlo	criteria_nhi	criteria_elo	criteria_ehi	pmin	pmax	qmin	qmax	remote_bus	schdvolt	stat	pout	qout	terminalvolt	remote_voltage	violate

Generator file format: (note: OOS name column in the database is currently being filled by facility_name from excel).

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	X	Y	Z	AA	AB	AC
1	oos_name	oos_monitored_end	oos_other_en	oos_clinic	oos_remote_en	oos_model	data_in_	data_in_	data_in_	data_in_	data_in_	data_in_	data_out_	data_out_	data_out_	data_out_	data_out_	data_out_	criteria_in_ssp	criteria_out_ssp	criteria_in_sspo	criteria_out_sspo	stat	oos_i	oos_n	angle_monitored	angle_remote	margin	violate

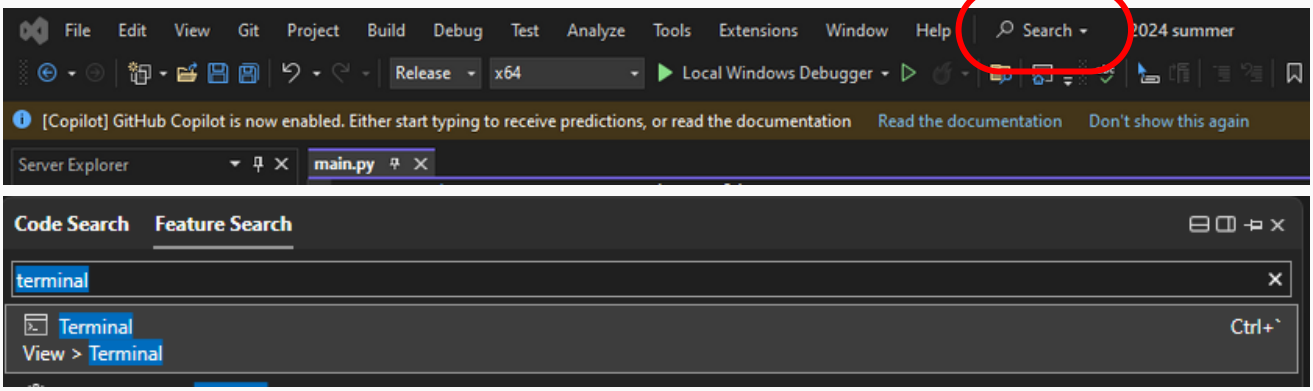
OOS file format

3. HOW TO USE

RUNNING THE GUI

1. Open the project in Visual Studio 2022

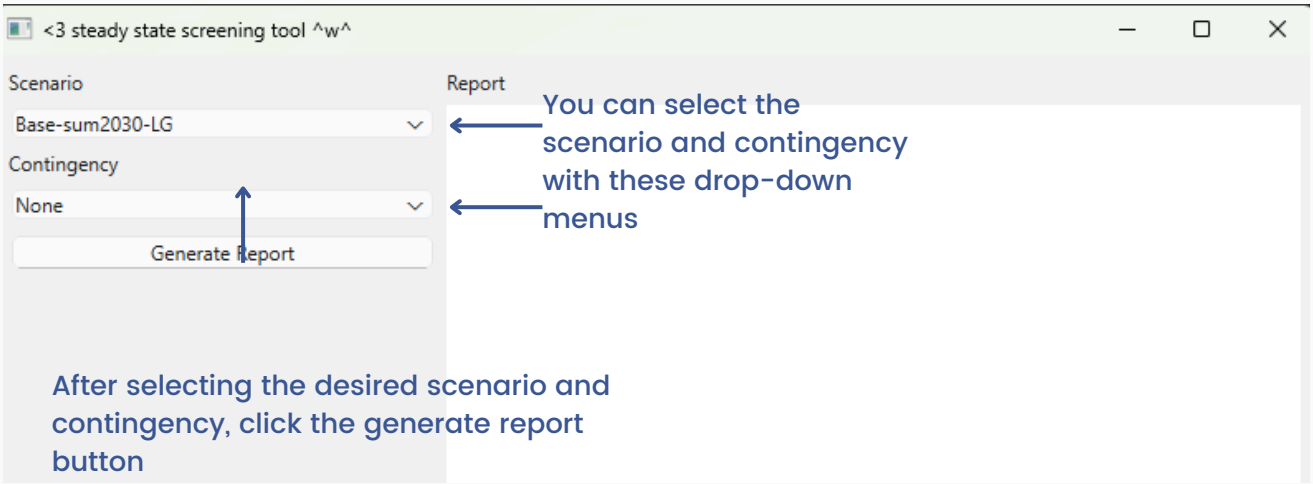
2. Select Search on the top bar and search Terminal



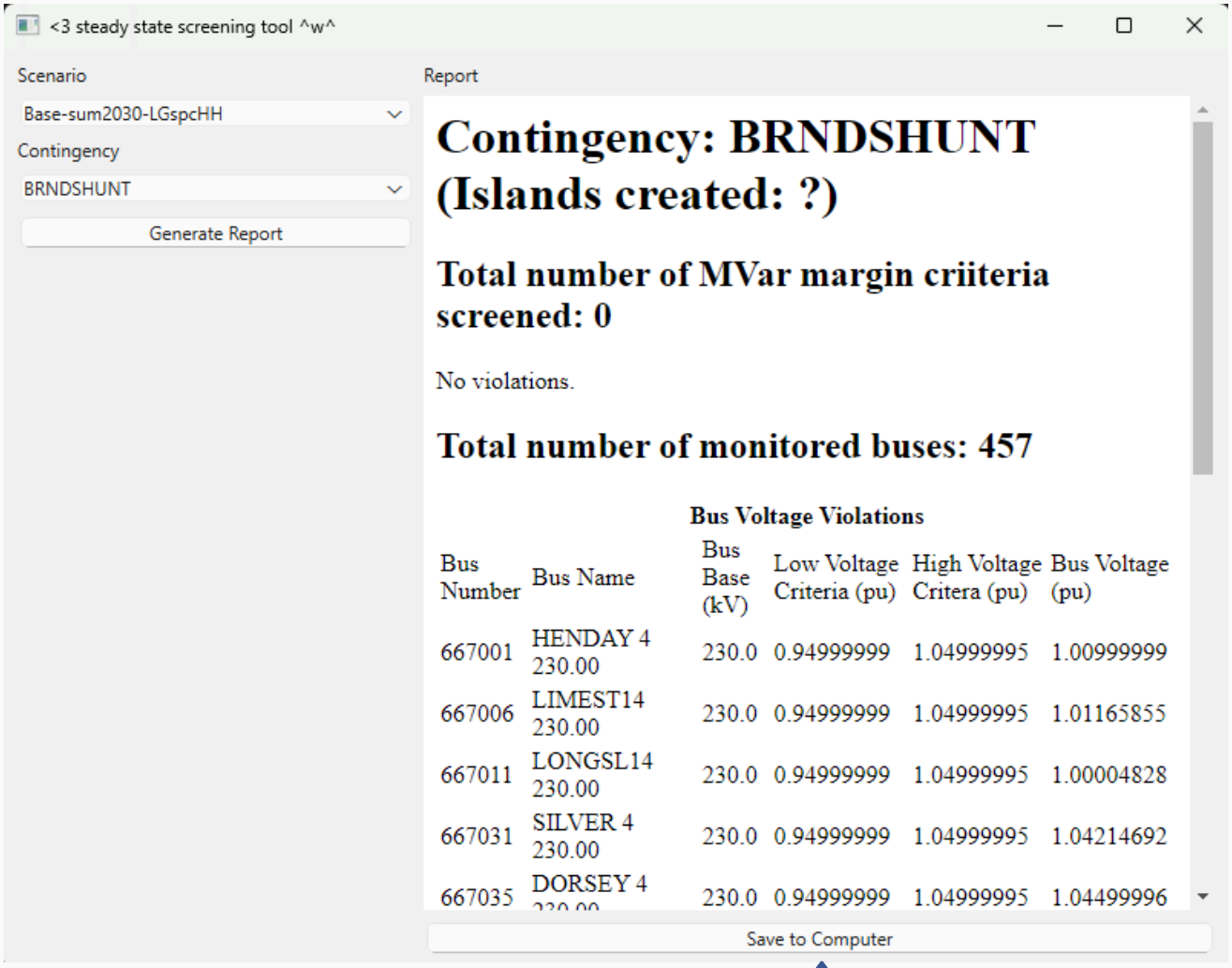
3. Run the desktop app with the following command

Ensure that you are in the same directory as where the project file is located.

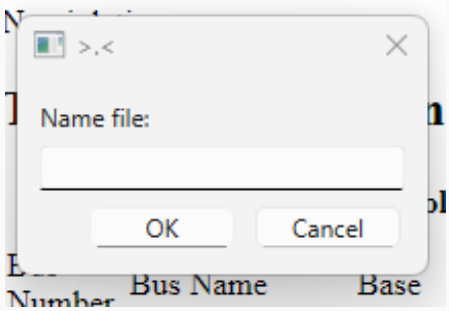
4. The desktop app should start running



5. You will see the report displayed on the left.

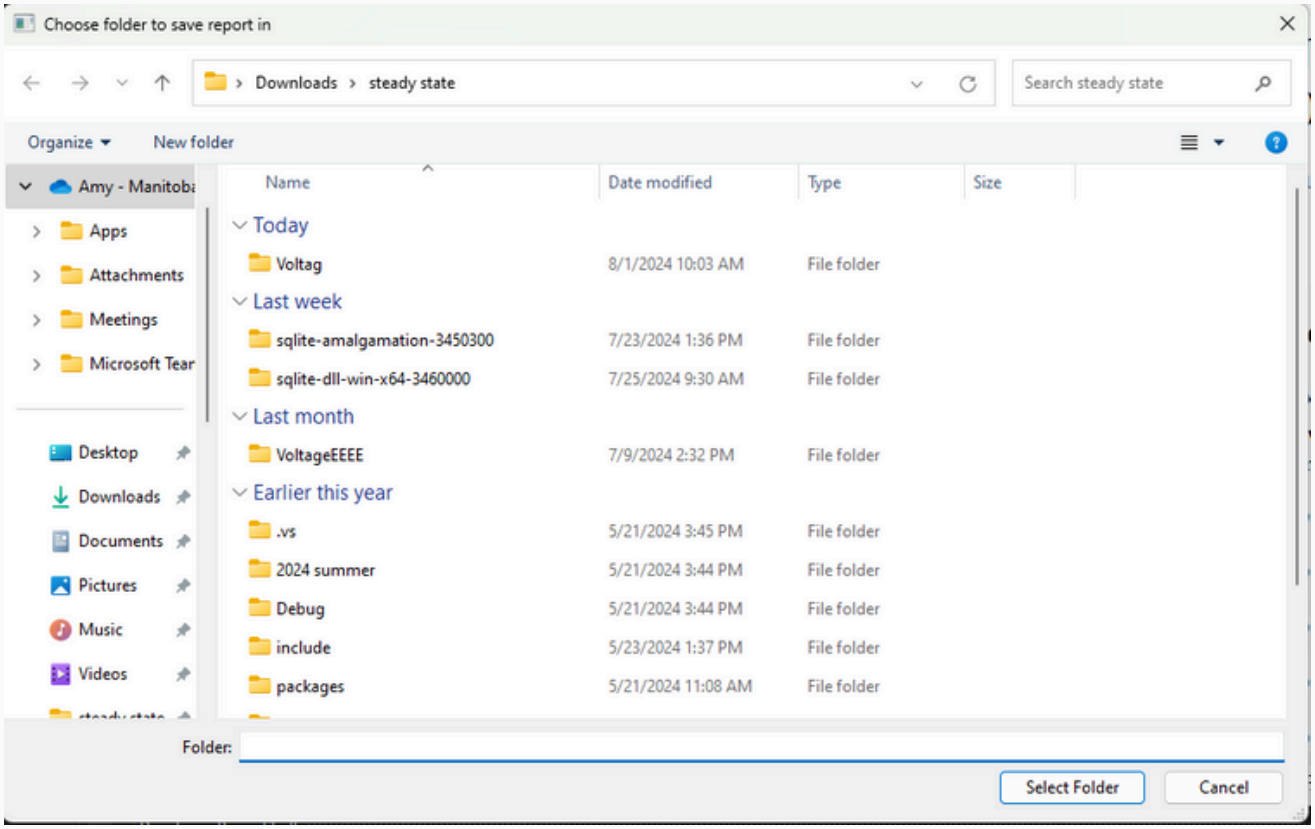


If you wish to save the report to your computer, then click the “Save to Computer” button

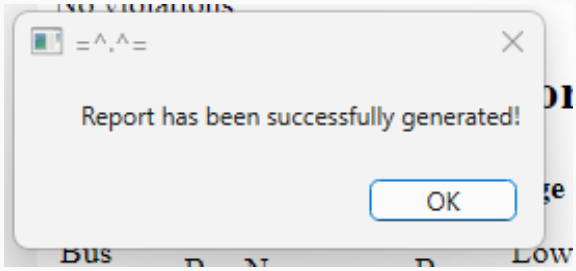


A dialog will pop up, and you can input the name you want the file to be saved as. Click OK when you have entered the name, and cancel if you do not want to save the file anymore.

6. You will be prompted to choose the location where you want to save the file



Click this button once you have selected the folder that you want.



Once this dialog window shows up, the report is done generating. It may take from 5-10 seconds. You may now view and/or download other reports.

POPULATE/UPDATE DATABASE

You will need to do this the first time you run the GUI, and after each time any files are changed or added.
Please refer to this video to understand how to add SQLite database connections.
Note: Visual Studio project should be located on the same level as the file folders.

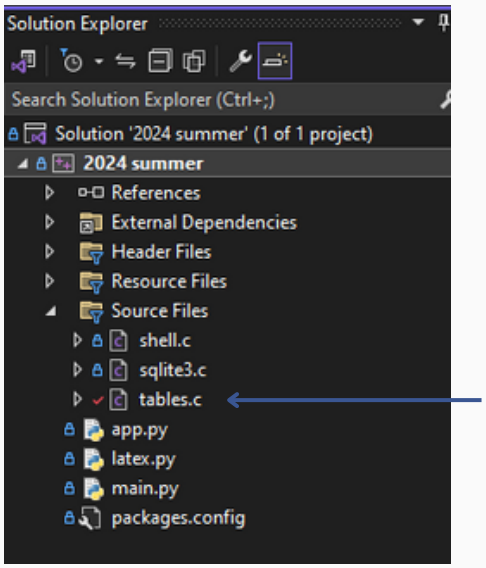
```
└──── SummerStudent2024/
    ├──── 2024summer.vcxproj
    ├──── Voltage
    ├──── ThermalBranch
    ├──── 2winding
    ├──── 3winding
    ├──── Generator
    └──── OOS
```

IMPORTANT NOTE 8/22/2024: Currently some of the file names are incorrect in that they do not follow the convention mentioned on this page. Steve did say he would fix this in his tool at some point. The issue is that some of them are missing the '@' symbol, so this will cause the program to crash.

1. Open the project in Visual Studio 2022

2. Open the tables.c file

If you do not see it on the right hand side, you can do View > Solution Explorer



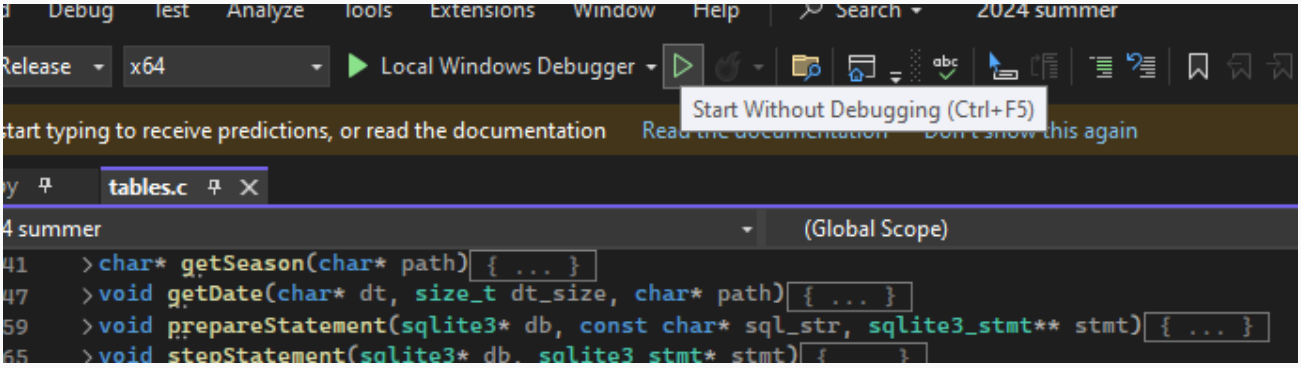
3. Go to the main function (at the bottom of the file) and add the command that you want

The first time you run it, you will want to do repopulateTables. This will take a very long time, a couple hours at least. For anything else, such as just adding or changing 1 or more files, updateTables will be sufficient and will only take a few minutes. You can also change the name of the database file at the top. Below, repopulateTables is commented so we are just updating the tables. The majority of the time, you will want to update the tables.

```
int main(int argc, char* argv[]) {
    char* file = "database2.db";
    sqlite3* db = NULL;
    int rc = 0;
    sqlite3_initialize();
    rc = sqlite3_open_v2(file, &db, SQLITE_OPEN_READWRITE | SQLITE_OPEN_CREATE, NULL);

    if (rc != SQLITE_OK) {
        handle_error(db, "Cannot open database");
    }
    //repopulateTables(db);
    updateTables(db);
    sqlite3_close(db);
    return 0;
}
```

4. Run it with the button at the top



4. BACKEND

The code to populate the SQLite database can be found in tables.c. This code is written in the C programming language and utilizes the SQLite C interface.

Overview of TABLES.C

GLOBAL VARIABLES

Specifies the path of the folders. If the folder names or directory structure is changed, you can change the path here to ensure that the code will be looking in the right place. Currently assumed that the project files will be on the same level as the folders containing the data.

```
char* VOLTAGE_FOLDER = "./Voltage";
char* THERMALBRANCH_FOLDER = "./ThermalBranch";
char* THERMAL2_FOLDER = "./thermal2winding";
char* THERMAL3_FOLDER = "./3-winding";
char* GENERATOR_FOLDER = "./generator";
char* OOS_FOLDER = "./oos";
```

The db variable is found in the main function. Can be changed depending what file you want the new tables to be stored in.

```
char* db = “database.db”
```

void handle_error(sqlite3* db, const char* msg)

Prints any error messages, closes database, and exits the program with exit code -1.
Args:

- sqlite3* db: sqlite3 database structure
- const char* msg: error message description

char* getScenario(char* path)

Returns the scenario of a given file as a string
Args:

- char* path: path of the file that you want to get the scenario from

char* getContingency(char* path)

Returns the contingency of a given file as a string
Args:

- char* path: path of the file that you want to get the contingency from

char* getSeason(char* path)

Returns the season of a given file as a string
Args:

- char* path: path of the file that you want to get the seasonfrom

void getDate(char* dt, size_t dt_size, char* path)

Returns the last modified date of a given file as a string
Args:

- char* dt: string buffer to store the date in
- size_t dt_size: size of string buffer
- char* path: path of the file that you want to get the date from

Usage:

```
char date[100];
getDate(date, sizeof(date), filePath);
```

void prepareStatement(sqlite3* db, const char* sql_str, sqlite3_stmt** stmt)

Prepares a given SQL statement for execution. SQL statements need to be prepared before they are executed.
Args:

- sqlite3* db: sqlite3 database structure
- const char* sql_str: string containing the SQL statement
- sqlite3_stmt** stmt: sqlite3 statement structure that will be bound to the SQL string

Usage:

```
sqlite3_stmt* stmt = NULL;
char* sql_str = "SELECT 3 * 4;“
prepareStatement(db, sql_str, &stmt);
```

void stepStatement(sqlite3* db, sqlite3_stmt* stmt)

Executes a prepared SQL statement
Args:

- sqlite3* db: sqlite3 database structure
- sqlite3_stmt** stmt: prepared sqlite statement

Usage:

```
sqlite3_stmt* stmt = NULL;
char* sql_str = "SELECT 3 * 4;“
prepareStatement(db, sql_str, &stmt);
stepStatement(db, stmt);
```

void processBusFile(sqlite3* db, FILE* file, sqlite3_stmt* statements[], char* scenario, char* contingency, char* date)

Processes data in the bus files and inserts into database
Args:

- sqlite3* db: sqlite database structure
- FILE* file: bus file to process
- sqlite3_stmt** statements[]: array of relevant sqlite statements
- char* scenario: associated scenario with file
- char* contingency: associated contingency with file
- char* date: associated last modified date with file

void processBranchFile(sqlite3* db, FILE* file, sqlite3_stmt* statements[], char* scenario, char* contingency, char* season, char* date)

Processes data in the branch files and inserts into database
Args:

- sqlite3* db: sqlite database structure
- FILE* file: branch file to process
- sqlite3_stmt** statements[]: array of relevant sqlite statements
- char* scenario: associated scenario with file
- char* contingency: associated contingency with file
- char* date: associated last modified date with file

void processT2File(sqlite3* db, FILE* file, sqlite3_stmt* statements[], char* scenario, char* contingency, char* date)

Processes data in the 2-winding transformer files and inserts into database
Args:

- sqlite3* db: sqlite database structure
- FILE* file: 2-winding transformer file to process
- sqlite3_stmt** statements[]: array of relevant sqlite statements
- char* scenario: associated scenario with file
- char* contingency: associated contingency with file
- char* date: associated last modified date with file

void processT3File(sqlite3* db, FILE* file, sqlite3_stmt* statements[], char* scenario, char* contingency, char* date)

Processes data in the 3-winding transformer files and inserts into database
Args:

- sqlite3* db: sqlite database structure
- FILE* file: 3-winding transformer file to process
- sqlite3_stmt** statements[]: array of relevant sqlite statements
- char* scenario: associated scenario with file
- char* contingency: associated contingency with file
- char* date: associated last modified date with file

void processGeneratorFile(sqlite3* db, FILE* file, sqlite3_stmt* statements[], char* scenario, char* contingency, char* date)

Processes data in the generator files and inserts into database
Args:

- sqlite3* db: sqlite database structure
- FILE* file: generator file to process
- sqlite3_stmt** statements[]: array of relevant sqlite statements
- char* scenario: associated scenario with file
- char* contingency: associated contingency with file
- char* date: associated last modified date with file

void processOOSFile(sqlite3* db, FILE* file, sqlite3_stmt* statements[], char* scenario, char* contingency, char* date)

Processes data in the OOS files and inserts into database
Args:

- sqlite3* db: sqlite database structure
- FILE* file: OOS file to process
- sqlite3_stmt** statements[]: array of relevant sqlite statements
- char* scenario: associated scenario with file
- char* contingency: associated contingency with file
- char* date: associated last modified date with file

void processFile(sqlite3* db, char*path, char* filename, sqlite3_stmt* statements[], int type)

Processes each file by calling the appropriate process{Specific}File() function
Args:

- sqlite3* db: sqlite database structure
- char* path: path of file to be processed

- char* filename: name of file
- sqlite3_stmt** statements[]: array of relevant sqlite statements
- int type: type of file to be processed and type of processing

type	function
0	bus file, immediately parsing
1	branch file, immediately parsing
2	2-winding transformer file, immediately parsing
3	3-winding transformer file, immediately parsing
4	generator file, immediately parsing
5	oos file, immediately parsing
10	bus file, checking for updates before parsing
11	branch file, checking for updates before parsing
12	2-winding transformer file, checking for updates before parsing
13	3-winding transformer file, checking for updates before parsing
14	generator file, checking for updates before parsing
15	oos file, checking for updates before parsing

void traverseDirectory(sqlite3* db, char*path, sqlite3_stmt* statements[], int type)

Traverses every file and subdirectory in a given directory, and calls processFile() on csv files
Args:

- sqlite3* db: sqlite database structure
- char* path: path of directory to be processed
- sqlite3_stmt** statements[]: array of relevant sqlite statements
- int type: type of file to be processed and type of processing

void checkForUpdates(sqlite3* db, FILE* file, sqlite3_stmt* statements[], char* scenario, char* contingency, char* season, char* date, int type)

Checks for updates in each folder
Args:

- sqlite3* db: sqlite database structure
- FILE* file: file to process
- sqlite3_stmt** statements[]: array of relevant sqlite statements
- char* scenario: associated scenario with file
- char* contingency: associated contingency with file
- char* season: associated season with file
- char* date: associated last modified date with file
- int type: number that specifies type of file

void populateScenCont(sqlite3* db, const char* path)

Populates scenario and contingency tables using the files in the given directory
Args:

- sqlite3* db: sqlite database structure
- char* path: path of directory to be processed

void populateBusTables(sqlite3* db, char* path)

Prepares SQL statements to input data into tables. Calls traverseDirectory() for the bus files
Args:

- sqlite3* db: sqlite database structure
- char* path: path of directory where bus files are located

void populateBranchTables(sqlite3* db, char* path)

Prepares SQL statements to input data into tables. Calls traverseDirectory() for the branch files
Args:

- sqlite3* db: sqlite database structure
- char* path: path of directory where branch files are located

void populateTransformer2Tables(sqlite3* db, char* path)

Prepares SQL statements to input data into tables. Calls traverseDirectory() for the 2-winding transformer files
Args:

- sqlite3* db: sqlite database structure
- char* path: path of directory where 2-winding transformer files are located

void populateTransformer3Tables(sqlite3* db, char* path)

Prepares SQL statements to input data into tables. Calls traverseDirectory() for the 3-winding transformer files
Args:

- sqlite3* db: sqlite database structure
- char* path: path of directory where 3-winding transformer files are located

void populateGeneratorTables(sqlite3* db, char* path)

Prepares SQL statements to input data into tables. Calls traverseDirectory() for the generator files
Args:

- sqlite3* db: sqlite database structure
- char* path: path of directory where generator files are located

void populateOOSTables(sqlite3* db, char* path)

Prepares SQL statements to input data into tables. Calls traverseDirectory() for the OOS files
Args:

- sqlite3* db: sqlite database structure
- char* path: path of directory where OOS files are located

void repopulateTables(sqlite3* db)

Prepares and executes SQL statements to delete and recreate every table in the database. Essentially resets the database. Calls populate{Specific}Tables() functions for each type of file, to repopulate the tables.
Args:

- sqlite3* db: sqlite database structure

void updateTables(sqlite3* db)

Checks for any new or updated files in every folder.
Args:

- sqlite3* db: sqlite database structure

In depth look at some functions in

TABLES.C

processBusFile()

```c
void processBusFile(sqlite3* db, FILE* file, sqlite3_stmt* statements[], char* scenario, char* contingency, char* date) {
    int rc = 0;
    char line[1024];
    while (fgets(line, 1024, file) != NULL) {
        char* bus_number = strtok(line, ",");
        char* bus_name = strtok(NULL, ",");
        char* area = strtok(NULL, ",");
        char* zone = strtok(NULL, ",");
        char* owner = strtok(NULL, ",");
        char* voltage_base = strtok(NULL, ",");
        char* criteria_nlo = strtok(NULL, ",");
        char* criteria_nhi = strtok(NULL, ",");
        char* criteria_elo = strtok(NULL, ",");
        char* criteria_ehi = strtok(NULL, ",");
        char* stat = strtok(NULL, ",");
        char* bus_pu = strtok(NULL, ",");
        char* bus_angle = strtok(NULL, ",");
        char* violate = strtok(NULL, ",");
        char* exception = strtok(NULL, ",");
```

While there are still available lines to parse in the file, we will split at the specified delimiter (the comma) and save each value to an appropriately named variable.

```c
sqlite3_bind_int64(statements[0], 1, atoi(bus_number));
sqlite3_bind_text(statements[0], 2, bus_name, -1, SQLITE_STATIC);
sqlite3_bind_int(statements[0], 3, atoi(area));
sqlite3_bind_int(statements[0], 4, atoi(zone));
sqlite3_bind_int(statements[0], 5, atoi(owner));
sqlite3_bind_double(statements[0], 6, atof(voltage_base));
sqlite3_bind_double(statements[0], 7, atof(criteria_nlo));
sqlite3_bind_double(statements[0], 8, atof(criteria_nhi));
sqlite3_bind_double(statements[0], 9, atof(criteria_elo));
sqlite3_bind_double(statements[0], 10, atof(criteria_ehi));
stepStatement(db, statements[0]);

sqlite3_bind_text(statements[1], 1, scenario, -1, SQLITE_STATIC);
sqlite3_bind_text(statements[1], 2, contingency, -1, SQLITE_STATIC);
sqlite3_bind_int64(statements[1], 3, atoi(bus_number));
sqlite3_bind_int(statements[1], 4, atoi(stat));
sqlite3_bind_double(statements[1], 5, atof(bus_pu));
sqlite3_bind_double(statements[1], 6, atof(bus_angle));
sqlite3_bind_int(statements[1], 7, (strncmp(violate, "Fail", 4) == 0 || atoi(violate) == 1) ? 1 : 0);
sqlite3_bind_int(statements[1], 8, exception ? atoi(exception) : 0);
sqlite3_bind_text(statements[1], 9, date, -1, SQLITE_STATIC);
stepStatement(db, statements[1]);
```

Inputted as a parameter we have an array of sqlite statements. For the specific bus tables, statements[0] populates the “Bus” table, and statement[1] populates the “Bus Simulation Results” table. The initialization and preparation of these statements is done in populateBusTables(). Here, we bind values to the parametes of the statements, so we can reuse the statement for every line in the files. The statements are then executed with stepStatement(). More information about binding values can be found here.

```
char* sql_str = "INSERT OR IGNORE INTO BUS (`Bus Number`, `Bus Name`, `Area`,
`Zone`, `Owner`, `Voltage Base`, `criteria_nlo`, `criteria_nhi`,
`criteria_elo`, `criteria_ehi`) VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?);";
```

Example of statement[0]. Each question mark represents a parameter.

```
    for (int x = 0; x < 2; x++) {
        sqlite3_reset(statements[x]);
        sqlite3_clear_bindings(statements[x]);
    }
}
```

After stepping through the statements, we have to reset and clear the bindings for each statement, so it can be used for the next line. This process will continue for every line in the field, and this function will be called for every file in the directory.
NOTE: this exact logic is the same for processT3File, processGeneratorFile, and processOOSFile, so this version of the documentation will not go through the code for those.

populateBusTables()

```
void populateBusTables(sqlite3* db, char* path) {
    sqlite3_stmt* stmt = NULL;
    sqlite3_stmt* stmt2 = NULL;

    char* sql_str = "INSERT OR IGNORE INTO BUS (`Bus Number`, `Bus Name`, `Area`, `Zone`, `Owner`, `Voltage Base`, `criteria_nlo`
    prepareStatement(db, sql_str, &stmt);
    char* sql_str2 = "INSERT OR IGNORE INTO `Bus Simulation Results` (`Scenario Name`, `Contingency Name`, `Bus Number`, `stat`,
    prepareStatement(db, sql_str2, &stmt2);

    sqlite3_stmt* statements[] = { stmt, stmt2 };
    traverseDirectory(db, path, statements, 0);

    for (int i = 0; i < 2; i++) {
        sqlite3_finalize(statements[i]);
    }
}
```

sql_str populates the “Bus” table, and sql_str2 populates the “Bus Simulation Results” table. It is passed to the next step, traverseDirectory with type = 0. After finishing traversing and processing the files, the statements are finalized to avoid resource leaks.

populateBranchTables()

```
void populateBranchTables(sqlite3* db, char* path) {
    sqlite3_stmt* stmt = NULL;
    sqlite3_stmt* stmt2 = NULL;
    sqlite3_stmt* stmt3 = NULL;
    sqlite3_stmt* stmt4 = NULL;
    sqlite3_stmt* stmt5 = NULL;

    char* sql_str = "INSERT OR IGNORE INTO Branch (`Branch Name`, `Metered Bus Number`, `Other Bus Number`, `Branch ID`, `V
    prepareStatement(db, sql_str, &stmt);
    char* sql_str2 = "INSERT OR IGNORE INTO `Branch Simulation Results` (`Scenario Name`, `Contingency Name`, `Branch Name`
    prepareStatement(db, sql_str2, &stmt2);
    char* sql_str3 = "SELECT exists(SELECT * FROM `Branch` WHERE `Branch Name` = ?);";
    prepareStatement(db, sql_str3, &stmt3);
    char* sql_str4 = "UPDATE Branch SET `RateA sum` = ?, `RateB sum` = ?, `RateC sum` = ? WHERE `Branch Name` = ?;";
    prepareStatement(db, sql_str4, &stmt4);
    char* sql_str5 = "UPDATE Branch SET `RateA win` = ?, `RateB win` = ?, `RateC win` = ? WHERE `Branch Name` = ?;";
    prepareStatement(db, sql_str5, &stmt5);

    sqlite3_stmt* statements[] = { stmt, stmt2, stmt3, stmt4, stmt5 };
    traverseDirectory(db, path, statements, 1);

    for (int i = 0; i < 5; i++) {
        sqlite3_finalize(statements[i]);
    }
}
```

sql_str populates the “Branch” table, sql_str2 populates the “Branch Simulation Results” table. sql_str3 checks to see if the branch has been already added to the table. sql_str4 updates the row in “Branch” to add summer rates, sql_str5 updates the row in “Branch” to add the winter rates. The reasoning behind this is that if there are two files but one has the rates of summer and the other has rates for winter, then they can both be saved.

processBranchFile()

```
void processBranchFile(sqlite3* db, FILE* file, sqlite3_stmt* statements[], char* scenario, char* contingency, char* se
    int rc = 0;
    char line[1024];
    while (fgets(line, 1024, file) != NULL) {
        char* branch_name = strtok(line, ",");
        char* metered_bus_number = strtok(NULL, ",");
        char* other_bus_number = strtok(NULL, ",");
        char* branch_id = strtok(NULL, ",");
        char* voltage_base = strtok(NULL, ",");
        char* rateA = strtok(NULL, ",");
        char* rateB = strtok(NULL, ",");
        char* rateC = strtok(NULL, ",");
        char* p_metered = strtok(NULL, ",");
        char* p_other = strtok(NULL, ",");
        char* q_metered = strtok(NULL, ",");
        char* q_other = strtok(NULL, ",");
        char* amp_angle_metered = strtok(NULL, ",");
        char* amp_angle_other = strtok(NULL, ",");
        char* amp_metered = strtok(NULL, ",");
        char* amp_other = strtok(NULL, ",");
        char* ploss = strtok(NULL, ",");
        char* qloss = strtok(NULL, ",");
        char* stat = strtok(NULL, ",");
        char* violate = strtok(NULL, ",");
        char* exception = strtok(NULL, ",");

        sqlite3_bind_text(statements[2], 1, branch_name, -1, SQLITE_STATIC);
        stepStatement(db, statements[2]);
```

While there are still available lines to parse in the file, we will split at the specified delimiter (the comma) and save each value to an appropriately named variable. We then execute the statement to check if the branch has already been added to the “Branch” table.

```
if (exists == 1) { // branch already exists
    if (season[0] == 's' || season[0] == 'S') { // summer
        sqlite3_bind_double(statements[3], 1, atof(rateA));
        sqlite3_bind_double(statements[3], 2, atof(rateB));
        sqlite3_bind_double(statements[3], 3, atof(rateC));
        sqlite3_bind_text(statements[3], 4, branch_name, -1, SQLITE_STATIC);
        stepStatement(db, statements[3]);
    }
    else { // winter
        sqlite3_bind_double(statements[4], 1, atof(rateA));
        sqlite3_bind_double(statements[4], 2, atof(rateB));
        sqlite3_bind_double(statements[4], 3, atof(rateC));
        sqlite3_bind_text(statements[4], 4, branch_name, -1, SQLITE_STATIC);
        stepStatement(db, statements[4]);
    }
}
```

If the branch has already been added, then we check whether the season of the current file is summer or winter. Then we will update the table accordingly with the rates in the file.

```
else { // branch doesn't exist
    sqlite3_bind_text(statements[0], 1, branch_name, -1, SQLITE_STATIC);
    sqlite3_bind_int64(statements[0], 2, atoi(metered_bus_number));
    sqlite3_bind_int64(statements[0], 3, atoi(other_bus_number));
    sqlite3_bind_text(statements[0], 4, branch_id, -1, SQLITE_STATIC);
    sqlite3_bind_double(statements[0], 5, atof(voltage_base));

    if (season[0] == 's' || season[0] == 'S') {
        sqlite3_bind_double(statements[0], 6, atof(rateA));
        sqlite3_bind_double(statements[0], 7, atof(rateB));
        sqlite3_bind_double(statements[0], 8, atof(rateC));
    }
    else {
        sqlite3_bind_double(statements[0], 9, atof(rateA));
        sqlite3_bind_double(statements[0], 10, atof(rateB));
        sqlite3_bind_double(statements[0], 11, atof(rateC));
    }
    stepStatement(db, statements[0]);
}
```

If the branch does not exist, then we will add it as normal to the “Branch” table

```
        sqlite3_bind_text(statements[1], 1, scenario, -1, SQLITE_STATIC);
        sqlite3_bind_text(statements[1], 2, contingency, -1, SQLITE_STATIC);
        sqlite3_bind_text(statements[1], 3, branch_name, -1, SQLITE_STATIC);
        sqlite3_bind_int(statements[1], 4, atoi(stat));
        sqlite3_bind_double(statements[1], 5, atof(p_metered));
        sqlite3_bind_double(statements[1], 6, atof(p_other));
        sqlite3_bind_double(statements[1], 7, atof(q_metered));
        sqlite3_bind_double(statements[1], 8, atof(q_other));
        sqlite3_bind_double(statements[1], 9, atof(amp_angle_metered));
        sqlite3_bind_double(statements[1], 10, atof(amp_angle_other));
        sqlite3_bind_double(statements[1], 11, atof(amp_metered));
        sqlite3_bind_double(statements[1], 12, atof(amp_other));
        sqlite3_bind_double(statements[1], 13, atof(ploss));
        sqlite3_bind_double(statements[1], 14, atof(qloss));
        sqlite3_bind_int(statements[1], 15, (strncmp(violate, "Fail", 4) == 0 || atoi(violate) == 1) ? 1 : 0);
        sqlite3_bind_int(statements[1], 16, exception ? atoi(exception) : 0);
        sqlite3_bind_text(statements[1], 17, date, -1, SQLITE_STATIC);

        stepStatement(db, statements[1]);

        for (int i = 0; i < 5; i++) {
            sqlite3_reset(statements[i]);
            sqlite3_clear_bindings(statements[i]);
        }
    }
}
```

Finally, we add the data to the “Branch Simulation Results” table, and then reset the statements for the next row in the file.

populateTransformer2Tables()

```
void populateTransformer2Tables(sqlite3* db, char* path)   {
    sqlite3_stmt* stmt = NULL;
    sqlite3_stmt* stmt2 = NULL;
    char* sql_str = "INSERT OR IGNORE INTO Transformer2 (`Xformer Name`, `Winding 1`, `Winding 2`, `Xfmr ID`, `MVA Base`,
    prepareStatement(db, sql_str, &stmt);
    char* sql_str2 = "INSERT OR IGNORE INTO `Transformer2 Simulation Results` (`Scenario Name`, `Contingency Name`, `Xfor
    prepareStatement(db, sql_str2, &stmt2);
    sqlite3_stmt* statements[] = { stmt, stmt2 };
    traverseDirectory(db, path, statements, 2);
    for (int i = 0; i < 2; i++) {
        sqlite3_finalize(statements[i]);
    }
}
```

sql_str populates “Transformer2” table. sql_str2 populates “Transformer2 Simulation Results”.

processT2File()

```
void processT2File(sqlite3* db, FILE* file, sqlite3_stmt* statements[], char* scenario, char* contingency
    char line[1024];
    int rc = 0;
    while (fgets(line, 1024, file) != NULL) {
        size_t line_size = snprintf(NULL, 0, line) + 2;
        char* space = malloc(line_size);
        space[0] = '$';
        space[1] = '\0';
        strcat(space, line);
```

Some of the rows in the transformer files do not have a name for the transformer_name column. This will mess up the parsing of the data, so this is a temporary solution. We add an arbitrarily chosen ‘$’ at the beginning of each line to act as a placeholder for the transformer name, so that it can parse correctly.

```
char* xformer_name = strtok(space, ",");
char* winding1 = strtok(NULL, ",");
char* winding2 = strtok(NULL, ",");
char* xfmr_id = strtok(NULL, ",");
char* mva_base = strtok(NULL, ",");
char* winding1_nominal_kv = strtok(NULL, ",");
char* winding2_nominal_kv = strtok(NULL, ",");
char* rateA_winding1 = strtok(NULL, ",");
char* rateB_winding1 = strtok(NULL, ",");
char* rateC_winding1 = strtok(NULL, ",");
char* rateA_winding2 = strtok(NULL, ",");
char* rateB_winding2 = strtok(NULL, ",");
char* rateC_winding2 = strtok(NULL, ",");
char* p_winding1 = strtok(NULL, ",");
char* p_winding2 = strtok(NULL, ",");
char* q_winding1 = strtok(NULL, ",");
char* q_winding2 = strtok(NULL, ",");
char* amp_winding1 = strtok(NULL, ",");
char* amp_winding2 = strtok(NULL, ",");
char* ploss = strtok(NULL, ",");
char* qloss = strtok(NULL, ",");
char* stat = strtok(NULL, ",");
char* violate = strtok(NULL, ",");
char* exception = strtok(NULL, ",");
```

Here we are splitting at the specified delimiter (the comma) and saving each value to an appropriately named variable, as per usual.

```
if (strcmp(xformer_name, "$") == 0) {
    memset(xformer_name, 0, strlen(xformer_name));
    strcat(xformer_name, winding1);
    strcat(xformer_name, ":");
    strcat(xformer_name, winding2);
    strcat(xformer_name, ":");
    strcat(xformer_name, xfmr_id);
}
else {
    for (int i = 1; i < strlen(xformer_name); i++) {
        xformer_name[i - 1] = xformer_name[i];
    }
    xformer_name[strlen(xformer_name) - 1] = '\0';
}
```

We check if the transformer name is '$'. If so, then we know that there was no transformer name, so we create one of the format winding1:winding2:xfmr_id. If the transformer name is not '$', then it will be something of the form '$TRANSFORMER_NAME'. We remove the $ from the string.

```
    sqlite3_bind_text(statements[0], 1, xformer_name, -1, SQLITE_STATIC);
    sqlite3_bind_int64(statements[0], 2, atoi(winding1));
    sqlite3_bind_int64(statements[0], 3, atoi(winding2));
    sqlite3_bind_text(statements[0], 4, xfmr_id, -1, SQLITE_STATIC);
    sqlite3_bind_double(statements[0], 5, atof(mva_base));
    sqlite3_bind_double(statements[0], 6, atof(winding1_nominal_kv));
    sqlite3_bind_double(statements[0], 7, atof(winding2_nominal_kv));
    sqlite3_bind_double(statements[0], 8, atof(rateA_winding1));
    sqlite3_bind_double(statements[0], 9, atof(rateB_winding1));
    sqlite3_bind_double(statements[0], 10, atof(rateC_winding1));
    sqlite3_bind_double(statements[0], 11, atof(rateA_winding2));
    sqlite3_bind_double(statements[0], 12, atof(rateB_winding2));
    sqlite3_bind_double(statements[0], 13, atof(rateC_winding2));
    stepStatement(db, statements[0]);


    sqlite3_bind_text(statements[1], 1, scenario, -1, SQLITE_STATIC);
    sqlite3_bind_text(statements[1], 2, contingency, -1, SQLITE_STATIC);
    sqlite3_bind_text(statements[1], 3, xformer_name, -1, SQLITE_STATIC);
    sqlite3_bind_int(statements[1], 4, atoi(stat));
    sqlite3_bind_double(statements[1], 5, atof(p_winding1));
    sqlite3_bind_double(statements[1], 6, atof(p_winding2));
    sqlite3_bind_double(statements[1], 7, atof(q_winding1));
    sqlite3_bind_double(statements[1], 8, atof(q_winding2));
    sqlite3_bind_double(statements[1], 9, atof(amp_winding1));
    sqlite3_bind_double(statements[1], 10, atof(amp_winding2));
    sqlite3_bind_double(statements[1], 11, atof(ploss));
    sqlite3_bind_double(statements[1], 12, atof(qloss));
    sqlite3_bind_int(statements[1], 13, strncmp(violate, "Fail", 4) == 0 || atoi(violate) == 1 ? 1 : 0);
    sqlite3_bind_int(statements[1], 14, exception ? atoi(exception) : 0);
    sqlite3_bind_text(statements[1], 15, date, -1, SQLITE_STATIC);
    stepStatement(db, statements[1]);

    for (int y = 0; y < 2; y++) {
        sqlite3_reset(statements[y]);
        sqlite3_clear_bindings(statements[y]);
    }
    free(space);
}
```

Here, we bind values to the parameters and execute the statements.

populateTransformer3Tables()

```
void populateTransformer3Tables(sqlite3* db, char* path) {
    sqlite3_stmt* stmt = NULL;
    sqlite3_stmt* stmt2 = NULL;
    char* sql_str = "INSERT OR IGNORE INTO Transformer3 (`amp_winding 2`, `Xformer Name`, `Winding 1`, `Winding 2`, `Winding 3`, `Xfmr ID`, `Winding 1 MVA Base`, 
    prepareStatement(db, sql_str, &stmt);
    char* sql_str2 = "INSERT OR IGNORE INTO `Transformer3 Simulation Results` (`Scenario Name`, `Contingency Name`, `Xformer Name`, `W1 Status`, `W2 Status`, `W3 S
    prepareStatement(db, sql_str2, &stmt2);
    sqlite3_stmt* statements[] = { stmt, stmt2 };
    traverseDirectory(db, path, statements, 3);
    for (int i = 0; i < 2; i++) {
        sqlite3_finalize(statements[i]);
    }
}
```

sql_str populates the “Transformer3” table, and sql_str2 populates the “Transformer3 Simulation Results” table. It is passed to the next step, traverseDirectory with type = 3. After finishing traversing and processing the files, the statements are finalized to avoid resource leaks.

processT3File()

processT3File follows the exact same logic as processBusFile() on page 19.

populateGeneratorTables()

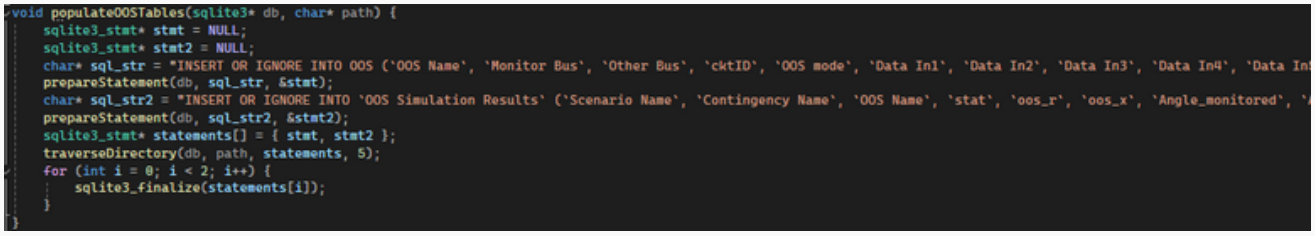
```
void populateGeneratorTables(sqlite3* db, char* path) {
    sqlite3_stmt* stmt = NULL;
    sqlite3_stmt* stmt2 = NULL;
    char* sql_str = "INSERT OR IGNORE INTO Generator (`Bus Number`, `Gen ID`, `Bus Name`, `Owner`, `Voltage Base`, `criteria_nlo`, `criteria_nhi`, `criteria_elo`, `criteria_ehi`, `Pmi
    prepareStatement(db, sql_str, &stmt);
    char* sql_str2 = "INSERT OR IGNORE INTO `Generator Simulation Results` (`Scenario Name`, `Contingency Name`, `Bus Number`, `Gen ID`, `stat`, `Scheduled Voltage`, `Terminal Voltage
    prepareStatement(db, sql_str2, &stmt2);
    sqlite3_stmt* statements[] = { stmt, stmt2 };
    traverseDirectory(db, path, statements, 4);
    for (int i = 0; i < 2; i++) {
        sqlite3_finalize(statements[i]);
    }
}
```

sql_str populates the “Generator” table, and sql_str2 populates the “Generator Simulation Results” table. It is passed to the next step, traverseDirectory with type = 4. After finishing traversing and processing the files, the statements are finalized to avoid resource leaks.

processGeneratorFile()

processGeneratorFile() follows the exact same logic as processBusFile() on page 19.

populateOOSTables()



sql_str populates the “OOS” table, and sql_str2 populates the “OOS Simulation Results” table. It is passed to the next step, traverseDirectory with type = 5. After finishing traversing and processing the files, the statements are finalized to avoid resource leaks.

processOOSFile()

processOOSFile() follows the exact same logic as processBusFile() on page 19.

traverseDirectory()

Using the external C library dirent, we are able to perform directory operations. We save the full file path to filePath, and then check if are in a file or a directory.

```c
        if (S_ISDIR(statbuf.st_mode)) { // recursively traverse through directories
            if (strcmp(dir->d_name, ".") == 0 || strcmp(dir->d_name, "..") == 0)
                continue;
            traverseDirectory(db, filePath, statements, type);
        }
        if (strstr(dir->d_name, ".csv") != NULL) { // if it's a csv file then we process
            char* filename = strdup(dir->d_name);
            if (!filename) {
                handle_error(db, "strdup error");
                closedir(d);
                exit(-1);
            }
            processFile(db, path, filename, statements, type);
        }
    }
    closedir(d);
}
```

If we are in another directory, then we recursively call traverseDirectory again. If it is a csv file, then we begin to process it by calling processFile. Anything else is ignored.

processFile()

```c
void processFile(sqlite3* db, char*path, char* filename, sqlite3_stmt* statements[], int type) {
    printf("processing file: %s\n", filename);
    char* contingency = getContingency(filename);
    char* scenario = getScenario(filename);
    char* season = getSeason(filename);
    char filePath[1024];
    snprintf(filePath, sizeof(filePath), "%s\\%s", path, filename);
    char date[100];
    getDate(date, sizeof(date), filePath);
    FILE* file = fopen(filePath, "r");
    if (file == NULL) {
        printf("Couldn't open file\n");
        sqlite3_close(db);
        exit(-1);
    }
    char line[1024];
    fgets(line, 1024, file); // gets rid of column name
    printf("LINE: %s\n", line);
```

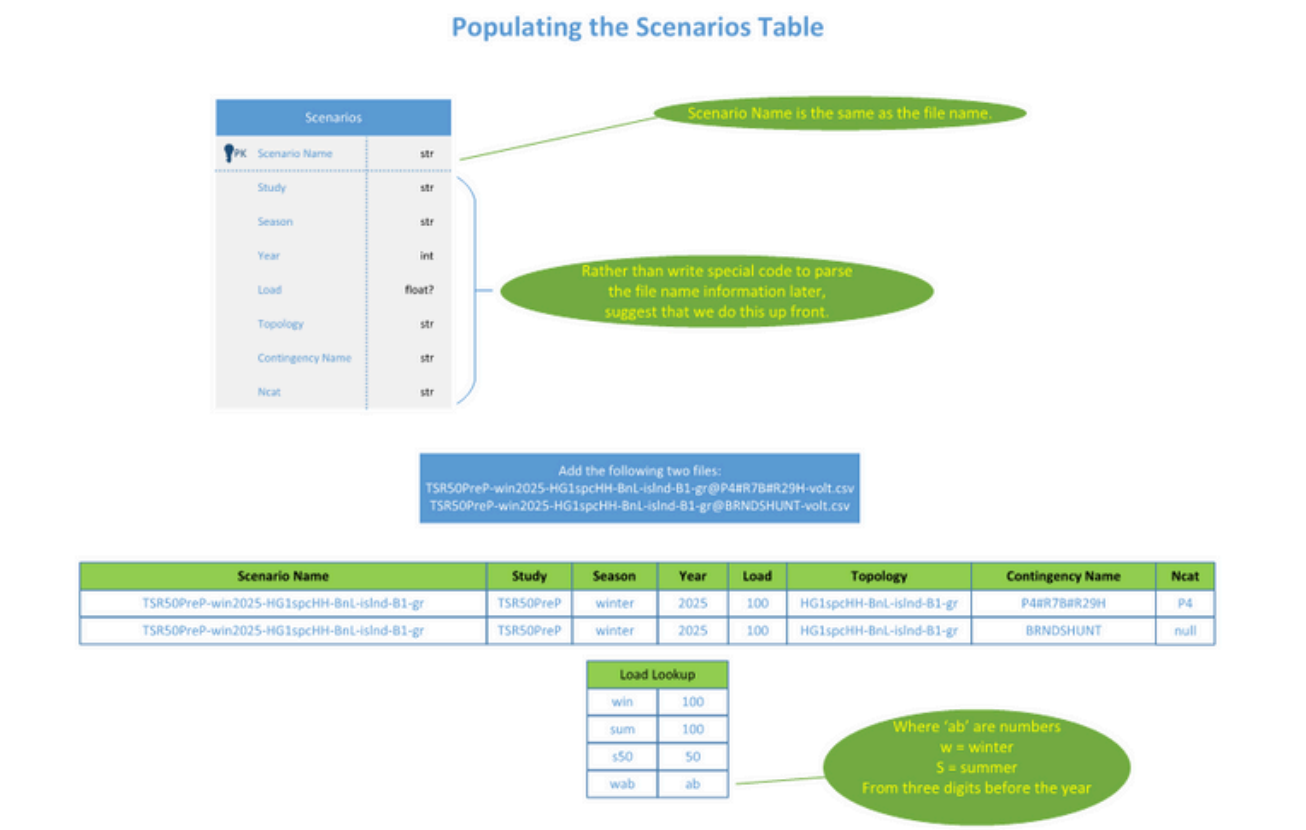
We start getting the contingency, scenario, season, and last modified date. Then we open the file and fgets the first line to get rid of the column names.

```c
if (type == 0) { // 0 is for bus table
    printf("bus table\n");
    processBusFile(db, file, statements, scenario, contingency, date);
}
else if (type == 1) { // branch table
    printf("branch table\n");
    processBranchFile(db, file, statements, scenario, contingency, season, date);
}
else if (type == 2) { // transformer2 table
    printf("t2 table\n");
    processT2File(db, file, statements, scenario, contingency, date);
}
else if (type == 3) { // transformer3 table
    printf("t3 table\n");
    processT3File(db, file, statements, scenario, contingency, date);
}
else if (type == 4) { // populate generator table
    printf("generator table\n");
    processGeneratorFile(db, file, statements, scenario, contingency, date);
}
else if (type == 5) { // populate oos table
    printf("oos table\n");
    processOOSFile(db, file, statements, scenario, contingency, date);
}
else if (type > 9) { // checks for updated files
    printf("check for updated files\n");
    checkForUpdates(db, file, statements, scenario, contingency, season, date, type % 10);
```

Depending on the type of file, we call the corresponding process function. See type table here on page 16 for the different type codes.

populateScenCont()

IMPORTANT: This function assumes filenames of the format
study-season-year-topology@contingency.csv

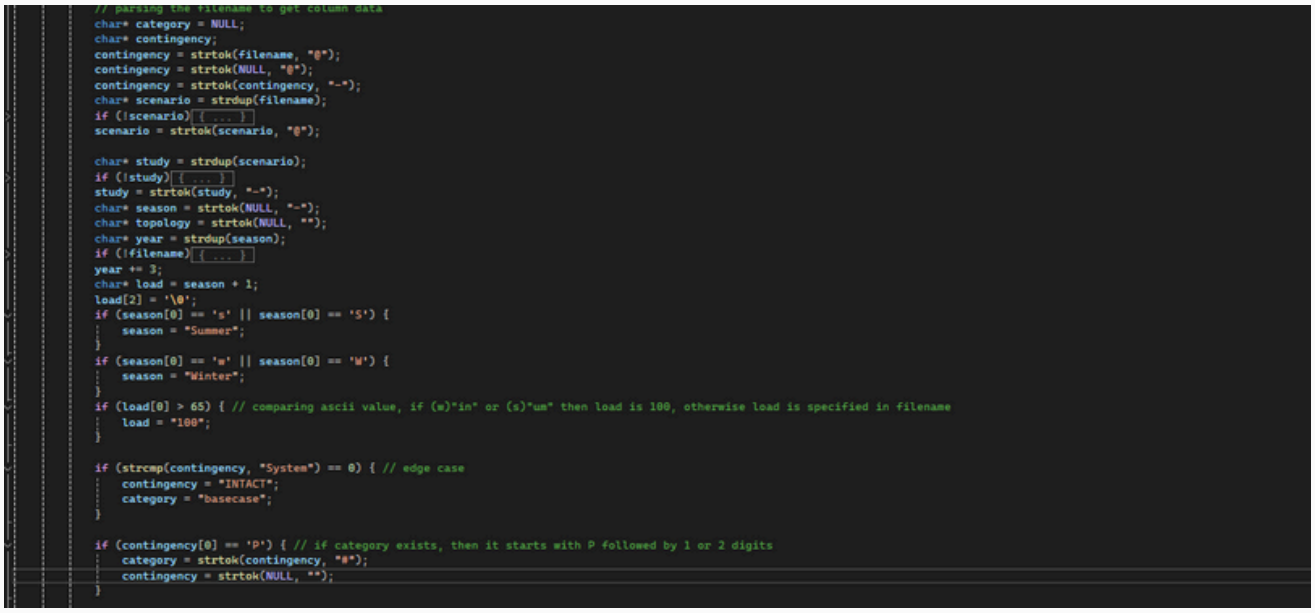



```c
void populateScenCont(sqlite3* db, const char* path) {
    sqlite3_stmt* stmt = NULL;
    int rc = 0;
    DIR* d;
    struct dirent* dir;
    d = opendir(path);
    if (d == NULL) {
        printf("Couldn't open directory\n");
        sqlite3_close(db);
        exit(-1);
    }

    while ((dir = readdir(d)) != NULL) {
        char filePath[1024];
        snprintf(filePath, sizeof(filePath), "%s/%s", path, dir->d_name);
        struct stat statbuf;
        if (stat(filePath, &statbuf) == -1) {
            printf("Couldn't get file stats\n");
            sqlite3_close(db);
            exit(-1);
        }

        if (S_ISDIR(statbuf.st_mode)) { // recursively traverse through directories
            if (strcmp(dir->d_name, ".") == 0 || strcmp(dir->d_name, "..") == 0)
                continue;
            populateScenCont(db, filePath);
        }
        if (strstr(dir->d_name, ".csv") != NULL) { // if it's a csv file then we process
            char* filename = strdup(dir->d_name);
            if (!filename) {
                handle_error(db, "strdup error");
                closedir(d);
                exit(-1);
            }
```

Begins by opening the directory and checking for csv files (same as traverseDirectory())



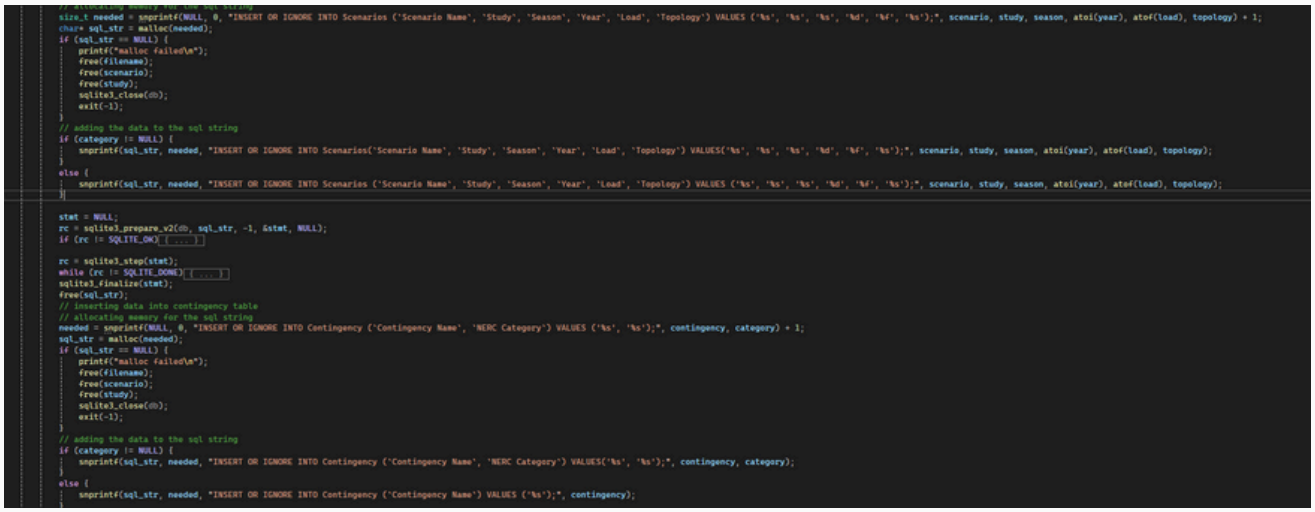
```
// parsing the filename to get column data
char* category = NULL;
char* contingency;
contingency = strtok(filename, "@");
contingency = strtok(NULL, "@");
contingency = strtok(contingency, "-");
char* scenario = strdup(filename);
if (!scenario) { ... }
scenario = strtok(scenario, "@");

char* study = strdup(scenario);
if (!study) { ... }
study = strtok(study, "-");
char* season = strtok(NULL, "-");
char* topology = strtok(NULL, "");
char* year = strdup(season);
if (!filename) { ... }
year += 3;
char* load = season + 1;
load[2] = '\0';
if (season[0] == 's' || season[0] == 'S') {
    season = "Summer";
}
if (season[0] == 'w' || season[0] == 'W') {
    season = "Winter";
}
if (load[0] > 65) { // comparing ascii value, if (w)"in" or (s)"um" then load is 100, otherwise load is specified in filename
    load = "100";
}

if (strcmp(contingency, "System") == 0) { // edge case
    contingency = "INTACT";
    category = "basecase";
}

if (contingency[0] == 'P') { // if category exists, then it starts with P followed by 1 or 2 digits
    category = strtok(contingency, "@");
    contingency = strtok(NULL, "");
}
```

Then there is basic logic to extract information from the file title.



```
// allocating memory for the sql string
size_t needed = snprintf(NULL, 0, "INSERT OR IGNORE INTO Scenarios (`Scenario Name`, `Study`, `Season`, `Year`, `Load`, `Topology`) VALUES (`%s`, `%s`, `%s`, `%d`, `%f`, `%s`);", scenario, study, season, atoi(year), atof(load), topology) + 1;
char* sql_str = malloc(needed);
if (sql_str == NULL) {
    printf("malloc failed\n");
    free(filename);
    free(scenario);
    free(study);
    sqlite3_close(db);
    exit(-1);
}
// adding the data to the sql string
if (category != NULL) {
    snprintf(sql_str, needed, "INSERT OR IGNORE INTO Scenarios(`Scenario Name`, `Study`, `Season`, `Year`, `Load`, `Topology`) VALUES(`%s`, `%s`, `%s`, `%d`, `%f`, `%s`);", scenario, study, season, atoi(year), atof(load), topology);
}
else {
    snprintf(sql_str, needed, "INSERT OR IGNORE INTO Scenarios (`Scenario Name`, `Study`, `Season`, `Year`, `Load`, `Topology`) VALUES (`%s`, `%s`, `%s`, `%d`, `%f`, `%s`);", scenario, study, season, atoi(year), atof(load), topology);
}

stmt = NULL;
rc = sqlite3_prepare_v2(db, sql_str, -1, &stmt, NULL);
if (rc != SQLITE_OK) { ... }

rc = sqlite3_step(stmt);
while (rc != SQLITE_DONE) { ... }
sqlite3_finalize(stmt);
free(sql_str);
// inserting data into contingency table
// allocating memory for the sql string
needed = snprintf(NULL, 0, "INSERT OR IGNORE INTO Contingency (`Contingency Name`, `NERC Category`) VALUES (`%s`, `%s`);", contingency, category) + 1;
sql_str = malloc(needed);
if (sql_str == NULL) {
    printf("malloc failed\n");
    free(filename);
    free(scenario);
    free(study);
    sqlite3_close(db);
    exit(-1);
}
// adding the data to the sql string
if (category != NULL) {
    snprintf(sql_str, needed, "INSERT OR IGNORE INTO Contingency (`Contingency Name`, `NERC Category`) VALUES(`%s`, `%s`);", contingency, category);
}
else {
    snprintf(sql_str, needed, "INSERT OR IGNORE INTO Contingency (`Contingency Name`) VALUES (`%s`);", contingency);
}
```

Then we insert parameters into the sql_str, and execute the statements. There are ones to insert into the Scenarios and Contingency tables.

repopulateTables()

```
void repopulateTables(sqlite3* db) {
    const char* queries[] = {
        "DROP TABLE IF EXISTS Scenarios;",
        "CREATE TABLE Scenarios (`Scenario Name` TEXT PRIMARY KEY, `Study` TEXT, Season TEXT, Year INT, Load FLOAT, Topology TEXT);",
        "DROP TABLE IF EXISTS Contingency;",
        "CREATE TABLE Contingency (`Contingency Name` TEXT PRIMARY KEY, `NERC Category` TEXT, `Date Last Modified` TEXT);",
        "DROP TABLE IF EXISTS BUS;",
        "CREATE TABLE BUS (`Bus Number` INT PRIMARY KEY, `Bus Name` TEXT, Area INT, Zone INT, Owner INT, `Voltage Base` FLOAT, criteria_nlo FLOAT, criteria_
        "DROP TABLE IF EXISTS `Bus Simulation Results`;",
        "CREATE TABLE `Bus Simulation Results` (`Scenario Name` TEXT, `Contingency Name` TEXT, `Bus Number` INT, stat INT, bus_pu FLOAT NOT NULL, bus_angle
        "DROP TABLE IF EXISTS Branch;",
        "CREATE TABLE Branch (`Branch Name` TEXT PRIMARY KEY, `Metered Bus Number` INT, `Other Bus Number` INT, `Branch ID` TEXT, `Voltage Base` FLOAT, `Rat
        "DROP TABLE IF EXISTS `Branch Simulation Results`;",
        "CREATE TABLE `Branch Simulation Results` (`Scenario Name` TEXT, `Contingency Name` TEXT, `Branch Name` TEXT, `stat` INT, `p_metered` FLOAT, `p_othe
        "DROP TABLE IF EXISTS `Transformer2`;",
        "CREATE TABLE Transformer2 (`Xformer Name` TEXT PRIMARY KEY, `Winding 1` INT, `Winding 2` INT, `Xfmr ID` TEXT, `MVA Base` FLOAT, `Winding 1 nominal
        "DROP TABLE IF EXISTS `Transformer2 Simulation Results`;",
        "CREATE TABLE `Transformer2 Simulation Results` (`Scenario Name` TEXT, `Contingency Name` TEXT, `Xformer Name` TEXT, `stat` INT, `p_winding 1` FLOAT
    };
    sqlite3_stmt* stmt = NULL;
    for (int i = 0; i < 16; i++) {
        stmt = NULL;
        prepareStatement(db, queries[i], &stmt);
        stepStatement(db, stmt);
        sqlite3_finalize(stmt);
    }
```

This function prepares and executes a series of sql statements that delete and remake all the tables. Essentially, it performs a “hard reset” of the database.

```
populateScenCont(db, ".");
populateBusTables(db, VOLTAGE_FOLDER);
populateBranchTables(db, THERMALBRANCH_FOLDER);
populateTransformer2Tables(db, THERMAL2_FOLDER);
```

Then it calls the different populate functions.

updateTables()

```
sqlite3_stmt* stmt = NULL;
char* sql_str = "SELECT `Date Last Modified` FROM `Bus Simulation Results` WHERE `Scenario Name` = ? and `Contingency Name` = ?;";
prepareStatement(db, sql_str, &stmt);
sqlite3_stmt* statements[] = { stmt };
traverseDirectory(db, VOLTAGE_FOLDER, statements, 10);
sqlite3_finalize(stmt);
stmt = NULL;
printf("DONE CHECKING BUS\n");
```

prepares a statement that will assist in checking the last modified date, to see if a particular file has been updated. The logic is that if the last modified date in the database is different than the last modified date in the directory, that means that the file has been modified, so then we will update the database. This prevents us from reparsing every single file, because that is very slow. Repeats preparing these statements for every type of file.

checkForUpdates()

```
sqlite3_bind_text(statements[0], 1, scenario, -1, SQLITE_STATIC);
sqlite3_bind_text(statements[0], 2, contingency, -1, SQLITE_STATIC);
sqlite3_bind_text(statements[0], 3, date, -1, SQLITE_STATIC);
printf("current file date: %s\n", date);
stepStatement(db, statements[0]);

const unsigned char* extracted_date = sqlite3_column_text(statements[0], 0);
printf("date: %s\n", extracted_date);
if (extracted_date != NULL && strcmp(extracted_date, date) == 0) {
    printf("No updates\n");
}
```

Extracts the last modified date of the associated file that is stored in the database, if exists. If there is a last extracted date, and that date is the same as the current date associated with the file, then that means there are no updates, so nothing happens and we move onto the next file.

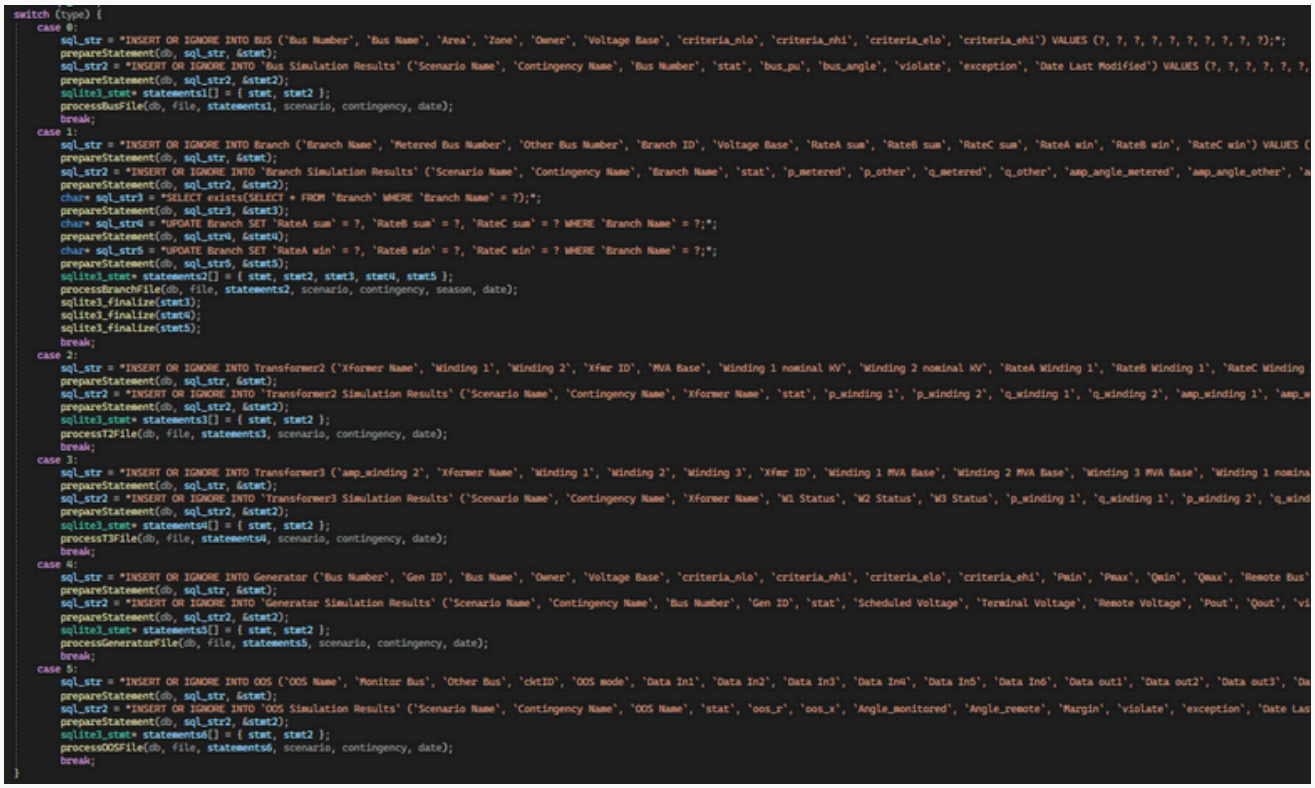
```
else {
    printf("updates found or new file found\n");
    sqlite3_stmt* stmt = NULL;
    sqlite3_stmt* stmt2 = NULL;
    sqlite3_stmt* stmt3 = NULL;
    sqlite3_stmt* stmt4 = NULL;
    sqlite3_stmt* stmt5 = NULL;

    char* table_name;
    switch (type) {
        case 0:
            table_name = "Bus";
            break;
        case 1:
            table_name = "Branch";
            break;
        case 2:
            table_name = "Transformer2";
            break;
        case 3:
            table_name = "Transformer3";
            break;
        case 4:
            table_name = "Generator";
            break;
        case 5:
            table_name = "OOS";
            break;
    }
```

Otherwise, that means the file is either new so the database search does not return anything, or the file has been recently updated so the date is different. In the switch statement we can determine what type it is. The type is originally inputted as 10, 11, 12.. in order to differentiate updating the database from completely repopulating the data. However, modulo 10 is then performed onto the type, so it becomes 0, 1, 2..

```
size_t needed = snprintf(NULL, 0, "DELETE FROM `%s Simulation Results` WHERE `Scenario Name` = '%s' and `Contingency Name` ='%s';", table_name, scenario, contingency) + 1;
char* sql_str = malloc(needed);
snprintf(sql_str, needed, "DELETE FROM `%s Simulation Results` WHERE `Scenario Name` = '%s' and `Contingency Name` ='%s';", table_name, scenario, contingency);
prepareStatement(db, sql_str, &stmt);
stepStatement(db, stmt);
sqlite3_finalize(stmt);
stmt = NULL;
```

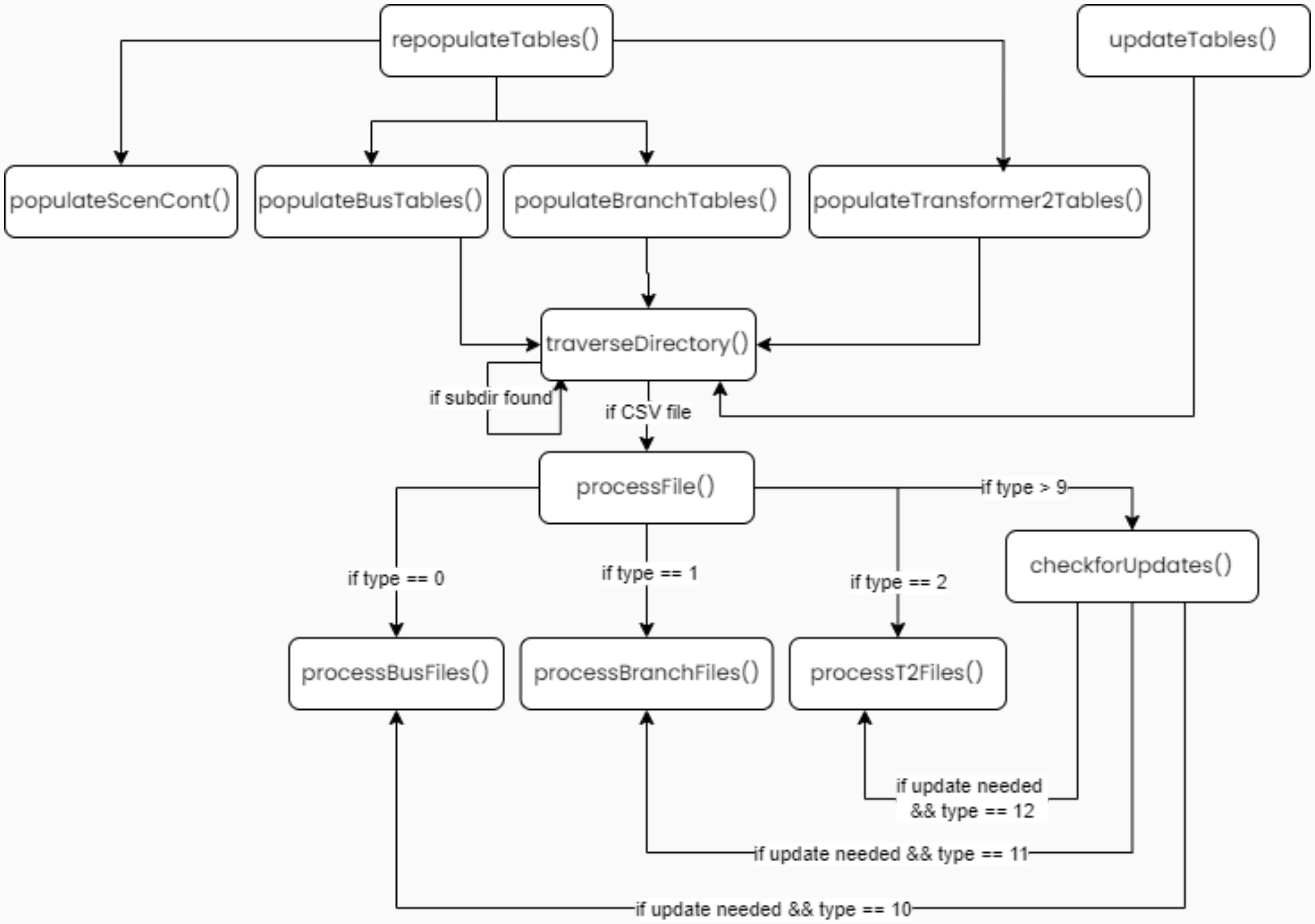
Then, all entries for the scenario and contingency are deleted from the corresponding table in the database.



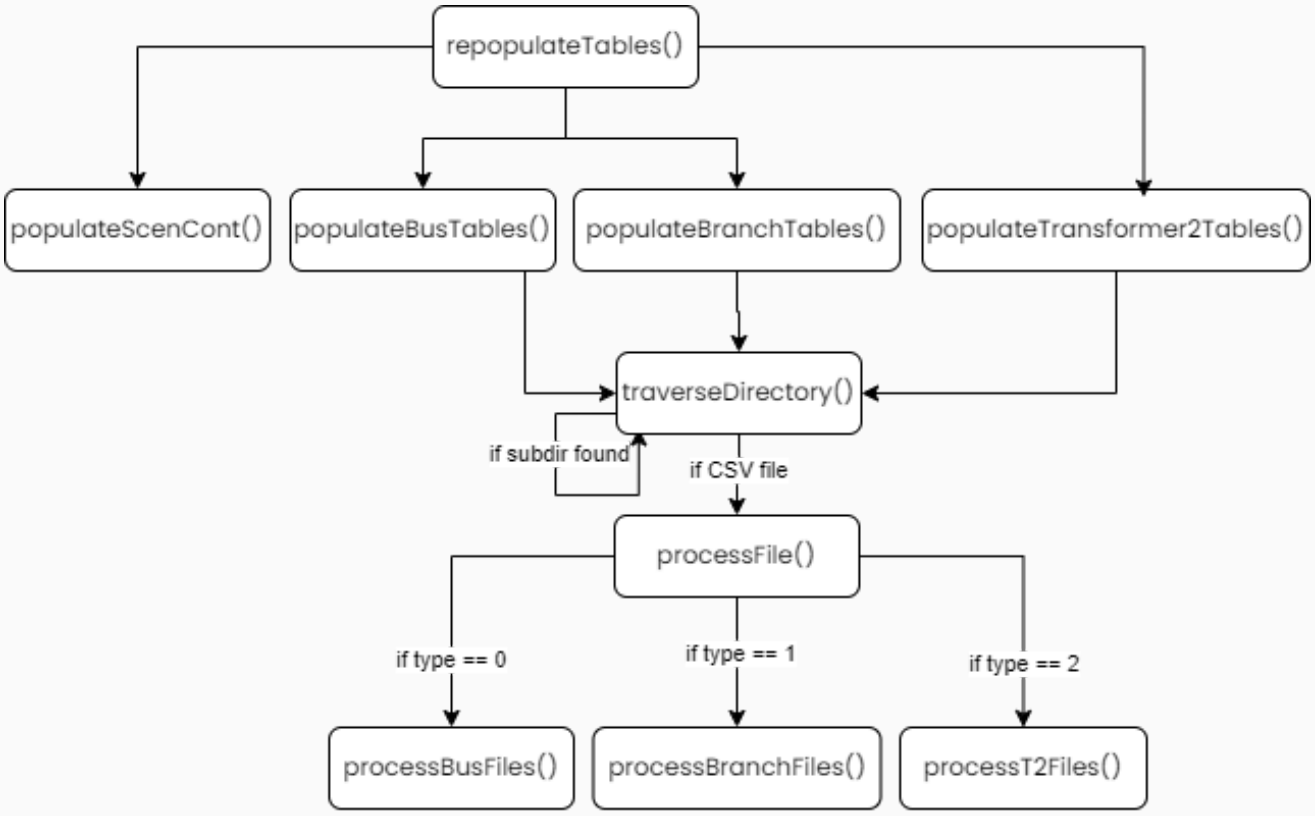
```
switch (type) {
    case 0:
        sql_str = "INSERT OR IGNORE INTO BUS (`Bus Number`, `Bus Name`, `Area`, `Zone`, `Owner`, `Voltage Base`, `criteria_nlo`, `criteria_nhi`, `criteria_elo`, `criteria_ehi`) VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?);";
        prepareStatement(db, sql_str, &stmt);
        sql_str2 = "INSERT OR IGNORE INTO `Bus Simulation Results` (`Scenario Name`, `Contingency Name`, `Bus Number`, `stat`, `bus_pu`, `bus_angle`, `violate`, `exception`, `Date Last Modified`) VALUES (?, ?, ?, ?, ?, ?,
        prepareStatement(db, sql_str2, &stmt2);
        sqlite3_stmt* statements1[] = { stmt, stmt2 };
        processBusFile(db, file, statements1, scenario, contingency, date);
        break;
    case 1:
        sql_str = "INSERT OR IGNORE INTO Branch (`Branch Name`, `Metered Bus Number`, `Other Bus Number`, `Branch ID`, `Voltage Base`, `RateA sum`, `RateB sum`, `RateC sum`, `RateA win`, `RateB win`, `RateC win`) VALUES (
        prepareStatement(db, sql_str, &stmt);
        sql_str2 = "INSERT OR IGNORE INTO `Branch Simulation Results` (`Scenario Name`, `Contingency Name`, `Branch Name`, `stat`, `p_metered`, `p_other`, `q_metered`, `q_other`, `amp_angle_metered`, `amp_angle_other`, `a
        prepareStatement(db, sql_str2, &stmt2);
        char* sql_str3 = "SELECT exists(SELECT * FROM `Branch` WHERE `Branch Name` = ?);";
        prepareStatement(db, sql_str3, &stmt3);
        char* sql_str4 = "UPDATE Branch SET `RateA sum` = ?, `RateB sum` = ?, `RateC sum` = ? WHERE `Branch Name` = ?;";
        prepareStatement(db, sql_str4, &stmt4);
        char* sql_str5 = "UPDATE Branch SET `RateA win` = ?, `RateB win` = ?, `RateC win` = ? WHERE `Branch Name` = ?;";
        prepareStatement(db, sql_str5, &stmt5);
        sqlite3_stmt* statements2[] = { stmt, stmt2, stmt3, stmt4, stmt5 };
        processBranchFile(db, file, statements2, scenario, contingency, season, date);
        sqlite3_finalize(stmt3);
        sqlite3_finalize(stmt4);
        sqlite3_finalize(stmt5);
        break;
    case 2:
        sql_str = "INSERT OR IGNORE INTO Transformer2 (`Xformer Name`, `Winding 1`, `Winding 2`, `Xfmr ID`, `MVA Base`, `Winding 1 nominal kV`, `Winding 2 nominal kV`, `RateA Winding 1`, `RateB Winding 1`, `RateC Winding
        prepareStatement(db, sql_str, &stmt);
        sql_str2 = "INSERT OR IGNORE INTO `Transformer2 Simulation Results` (`Scenario Name`, `Contingency Name`, `Xformer Name`, `stat`, `p_winding 1`, `p_winding 2`, `q_winding 1`, `q_winding 2`, `amp_winding 1`, `amp_w
        prepareStatement(db, sql_str2, &stmt2);
        sqlite3_stmt* statements3[] = { stmt, stmt2 };
        processT2File(db, file, statements3, scenario, contingency, date);
        break;
    case 3:
        sql_str = "INSERT OR IGNORE INTO Transformer3 (`amp_winding 2`, `Xformer Name`, `Winding 1`, `Winding 2`, `Winding 3`, `Xfmr ID`, `Winding 1 MVA Base`, `Winding 2 MVA Base`, `Winding 3 MVA Base`, `Winding 1 nomina
        prepareStatement(db, sql_str, &stmt);
        sql_str2 = "INSERT OR IGNORE INTO `Transformer3 Simulation Results` (`Scenario Name`, `Contingency Name`, `Xformer Name`, `W1 Status`, `W2 Status`, `W3 Status`, `p_winding 1`, `q_winding 1`, `p_winding 2`, `q_wind
        prepareStatement(db, sql_str2, &stmt2);
        sqlite3_stmt* statements4[] = { stmt, stmt2 };
        processT3File(db, file, statements4, scenario, contingency, date);
        break;
    case 4:
        sql_str = "INSERT OR IGNORE INTO Generator (`Bus Number`, `Gen ID`, `Bus Name`, `Owner`, `Voltage Base`, `criteria_nlo`, `criteria_nhi`, `criteria_elo`, `criteria_ehi`, `Pmin`, `Pmax`, `Qmin`, `Qmax`, `Remote Bus`
        prepareStatement(db, sql_str, &stmt);
        sql_str2 = "INSERT OR IGNORE INTO `Generator Simulation Results` (`Scenario Name`, `Contingency Name`, `Bus Number`, `Gen ID`, `stat`, `Scheduled Voltage`, `Terminal Voltage`, `Remote Voltage`, `Pout`, `Qout`, `vi
        prepareStatement(db, sql_str2, &stmt2);
        sqlite3_stmt* statements5[] = { stmt, stmt2 };
        processGeneratorFile(db, file, statements5, scenario, contingency, date);
        break;
    case 5:
        sql_str = "INSERT OR IGNORE INTO OOS (`OOS Name`, `Monitor Bus`, `Other Bus`, `cktID`, `OOS mode`, `Data In1`, `Data In2`, `Data In3`, `Data In4`, `Data In5`, `Data In6`, `Data out1`, `Data out2`, `Data out3`, `Da
        prepareStatement(db, sql_str, &stmt);
        sql_str2 = "INSERT OR IGNORE INTO `OOS Simulation Results` (`Scenario Name`, `Contingency Name`, `OOS Name`, `stat`, `oos_r`, `oos_x`, `Angle_monitored`, `Angle_remote`, `Margin`, `violate`, `exception`, `Date Las
        prepareStatement(db, sql_str2, &stmt2);
        sqlite3_stmt* statements6[] = { stmt, stmt2 };
        processOOSFile(db, file, statements6, scenario, contingency, date);
        break;
}
```

The SQL statements for the corresponding table are prepared, and then the file is processed again, readding the data to the table.

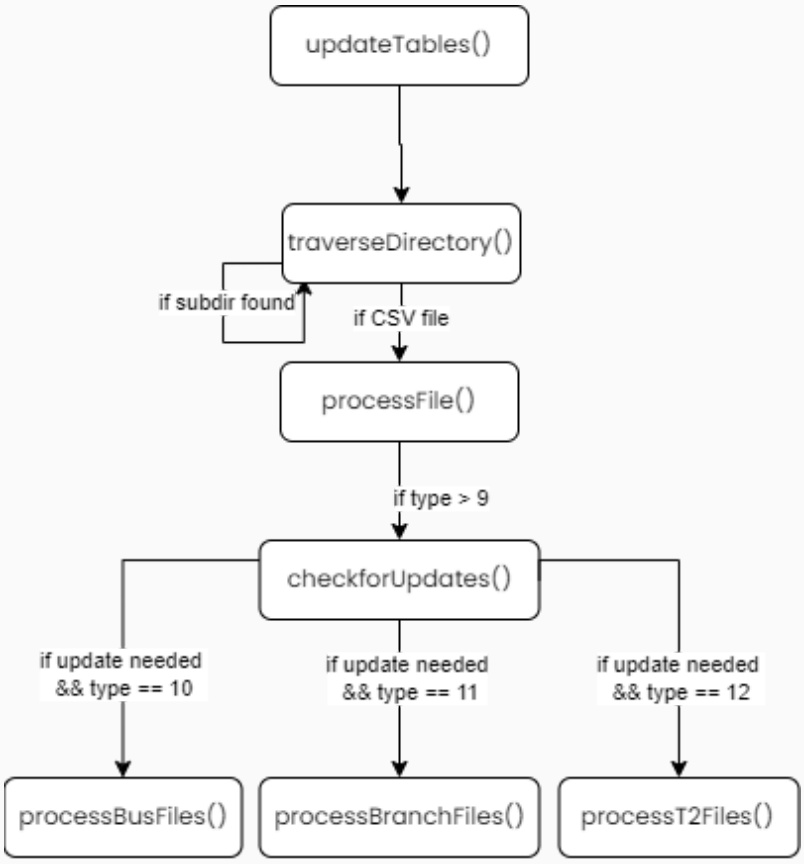
Business Logic of
TABLES . C



Logic for entire code



Logic for repopulating tables in database



Logic for updating tables

5. FRONTEND

The code for the graphical user interface can be found in main.py. This code is written in the Python programming language and primarly uses the PyQt6 and pylatex libraries.

Overview of MAIN.PY

INSTANCE VARIABLES

self.contingency: Current selected contingency
self.scenario: Current selected scenario
self.season: Current selected scenario
self.num_thermalbranch: Total number of thermal branch violations
self.num_voltage: Total number of voltage violations
self.num_trans2: Total number of 2-winding transformer violations
self.bus_data: An array of the voltage violations and the associated data for the selected contingency and scenario.
self.branch_data: An array of the thermal branch violations and the associated data for the selected contingency and scenario.
self.trans2_data: An array of the 2-winding transformer violations and the associated data for the selected contingency and scenario.
self.column_names: A dictionary that maps the table names with a list of column names
self.bus_table_index: A dictionary that maps the column name with the index of the data in self.bus_data. The data is in no particular order, just due to how the data was retrieved.
self.branch_table_index: A dictionary that maps the column name with the index of the data in self.branch_data
self.trans2_table_index: A dictionary that maps the column name with the index of the data in self.trans2_data

__init__

Initializes instance variables, lays out GUI and creates all elements

add_data_to_combobox(combobox, column, table)

Adds contingencies and scenarios to their corresponding comboboxes
Args:

- combobox: combox that we want to fill
- column: name of column in database table that we want to get the data from
- table: name of table that we want to get the data from

retrieve_data()

Once a scenario and a contingency (optional) is selected, then retrieves all the violations with the same scenario (and contingency).

generate_report()

Called directly from retrieve_data(). Puts the retrieved data into a LaTeX file using the PyLaTeX library.

create_table(table_name, data, index)

Helper function for generate_report() that creates the tables containing the violations and other data
Args:

- table_name: name of table that you want (function will append "Violations")
- data: data extracted in retrieve_data (stored in instance variables)
- index: dictionary instance variable that maps the data with the index in the list

display_report(tex_file)

Displays the generated tex document as HTML (it is faster to display HTML than to convert to a PDF and display that).
Args:

- tex_file: name of tex file

save_report()

Will export .tex document to a PDF and save to your computer

done_alert()

Alerts user when the PDF document is finished exporting

In depth look at some functions in

MAIN.PY

add_data_to_combobox()

```
def add_data_to_combobox(self, combobox, column, table):
    conn = sqlite3.connect(self.db)
    cursor = conn.cursor()
    query = f"SELECT {column} FROM {table}"
    cursor.execute(query)
    result = cursor.fetchall()
    if table == "Contingency":
        combobox.addItem("None")
    for row in result:
        combobox.addItem(row[0])
    conn.commit()
    conn.close()
```

First, an SQLite database connection is opened. The specified column is selected from the corresponding table. Then, every result row is added to the corresponding combobox. If the table is the contingency table, a “None” option is also added.

```
self.scenario_cb = QComboBox()
self.add_data_to_combobox(self.scenario_cb, "`Scenario Name`", "Scenarios")
vlayout.addWidget(self.scenario_cb)

contingency_label = QLabel('Contingency')
contingency_label.setMaximumHeight(15)
vlayout.addWidget(contingency_label)

self.contingency_cb = QComboBox()
self.add_data_to_combobox(self.contingency_cb, "`Contingency Name`", "Contingency")
vlayout.addWidget(self.contingency_cb)
```

This is how this function is called in __init__.

retrieve_data()

This function retrieves data from the database to find information about the violating situations for the selected scenario and contingency.

```
# GET THE SEASON
query = f"SELECT `Season` FROM `Scenarios` WHERE `Scenario Name` = \"{self.scenario}\";"
cursor.execute(query)
self.season = cursor.fetchall()[0][0]
if self.season == "Winter":
    self.branch_table_index["Rating (Amps)" ] = 8
else:
    self.branch_table_index["Rating (Amps)" ] = 7
```

Here we retrieves the Season according to the Scenario. The branch_table_index will be updated according to the season for the Rating (Amps).

```
# VOLTAGE TABLES
query = f"SELECT `Bus Number`, bus_pu FROM `Bus Simulation Results` WHERE `Scenario Name` = \"{self.scenario}\" and `Contingency Name`
cursor.execute(query)
self.bus_data = cursor.fetchall()

for i in range(len(self.bus_data)):
    query = f"SELECT `Bus Name`, `Voltage Base`, `criteria_nlo`, `criteria_nhi` FROM BUS WHERE `Bus Number` = \"{self.bus_data[i][0]}\"
    cursor.execute(query)
    bus_result_part = cursor.fetchall()
    self.bus_data[i] += bus_result_part[0]

query = f"SELECT COUNT(`Bus Number`) FROM `BUS`;"
cursor.execute(query)
self.num_voltage = cursor.fetchall()[0][0]
```

This retrieves data from the Bus Simulation Results table, with the criteria that the scenario and contingency name match the selected ones, and where there is a violation but no exception. Then, for each bus selected, more information is retrieved from the Bus table about each bus. This data is saved into self.bus_data. Finally, the number of buses is retrieved from the Bus table using an aggregate function. Similar logic is applied to retrieve branch and 2-winding transformers data.

```
self.generate_report()
self.doc.generate_tex("tex")
self.display_report("tex.tex")
conn.close()
```

After data is retrieved and stored in self.bus_data, ,self.branch_data, and self.trans2_data. the generate_report() function is called which creates a latex formatted document utilizing the pylatex library. Then a .tex document is exported from it, arbitrarily called tex.tex. The display report function is called which will display the data in HTML form.

generate_report()

```
with self.doc.create(Section(f"Contingency: {self.contingency} (Islands created: ?)", False)):
    with self.doc.create(Subsection(f"Total number of MVar margin criiteria screened: 0", False)):
        self.doc.append("No violations.")
    with self.doc.create(Subsection(f"Total number of monitored buses: {self.num_voltage}", False)):
        if (len(self.bus_data) != 0):
            self.create_table("Bus Voltage", self.bus_data, self.bus_table_index)
        else:
            self.doc.append("No violations.")
    with self.doc.create(Subsection(f"Total number of monitored generators: 0", False)):
        self.doc.append("No violations.")
    with self.doc.create(Subsection(f"Total number of OOS margin criteria screened: 0", False)):
        self.doc.append("No violations.")
    with self.doc.create(Subsection(f"Total number of monitored branches: {self.num_thermalbranch}", False)):
        if (len(self.branch_data) != 0):
            self.create_table("Branch Thermal", self.branch_data, self.branch_table_index)
        else:
            self.doc.append("No violations.")
    with self.doc.create(Subsection(f"Total number of monitored two winding transformers: {self.num_trans2}", False)):
        if (len(self.trans2_data) != 0):
            self.create_table("Two Winding Transformer Thermal", self.trans2_data, self.trans2_table_index)
        else:
            self.doc.append("No violations.")
    with self.doc.create(Subsection(f"Total number of monitored three winding transformers: 0", False)):
        self.doc.append("No violations.")
```

This formats the latex document using pylatex functions. A main section listing the Contingency is created, then subsections for the buses, branches, and 2-winding transformers. If the self.data is empty, then no violations are found and it will be reflected as such. Otherwise, violations have been found so the create_table function is called todisplay the data in a table. Please refer to the pylatex documentation (linked) for further clarification.

create_table()

```
def create_table(self, table_name, data, index):
    columns = self.column_names[table_name]
    table_str = '| '
    for i in range(len(columns)):
        if columns[i] == "Branch Name":
            table_str += 'p{3cm} |'
        elif columns[i] == "Bus Name":
            table_str += 'p{4cm} |'
        elif columns[i] == "Transformer Name":
            table_str += 'p{1.7cm} |'
        elif columns[i] == "ID":
            table_str += 'p{0.3cm} |'
        else:
            table_str += ' X |'
```

The columns are retrieved from the dictionary using the table_name. The table is formatted in the convention outlined here, in a form that of '| c | c | c |', with ' | ' to represent vertical lines. Then, there is a list of edge cases for certain columns that tend to have longer names than other columns. Columns not specified receive ' X | ', which means that the width will be distributed evenly between the columns with that notation.

```
    with self.doc.create(Tabularx(table_str)) as table:
        table.add_hline()
        table.add_row([MultiColumn(len(columns), align='|c|', data=f"{table_name} Violations")])
        table.add_hline()
        if table_name == "Two Winding Transformer Thermal":
            table.add_row([
                "Transformer Name", MultiColumn(2, align='|c|', data="Bus Number"), "ID", "Base (MVA)", MultiColumn(3, align='|c|', data="Winding 1"), MultiColumn(3, align
            ])
            table.add_hline(2, 3)
            table.add_hline(6, 11)
            table.add_row(["", "Winding 1", "Winding 2", "", "", columns[5], columns[6], columns[7], columns[5], columns[6], columns[7]])
        else:
            table.add_row(columns)
```

Here, we begin the creation of the table. We specify a special case for the two-winding transformer thermal, because of the way that the table is formatted with overlapping multi-columns. Otherwise, the columns are added simply sequentially.

```
        for i in range(len(data)):
            table.add_hline()
            data_row = []
            for j in range(len(columns)):
                temp = index[columns[j]]
                if isinstance(temp, list):
                    if j > 7 and table_name == "Two Winding Transformer Thermal":
                        temp = index[columns[j]][1]
                    else:
                        temp = index[columns[j]][0]
                data_row.append(data[i][temp])
            table.add_row(data_row)
            table.add_hline()
```

Then, the data is added line by line. There is another special case for the 2-winding transformer tables. Data needs to be added to the table row by row, so the data_row list concatenates all of the data together.

display_report()

```
def display_report(self, tex_file):
    output = pypandoc.convert_file(tex_file, 'html', format='latex')
    with open("report.html", 'w') as f:
        f.write(output)
    file_path = os.path.abspath(os.path.join(os.path.dirname(__file__), "report.html"))
    local_url = QUrl.fromLocalFile(file_path)
    self.webView.setUrl(local_url)
    if os.path.exists(tex_file):
        os.remove(tex_file)
```

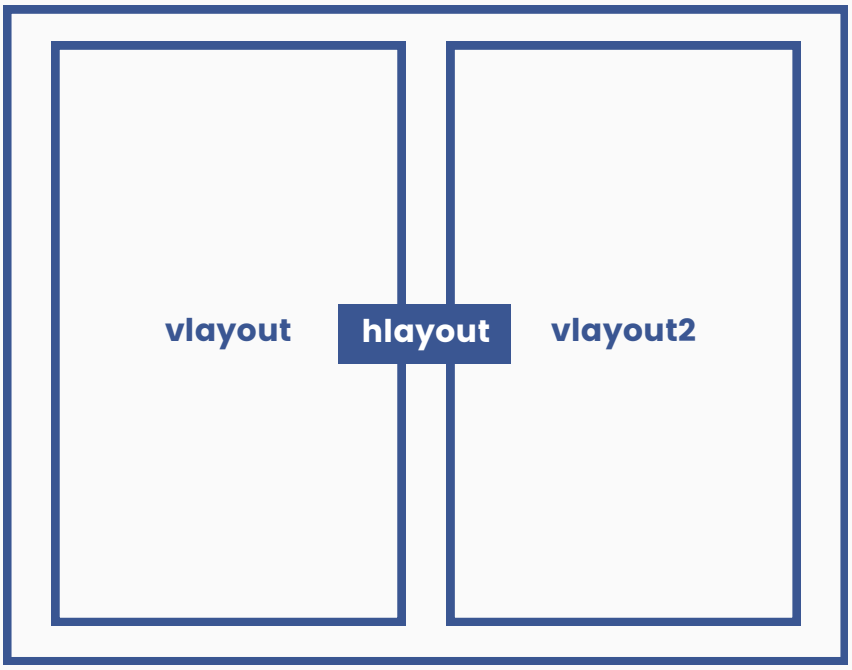
We chose to display the report preview in HTML format due to the slow conversion of a PDF file. We utilize the pypandoc library to convert the latex file into an html file “report.html”. We then take the path of the file and display it in the webview. The generated latex formatted file is deleted to reduce clutter in the directory.

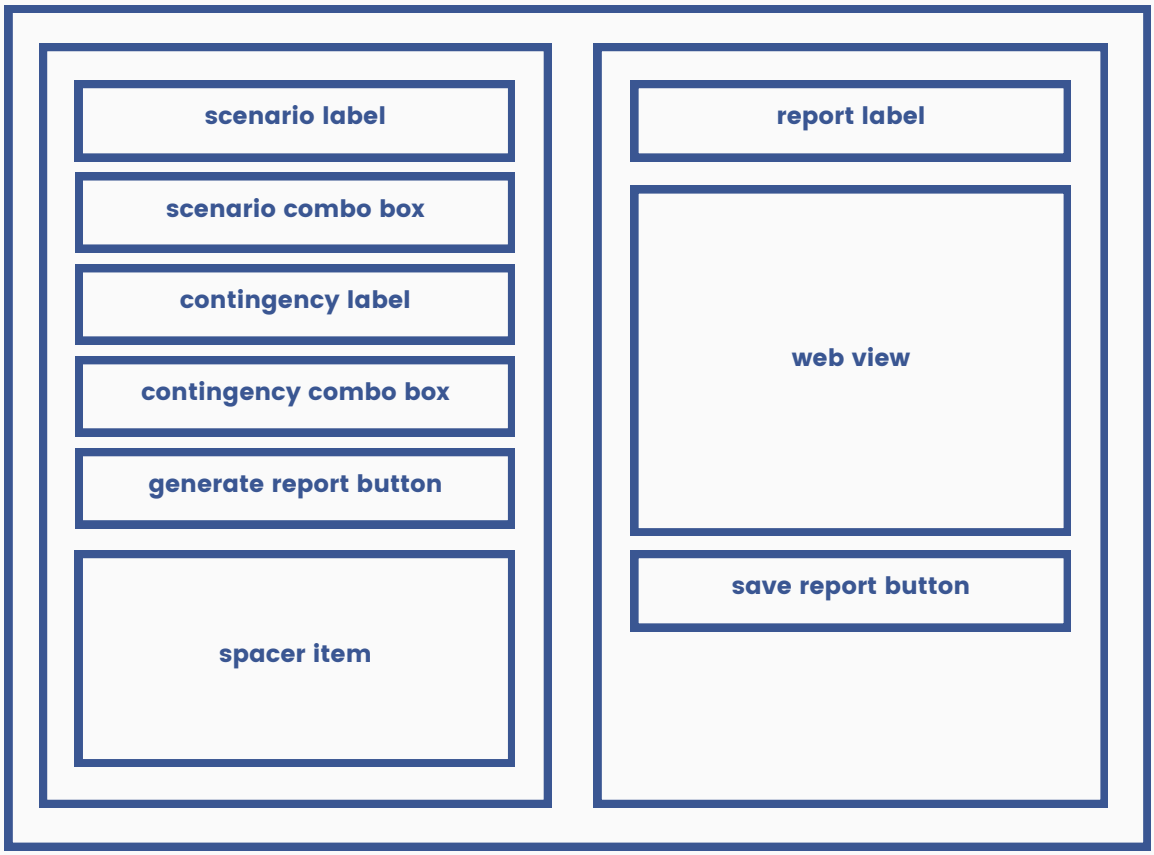
save_report()

```
def save_report(self):
    dialog = QInputDialog()
    text, ok = dialog.getText(self, '>.<', 'Name file (required):')
    dir= QFileDialog.getExistingDirectory(self, "Choose folder to save report in", ".")
    if ok and dir:
        path = dir + '/' + text
        self.doc.generate_pdf(filepath=path, clean_tex=False)
        if os.path.exists("report.html"):
            os.remove("report.html")
        self.done_alert()
```

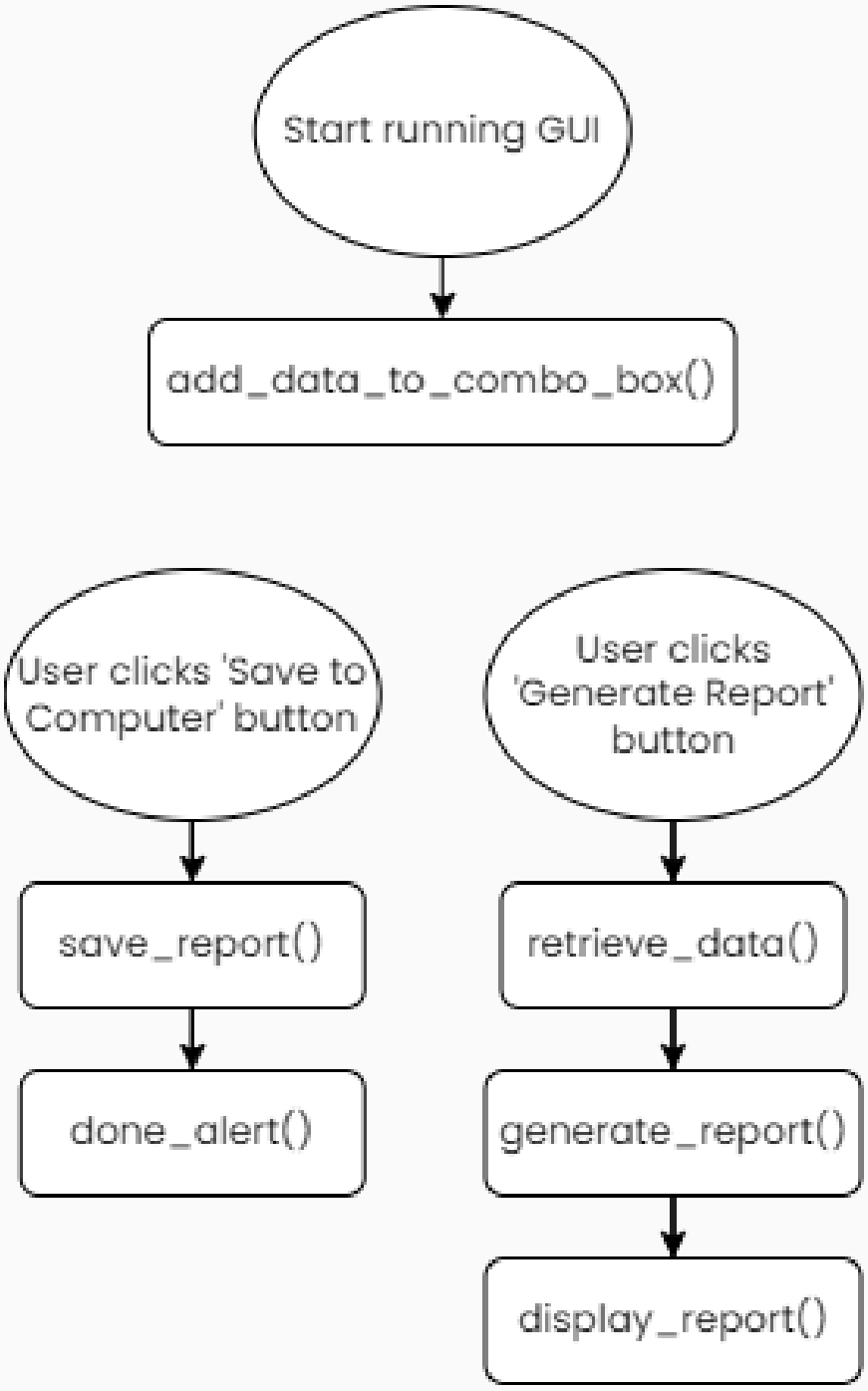
This displays the dialog to name the file and select the directory in which the file will be saved. QInputDialog and QFileDialog are initialized to accomplish these purposes. The path of the file is created by concatenating the selected directory and the desired name of the file. pylatex generate_pdf function is called which will generate a pdf from the latex formatted document. report.html is also removed for cleanliness. Once the pdf is generated, done_alert function is called, which will alert the user that the pdf has been generated

Format of the GUI in
MAIN . PY





Business Logic of
MAIN . PY



REFERENCES

Using SQLite - Jay A. Kreibich (accessed through O'Reilly)
Link to Canva here (to edit documentation)

Contact

Created by Amy Ding (2024 Engineering Co-op) with guidance from Gordon Wei and Steve Shelemy
Please feel free to direct any questions to amyding04@gmail.com