

# METHODOLOGY FOR LIGHTNING PERFORMANCE IMPROVEMENT

by

**Matthieu Bertin**

B.S. in Electrical Engineering, INSA de Lyon, 2015

Submitted to the Graduate Faculty of  
the Swanson School of Engineering in partial fulfillment  
of the requirements for the degree of  
Master of Science

University of Pittsburgh

2017



UNIVERSITY OF PITTSBURGH  
SWANSON SCHOOL OF ENGINEERING

This thesis was presented

by

Matthieu Bertin

It was defended on

September 28, 2017

and approved by

Dr. Brandon Grainger, PhD., Assistant Professor, Department of Electrical and Computer  
Engineering

Dr. Alexis Kwasinski, PhD., Associate Professor, Department of Electrical and Computer  
Engineering

Dr. Gregory Reed, PhD., Professor, Department of Electrical and Computer Engineering

Thesis Advisor: Dr. Brandon Grainger, PhD., Assistant Professor, Department of  
Electrical and Computer Engineering



Copyright © by Matthieu Bertin  
2017



# **METHODOLOGY FOR LIGHTNING PERFORMANCE IMPROVEMENT**

Matthieu Bertin, M.S.

University of Pittsburgh, 2017

A counterpoise ground model is added to EPRIs open source transient simulator. The approach to integrate this model is based on nodal analysis of the counterpoise conductor and electrical transient computations. A new graphical user interface for OpenEtran is added and described in this thesis. The user-interface regroups a tab window replacing the previous Excel spreadsheet interface for OpenEtran, and adds a line visualization tool for more efficient line design in terms of shield wires, grounds, insulation and line arresters.



## TABLE OF CONTENTS

<b>PREFACE</b> . . . . .	ix
<b>1.0 INTRODUCTION</b> . . . . .	1
<b>2.0 LITERATURE REVIEW OF THE PREVIOUS COUNTERPOISE RE- SEARCH EFFORTS</b> . . . . .	4
<b>3.0 C-PROGRAMMING AND COUNTERPOISE MODEL</b> . . . . .	6
3.1 INTRODUCTION . . . . .	6
3.2 THEORETICAL MODEL OF THE COUNTERPOISE . . . . .	7
3.3 IMPLEMENTATION OF THE MODEL . . . . .	9
3.3.1 THE GSL LIBRARY . . . . .	9
3.3.2 THE IMPLEMENTATION PROCESS . . . . .	9
<b>4.0 THE OPENETRAN GUI</b> . . . . .	14
4.1 INTRODUCTION . . . . .	14
4.2 OPENETRAN WINDOW . . . . .	14
4.2.1 USER LEVEL DESCRIPTION . . . . .	14
4.2.2 CODE ORGANIZATION AND INTERNAL CALCULATIONS . . . . .	18
4.2.2.1 INITIALIZATION OF THE APPLICATION: . . . . .	18
4.2.2.2 ADDING AND DELETING WIDGETS DYNAMICALLY: . . . . .	19
4.2.2.3 SAVING AND LOADING A PROJECT: . . . . .	19
4.2.2.4 CALCULATING THE COUNTERPOISE LOW CURRENT RESISTANCE . . . . .	23
4.2.2.5 SIMULATING A PROJECT . . . . .	23
4.3 VISUALIZATION TOOL . . . . .	26



4.3.1 USER LEVEL DESCRIPTION . . . . .	26
4.3.2 CODE ORGANIZATION AND INTERNAL CALCULATIONS . . . .	28
4.3.2.1 INITIALIZATION OF THE APPLICATION: . . . . .	28
4.3.2.2 PAINTING ELEMENTS ON THE DRAW VIEW: . . . . .	28
4.3.2.3 CALCULATING THE FLASHOVER RATE . . . . .	34
<b>5.0 CASE STUDIES . . . . .</b>	<b>41</b>
5.1 NEPC230 - NEW ENGLAND 230kV STEEL . . . . .	41
<b>6.0 CONCLUSION . . . . .</b>	<b>43</b>
<b>BIBLIOGRAPHY . . . . .</b>	<b>44</b>



## LIST OF FIGURES

1.1	Distribution line model in OpenEtran . . . . .	1
2.1	Counterpoise effect on the voltage distribution around a fault [6] . . . . .	4
3.1	Physical representation of a counterpoise . . . . .	6
3.2	Distributed model of the counterpoise . . . . .	7
3.3	C-code declaration of all counterpoise fields . . . . .	10
3.4	Memory allocation code . . . . .	10
3.5	C-code to solve for the counterpoise voltages . . . . .	12
3.6	Counterpoise with compensation current source . . . . .	13
4.1	Excel spreadsheet for OpenEtran . . . . .	15
4.2	OpenEtran GUI - Simulation tab . . . . .	15
4.3	Plots results in one-shot mode simulation . . . . .	16
4.4	OpenEtran GUI - Component tab . . . . .	17
4.5	addWidget function . . . . .	18
4.6	PyQt Slots . . . . .	19
4.7	Add/Delete Widgets . . . . .	20
4.8	Main GUI structure . . . . .	21
4.9	Read Widgets . . . . .	22
4.10	Input parser function . . . . .	24
4.11	Subprocess run function . . . . .	25
4.12	OpenEtran call arguments in critical current mode . . . . .	25
4.13	Phase visualization tool . . . . .	27
4.14	Rescaled viewing section . . . . .	31



4.15 Function to draw a phase . . . . .	33
4.16 <i>drawArcs</i> function . . . . .	33
4.17 <i>drawObjects</i> function . . . . .	35
4.18 Critical current file . . . . .	35
4.19 Critical current parser . . . . .	36
5.1 NEPC230 Ground currents . . . . .	41
5.2 NEPC230 Arrester voltages . . . . .	42



## PREFACE

First, I would like to thank the International Relations team at INSA de Lyon and Dr. Mahmoud El Nokali for giving me the opportunity to study at the Swanson School of Engineering.

I also thank my advisor Dr. Brandon Grainger for his guidance during my time in the Electrical Power Systems Laboratory, along with Dr. Thomas McDermott (now with Pacific Northwest National Laboratory) for his great help during the duration of this project. I also thank Dr. Gregory Reed and Dr. Alexis Kwasinski for serving on my thesis committee.

I thank everyone in the laboratory: Patrick Lewis, JJ Petti, Hashim Al Hassan, Ansel Barchowsky, Alvaro Cardoza, Chris Scioscia, Santino Graziani, Jake Friedrich, Andrew Bulman, Samantha Morello, Dr. Katrina Kelly-Pitou and Carrie Snell for their heart warming welcome they gave me when I first arrived in the United States, and for maintaining a relaxed and friendly work environment throughout the length of my stay.

Finally, I want to thank my parents and my brother, Sylvie, Patrice and Thomas Bertin, for their love and encouraging support.

This work was sponsored by CEATI International Inc. through project number GLIG 3720: Methodology for Lightning Performance Improvement.



## 1.0 INTRODUCTION

Lightning is a significant cause of outages and damage to power systems all over the world. It is necessary to protect transmission and distribution lines against these strikes, by adding shielding devices such as surge arresters, insulators or grounding systems. This study is based around the transient simulation engine of Electric Power Research Institute (EPRI)'s Lightning Protection Design Workstation (LPDW): OpenEtran. This software has been released under an open-source license (GPL v3) in 2002 so it can be added to other projects, such as IEEE Flash [1]-[2]. Fig. 1.1 shows a typical overhead line system that can be simulated in OpenEtran [1]. This line has several poles, with the neutral wire being grounded at the different poles. Some of the optional components (arrester, insulator etc.) that can be added to the system are also shown in this figure.

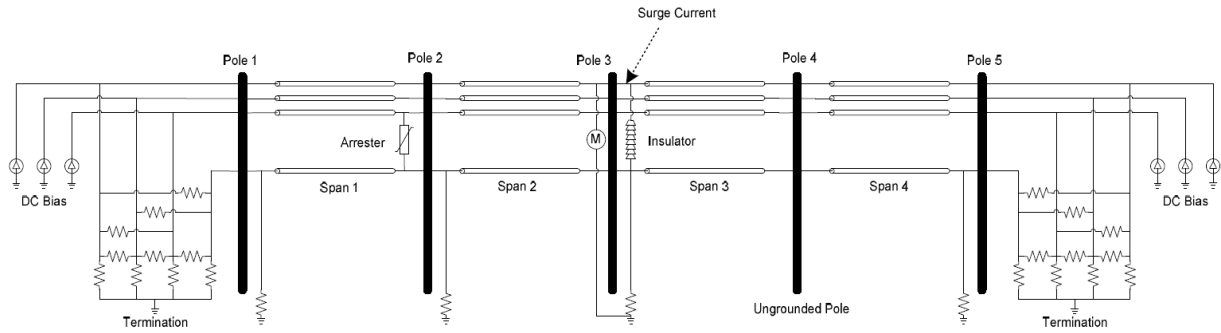


Figure 1.1: Distribution line components model in OpenEtran [1]

The capabilities of OpenEtran include:

- Calculating critical current magnitudes that can cause flashovers at different locations on the line.



- Analysing insulators, surge arresters or grounds with time-dependent or nonlinear models.
- Modeling traveling wave models for transmission lines using the Bergeron/Dommel method of the Electromagnetic Transient (EMT) software [3]-[4].
- Using several lightning stroke shapes, for first and subsequent strokes.

However, unlike EMT softwares, OpenEtran only focuses on lightning transients and does not address steady-state analysis, switching event surges, power electronics or control effects on systems.

This project has been funded by CEATI International Inc., with an aim to provide a more efficient and user-friendly lightning analysis tool, to simplify the benchmarking of lightning performance results to predictions and to promote updates to the IEEE design guide for transmission line lightning protection. The two tasks that were performed in this work are: adding a new grounding model to the OpenEtran kernel (counterpoise model) and creating a new Graphical User Interface (GUI) in order to make the software more user-friendly and efficient. The short-term benefits of this work include:

- The addition of a more efficient and user-friendly tool for lightning performance analysis, as compared to general-purpose EMT programs. This should enable broader use of EMT methods for lightning analysis in utility companies.
- Better evaluation of the impact of pending changes to surge arresters standards. New versions of IEC 60099 specify arresters duty in terms of current, charge and energy in varying circumstances. This enables more comprehensive evaluation of line trip-out rates vs. line arrester capabilities and failure rates.
- Designing and benchmarking lightning performance improvement programs. Given the stochastic variations in lightning activity and severity, accurate predictions and cost estimates become critical.
- Promote updates to the IEEE guide for transmission line lightning protection [5]. The present version of this guide does not address transmission line arresters, nor does it address EMT modeling for lightning analysis.



In this report, the work put forth to implement the counterpoise ground model into OpenEtran is described, theoretically and from a practical standpoint. Next the new GUI is introduced, with its capacities first described at user-level, then more in details.



## 2.0 LITERATURE REVIEW OF THE PREVIOUS COUNTERPOISE RESEARCH EFFORTS

When doing classic circuit analysis, we consider the earth to be a perfect medium with no resistance and that can dissipate any fault current. This is not the case in reality, as earth also has a resistance and voltage rises can be experienced when intense fault current from a power system or from a lightning bolt leak into the ground. This can cause serious safety issues, as the voltage levels can be dangerously high if the earth has a high resistivity, which proves to be quite common.

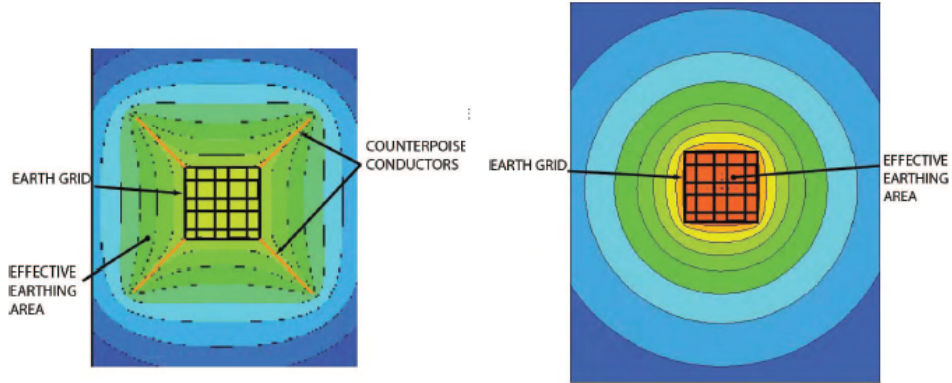


Figure 2.1: Counterpoise effect on the voltage distribution around a fault [6]

A solution to this issue is to install an earth grid around a power system. An earth grid is made of several horizontal and vertical metal rods, which are used to reduce the footing impedance and the voltage levels around the fault. However, due to the proximity of the different conductors the performances of an earth grid can quickly saturate, hence



diminishing the safety level of the installation [6]. A way to avoid that problem is to install horizontal grounding rods, with length ranging from several meters to tens or hundreds of meters, and connect them to the earth grid. These conductors are called counterpoise rods. When directed away from the earth grid, they allow the creation of a larger effective earth grid (see Fig. 2.1), hence increasing the overall performances of the system in avoiding overvoltages to nearby structures.

Ground work on the study of the counterpoise conductors started in the 1930s, with research papers from L. V. Bewley [8] and Charles L. G. Fortescue [7]. At this time, counterpoises were installed mainly to increase lightning protection of transmission lines [8]. The first studies of the counterpoise effects were mostly experimental, where the performances of several counterpoise systems (crowfoot, parallel conductors [7]) were measured first, then empirical models were designed.

A few years later, a substantial research effort was made to find a precise analytical model of the surge response of grounding systems that accounts for soil ionization. Indeed, lightning surge currents induces an ionization of the soil, which makes the area around the grounding rod electrically conductive and changes the surge impedance [9]. This greatly increases the difficulty of the analysis. When accounting for the soil ionization, a trend when developing counterpoise models is to treat the conductor as a transmission line with lumped parameters. It is done in [10], [11] or [12]. [11] also takes into account the frequency-dependent aspect of ground resistivity. This aspect was not considered in the model implemented in OpenEtran, but it can be part of a future work effort on the project.



### 3.0 C-PROGRAMMING AND COUNTERPOISE MODEL

#### 3.1 INTRODUCTION

In this section, all the work that was done to integrate the counterpoise model into OpenEtran is presented. First, the theoretical model used to describe the counterpoise behavior under lightning transient conditions is explained, then the different steps to implement this model are described. Finally, the organization of the code itself within OpenEtran is given in detail.

A counterpoise is a grounding system consisting of one or several buried horizontal conductors, each of them linked to a transmission tower so as to reduce its footing impedance [8] by allowing the lightning impulse current to flow more easily into the ground. It is also used in telecommunications as a ground substitute for antennas when the earth has a high resistivity [12]. Counterpoise conductors have a length that can range from a few meters to more than a hundred meters.

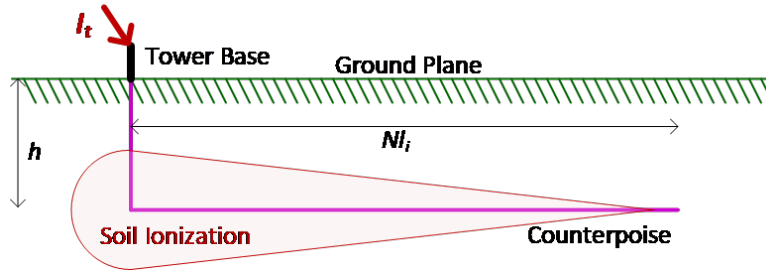


Figure 3.1: Physical representation of a counterpoise under lightning conditions



### 3.2 THEORETICAL MODEL OF THE COUNTERPOISE

The previous grounding model in OpenEtran was limited to vertical rods with soil ionization, following the model originated by K.-H. Weck [13]. It is described by (3.1) and (3.2).

$$I_{brk} = \frac{\rho E_0}{2\pi R_{60}^2} \quad (3.1)$$

$$R_G = \frac{R_{60}}{\sqrt{1 + \frac{I}{I_{brk}}}} \quad (3.2)$$

$R_{60}$  is the low-current ground resistance,  $\rho$  is the soil resistivity,  $E_0$  is the soil breakdown gradient,  $I$  is the iterated peak ground current, and  $R_G$  is the iterated high-current ground resistance. Frequency-dependent effects could be approximated with lumped L and C elements, but this is not presently done in OpenEtran.

As stated before, the purpose of this work is to add another common lightning protection system: the counterpoise conductor. The model that was used to describe its behavior under lightning impulse conditions is the one developed in [12]. It represents the grounding electrode as a transmission line with distributed lumped parameters, as shown in Fig. 3.2.

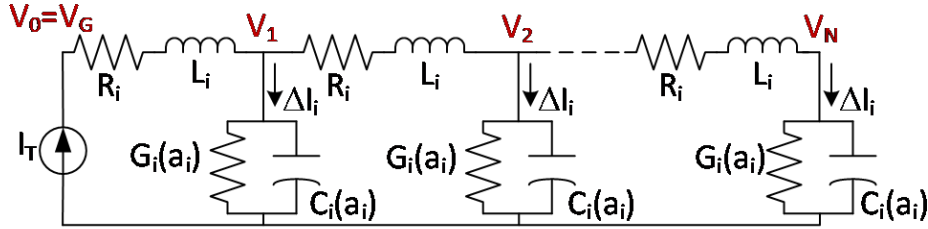


Figure 3.2: Distributed model of the counterpoise [12]

The components associated with the counterpoise model can be calculated with (3.3)-(3.5) from [12].

$$L_i \simeq \frac{\mu_0 l_i}{2\pi} \left( \ln \frac{2l_i}{a} - 1 \right) \quad (3.3)$$

$$C_i = C(a_i) + C(2h - a_i) \quad (3.4)$$

$$G_i = \frac{C_i}{\epsilon \rho} \quad (3.5)$$



Where  $\mu_0$  is vacuum's magnetic permeability,  $l_i$  is the length of a segment of the counterpoise,  $a$  is the radius of the counterpoise conductor,  $\rho$  is the resistivity of earth,  $h$  is the depth at which the counterpoise is buried,  $a_i$  is the radius of the ionized zone around the counterpoise and  $\epsilon$  is the electric permittivity of earth.

In developing the user interface for this model, we used a ladder network reduction formula from [14] to solve for  $G_i$  and  $R_i$  that would match the users input low-frequency counterpoise resistance, which is known from measurement. After a review of [15] and other references, we used these  $R_i$  values instead of the much higher ones calculated from [12].

Equation (3.4) gives the capacitance of a segment when the conductor is buried at a depth of  $h$ . It is a function of the capacitance of that same segment in an infinite medium, which is defined as:

$$C(a_i) = \frac{2\pi\epsilon l_i}{\frac{a_i}{l_i} + \ln \frac{l_i + \sqrt{l_i^2 + a_i^2}}{a_i} - \sqrt{1 + \left(\frac{a_i}{l_i}\right)^2}} \quad (3.6)$$

Finally, the ionized zone radius can be obtained using the value of the leaked current into the earth at each segment [12]:

$$a_i = \frac{\Delta i_i \rho}{2\pi E_C l_i} \quad (3.7)$$

In (3.7),  $\Delta i_i$  is the leaked current into the earth and  $E_C$  is the critical electrical field gradient at which the soil starts to ionize. It is typically 300kV/m [12].

The important aspect of this model is that, since it takes the soil ionization into account, it is highly nonlinear. Indeed, the shunt capacitance and conductance in (3.4) and (3.5) are a function of the ionization radius, which changes continuously during the impulse. *Hence, the values of  $C_i$  and  $G_i$  will also vary during the analysis and the computer model will need to be updated at each time step during a simulation.* In this model, the ground resistivity  $\rho$  is considered uniform along the whole length of the line and is constant. From these assumptions, the resistance  $r_i$  and inductance  $L_i$  are constant over time. They are also the same for each segment.



### 3.3 IMPLEMENTATION OF THE MODEL

#### 3.3.1 THE GSL LIBRARY

OpenEtran relies heavily on the GSL library, which is very commonly used as it regroups many tools to perform a large panel of operations over different subjects, such as: linear algebra, complex numbers, large vectors and matrices management, fast Fourier transform [16] etc. This tool is perfectly adapted to the counterpoise project, since it is needed to perform large matrices transforms, decompositions and linear systems solving. Using the library greatly simplifies the coding, as the operations like the LU decomposition of a matrix, which is used in the implemented algorithm (refer to section 3.3.2), can be performed by just calling the appropriate function. The library code is also optimized so the calculations are done much faster than if they had been recoded.

#### 3.3.2 THE IMPLEMENTATION PROCESS

The OpenEtran software gives the value of  $I_t$ , the current injected into the conductor at the beginning of the simulation. The goal of the counterpoise implementation is to simulate the evolution of the ground current by calculating the leakage rate into the earth. To perform this task, the voltages at each node must be calculated using nodal analysis, then the currents in each branch are computed using the impedances of each element.

The code for this implementation of the counterpoise model is separated, as it's usually done in C/C++ programming, between a header and code file. The general ground structure with all the data fields, along with all the function prototypes, is stored in the header file *ground.h*. The new data fields relative to the counterpoise are either floating point numbers containing counterpoise parameters, such as the radius or depth, or GSL vectors and matrices to store the voltages/currents at each node and the admittances. The complete list of new parameters is shown in Fig. 3.3.

When OpenEtran starts, the first step is to read the input file then to add all the parameters to the main ground structure. This is done by the function *add\_counterpoise*, analogous to the *add\_ground* function which already existed in OpenEtran. The point of



```

/* Counterpoise attributes and variables follow */
int counterpoise; /* 1 if there's a counterpoise, 0 otherwise */
double depthC; /* Depth in m. of counterpoise conductor */
double radiusC; /* Radius in m. of counterpoise conductor */
double lengthC; /* Total length in m. of counterpoise conductor */
double li; /* Length of 1 segment of counterpoise conductor */
int numSeg; /* Number of segments (RLC & G cells) in the counterpoise conductor */
double relPerm; /* Relative electrical permittivity of soil */
double e0; /* Critical field gradient (electric field intensity on the boundary of the soil ionized zone) */
double rho; /* Resistivity of soil */
/* solve a symmetric tridiagonal system of order numSeg+1 */
gsl_vector *yDiag; /* Diagonal elements */
gsl_vector *yOff; /* Off-Diagonal elements (will be constant) */
gsl_vector *voltage; /* Voltage at each node */
gsl_vector *current; /* Current at each node */
double ri; /* series elements do not depend on current */
double Li;
gsl_vector *Ci; /* shunt elements in each segment depend on current */
gsl_vector *Gi;
gsl_vector *hRL; /* History current in series ri, Li */
gsl_vector *hC; /* History current in shunt Ci */

```

Figure 3.3: C-code declaration of all counterpoise fields

adding a separate function was to minimize the structural changes to the original program and address the fact that the counterpoise addition is optional, so this new function will not be called if not all the mandatory parameters are present in the file. In this case the original model of (3.1) is used.

Once the structure is updated, the program allocates memory space to store the different vectors and matrices, such as the system's admittance matrix. This is done using built-in GSL functions, which return a pointer to a vector or matrix, as shown in Fig. 3.4.

```

/* Allocations */
ptr->ybus = gsl_matrix_alloc(numSeg, numSeg);
ptr->yTri = gsl_matrix_alloc(numSeg, numSeg);
ptr->yPerm = gsl_permutation_alloc(numSeg);

ptr->voltage = gsl_vector_calloc(numSeg);
ptr->current = gsl_vector_calloc(numSeg);

ptr->Ci = gsl_vector_alloc(numSeg);
ptr->Gi = gsl_vector_alloc(numSeg);

ptr->hist = gsl_vector_calloc(numSeg);

```

Figure 3.4: C-code for matrices memory allocations



The next step in the implementation is to calculate the conductor's admittance ( $Y_{bus}$ ) matrix in preparation for solving (3.8). From Fig. 3.2, this will be symmetric tridiagonal.

$$Y_{bus} \cdot I = V \quad (3.8)$$

From the trapezoidal integration formulas in [4], the admittance contributions at each segment can be computed with (3.9) and (3.10).

$$Y_{series} = \frac{1}{R_i + \frac{2L_i}{\Delta t}} \quad (3.9)$$

$$Y_{shunt} = G_i + \frac{2C_i}{\Delta t} \quad (3.10)$$

With  $\Delta t$  being the simulation time step. The values for the capacitance and conductance are initially calculated using  $a_i$  equal to the conductor radius  $a$  for the first simulation time step. Off-diagonal elements in (3.8) are all equal to  $-Y_{series}$ , and remain constant through the simulation. The first diagonal element is  $Y_{series}$ , the last one is  $Y_{series} + Y_{shunt}$ , and all others are  $2Y_{series} + Y_{shunt}$ . These diagonal elements can change during simulation according to (3.4)-(3.7).

This concludes the initialization steps. Now the program starts the simulation process and enters the main function *check\_ground* in which the counterpoise model is solved. In this function, the program first gets the injected voltage from the pole into the grounding system, which is also the voltage at node 0 of the electrode. From this initial voltage, the current at node 0 is determined using the pole admittance and the counterpoise's current at the previous time step. This value is stored in the current vector in the main ground structure. The program then solves the system in (3.8) using the built-in GSL function *gsl\_linalg\_solve\_symm\_tridiag* and updates the voltage vector.

Once the voltages have been calculated using nodal analysis, the program enters the function *updateModel* in which the currents through each branch of the network are updated for the given time step. The right-hand side of (3.8) consists of the tower current,  $I_T$ , injected at node 0, along with trapezoidal integration history currents at all nodes from the series  $R_i$  and  $L_i$  components (flowing left to right) and  $C_i$  components (node to ground). Initially



```

/* see if the ground resistance is reduced by impulse current flow */
void check_ground (struct ground *ptr) {
    double It, Vg, Vt, Imag, Vt;

    /* Voltage at node 0 */
    Vt = gsl_vector_get (ptr->parent->voltage, ptr->from) - gsl_vector_get (ptr->parent->voltage, ptr->to);

    /* Total ground current */
    It = Vt * ptr->y + ptr->i;
    ptr->amps = It;
    Imag = fabs (It);

    /* Counterpoise model if there's a counterpoise at the base of each pole */
    if (ptr->counterpoise) {
        /* We start by solving for the voltages at each node; history currents already loaded */
        *gsl_vector_ptr(ptr->current, 0) += It; /* current injection from the balance of system */
        gsl_linalg_solve_symm_tridiag(ptr->yDiag, ptr->yOff, ptr->current, ptr->voltage);
        Vg = gsl_vector_get(ptr->voltage, 0); /* counterpoise/tower base voltage */

        /* Update the components, history currents and Ybus diagonals for the next time step */
        updateModel(ptr);
    }
}

```

Figure 3.5: C-code to solve for the counterpoise voltages

these are all zero. When (3.8) is solved at a time step, the actual series and shunt currents are found from (3.11)-(3.13).

$$I_{RLi}(t) = Y_{series} (V_i(t) - V_{i+1}(t)) + h_{RL}(t)(t - \Delta t) \quad (3.11)$$

$$I_{Ci}(t) = \frac{2C_i}{\Delta t} \cdot V_i(t) + h_{Ci}(t - \Delta t) \quad (3.12)$$

$$\Delta I_i(t) = I_{Ci}(t) + G_i V_i(t) \quad (3.13)$$

Where  $h_{RLi}$  and  $h_{Ci}$  are the previous trapezoidal history currents [4]. For the next time step, the leaked segment currents from (3.13) update (3.4)-(3.7) and (3.10). The history currents are then updated from (3.14)-(3.15) [4].

$$h_{RLi}(t) = Y_{series} \left[ \left( \frac{2L_i}{\Delta t} - R_i \right) I_{RLi}(t) + V_i(t) - V_{i+1}(t) \right] \quad (3.14)$$

$$h_{Ci}(t) = -I_{Ci}(t) - \frac{2C_i}{\Delta t} V_i(t) \quad (3.15)$$

The first node voltage obtained from (3.8), namely  $V_0$ , approximates the counterpoise voltage,  $V_G$ , in Fig. 3.2. They won't match because of nonlinearities in the model. We also have to account for the tower surge impedance, approximated with inductance  $L_T$  in Fig. 3.6. The tower-top voltage is then  $V_T$ . In OpenEtran, the tower and ground impedances are combined in a single element, connected to a shield or neutral.



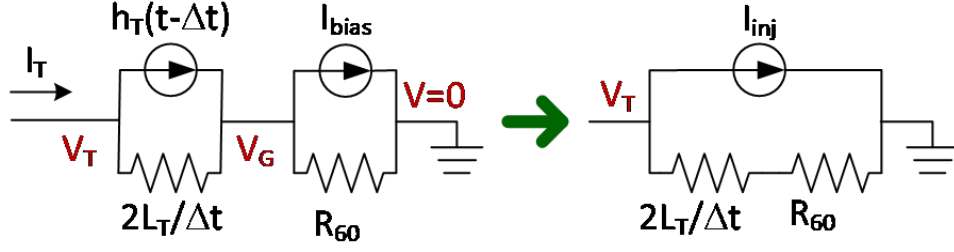


Figure 3.6: Counterpoise with compensation current source

The circuit in Fig. 3.6 is connected to the rest of the system. In order to make this represent the counterpoise model depicted in Fig. 3.2 and solved in (3.3)-(3.15), we use the compensation method [3]. A bias current, defined in (3.16), circulates through the nominal counterpoise resistance,  $R_{60}$ . This accounts for the counterpoise nonlinearities, with a lag of  $\Delta t$ . This lag is not harmful in the ground model because the current variations are relatively slow and smooth, compared to the surge arrester models, which OpenEtran solves by iteration within each time step.

$$I_{bias} = \frac{V_G}{R_{60}} - I_T = \frac{V_0}{R_{60}} - I_T \quad (3.16)$$

The tower inductor current and history terms in Fig. 3.6 are calculated as in (3.11) and (3.14), but with zero resistance. Using Kirchhoffs Voltage Law, we can eliminate the  $V_G$  node using (3.17) to arrive at the simpler model to the right in Fig. 3.6. OpenEtran uses this right-hand model to calculate the next time steps current injection,  $I_T$  in Fig. 3.2.

$$I_{inj} = \frac{h_T \cdot \frac{2L_T}{\Delta t} + I_{bias}R_{60}}{R_{60} + \frac{2L_T}{\Delta t}} \quad (3.17)$$

Finally, it is important to mention that the counterpoise model is an add-on and has not replaced the original ground rod model, which is described in the OpenEtran user manual. It is still possible to use the ground rod model defined by (3.1) and (3.2) by simply not filling in the counterpoise parameters part of the new user interface described next.



## 4.0 THE OPENETRAN GUI

### 4.1 INTRODUCTION

In this part, the new Graphical User Interface (GUI) for OpenEtran is described extensively. First, the report will focus on the user-level features of the program, then the next part dives into the technical details, for both the OpenEtran window and the visualization tool.

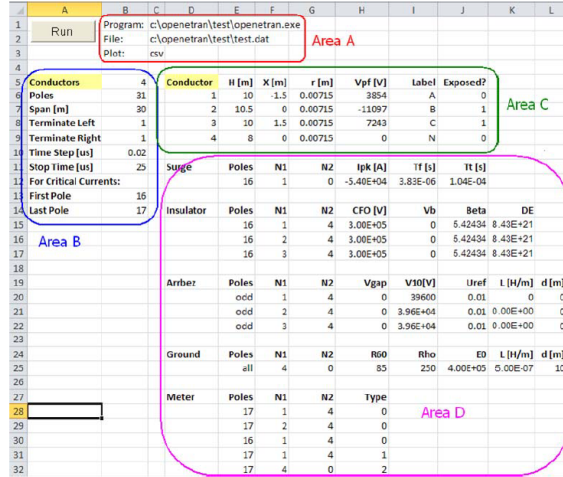
### 4.2 OPENETRAN WINDOW

#### 4.2.1 USER LEVEL DESCRIPTION

The original Excel interface for OpenEtran is described in [13]. As seen in Fig. 4.1, this spreadsheet interface contains four input regions: program and input/output files paths (area A), simulation parameters (area B), mandatory conductor parameters (area C) and optional component parameters (area D). This spreadsheet interface presents many issues: it crashes frequently, it is usable on Windows only and it does not make any interpretations of the output files, so the user needs to format the output data after each simulation to get the plots. This results in high losses of efficiency and productivity.

The new user-interface, programmed in Python v3.5, is separated in two windows: one replaces the Excel spreadsheet interface for OpenEtran (see Fig. 4.2), the other is a line visualization tool that allows the user to see the level of each conductor's exposure and vulnerability to lightning, with the possibility of calculating the flashover rate (see section 4.3).





Conductors	Conductor	H [m]	X [m]	r [m]	Vpf [V]	Label	Exposed?
Poles	1	10	-1.5	0.00715	3854	A	0
Span [m]	2	10.5	0	0.00715	-11097	B	1
Terminate Left	3	10	1.5	0.00715	7243	C	1
Terminate Right	4	8	0	0.00715	0	N	0

Surge	Poles	N1	N2	Ipk [A]	Tf [s]	Tt [s]
16	1			-5.40E+04	3.83E-06	1.04E-04

Insulator	Poles	N1	N2	CFO [V]	Vb	Beta	DE
16	1	4		3.00E+05	0	5.42434	8.43E+21
16	2	4		3.00E+05	0	5.42434	8.43E+21
16	3	4		3.00E+05	0	5.42434	8.43E+21

Arrbez	Poles	N1	N2	Vgap	V10[V]	Uref	L [H/m]	d [m]
oocd	1	4		0	39600	0.01	0	0
oocd	2	4		0	3.96E+04	0.01	0.00E+00	0
oocd	3	4		0	3.96E+04	0.01	0.00E+00	0

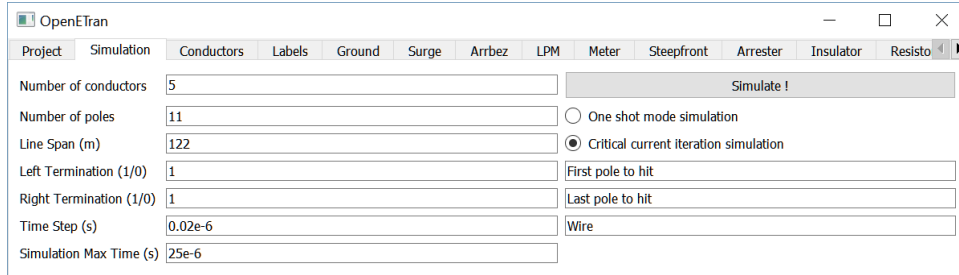
  

Ground	Poles	N1	N2	R60	Rho	EO	L [H/m]	d [m]
all	4			85	250	4.00E+05	5.00E-07	10

Meter	Poles	N1	N2	Type
17	1	4		0
17	2	4		0
16	1	4		0
17	1	4		1
17	4			2

Figure 4.1: Excel spreadsheet for OpenEtran



Project	Simulation	Conductors	Labels	Ground	Surge	Arrbez	LPM	Meter	Steepfront	Arrester	Insulator	Resisto
Number of conductors	5	<div>Simulate !</div> <div> <input type="radio"/> One shot mode simulation           <input checked="" type="radio"/> Critical current iteration simulation         </div> <div>First pole to hit</div> <div>Last pole to hit</div> <div>Wire</div>										
Number of poles	11											
Line Span (m)	122											
Left Termination (1/0)	1											
Right Termination (1/0)	1											
Time Step (s)	0.02e-6											
Simulation Max Time (s)	25e-6											

Figure 4.2: OpenEtran GUI - Simulation tab

Fig. 4.2 shows the tab window for the new OpenEtran interface. In this GUI, the first tab is the Project Tab, in which the user can save/load a project and switch between a simplified or full interface. The simplified interface only shows the most commonly used components: pole/phase labels, grounds, surges, arrbez, LPM and meters. One can refer to the OpenEtran manual [13] for complete description on the different components. The input files for the GUI are JSON files, where all components and simulation parameters are stored. During a simulation, the GUI first translates the project JSON file into a DAT file that is directly used by OpenEtran.



The tab shown in Fig. 4.2 regroups the simulation parameters. There are two possible types of simulation in OpenEtran: a *one-shot* mode with plot files and a *critical current iteration* mode.

In one-shot simulation mode, OpenEtran performs a time domain simulation of the system and writes in a CSV file the values of current / voltage at locations specified by the user (by placing **Meter** components on the line). Once the simulation is done, the GUI reads the output CSV file and plots the curves using the Matplotlib framework, which is very similar to the plotting framework of Matlab. An overview of the output curves in one-shot mode is shown in Fig. 4.3.

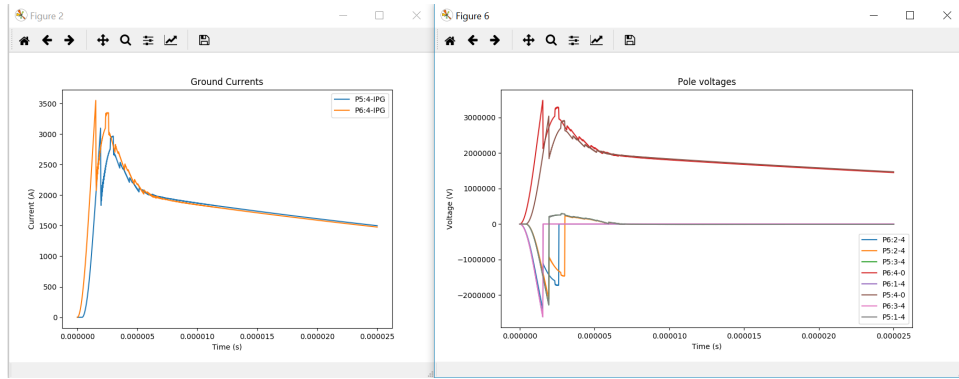


Figure 4.3: Plots results in one-shot mode simulation

The critical current mode forces OpenEtran to calculate the minimum lightning current value that can create an insulation back-flashover at a certain location on the line. To do the analysis, the user needs to specify which poles in the line need to be hit by lightning. This is done by defining a pole sequence using the *First pole to hit* and *Last pole to hit* text fields in the Simulation tab (see Fig. 4.2), and also by defining a wire sequence. This sequence tells OpenEtran which phases are to be considered in the analysis. For example, if the user enters "1 1 1 0 1" in the *Wire* text field for a five phase system, OpenEtran will calculate the critical current for phases 1, 2, 3 and 5. Phase 4 will be discarded in this example. To launch the simulation the user simply needs to press the button *Simulate*. The output values are then written in a text file, located in the same folder as the input file for OpenEtran (see section 4.2.2 for more detail).



The other tabs are all component tabs, similar to the one shown in Fig. 4.4. In these tabs, the user enters the necessary parameters for each component and can add/delete them dynamically using the appropriate push buttons. Only the conductors data is mandatory. The line must have at least one wire, but all the other components are optional. In every components tabs, the two common text fields are *Poles* and *Pairs*. The Poles text field specifies at which poles the component is placed, this input accepts either a single pole number, a sequence such as "1 2" for poles 1 and 2, or the words *even*, *odd* and *all* for even numbered poles, odd numbered poles or all poles respectively. The Pairs input specifies between which phases the component needs to be put. For example, a pairs input of "1 2" means the component is placed between phases 1 and 2. A pairs input of "1 0" means between phase 1 and the ground.

The only feature other than adding/deleting elements is in the Ground tab. It is possible to calculate the low current resistance, also called power frequency resistance (R60) for the counterpoise. For executing this calculation, the user simply needs to press the button *Get Counterpoise R60* in the ground tab. The value of low current resistance for each ground component is then updated in the R60 text field. More detail about this calculation is given in section 4.2.2.4.

OpenEtran			
Project Simulation Conductors Labels Ground Surge Arrebez LPM Meter Steepfront Arrester Insulator Resistor Capacitor Inductor Customer Pipegap			
New		Delete	
Get Counterpoise R60			
Ground			
R60 (Ohm)	200	Resistivity (Ohm.m)	100
Download Inductance (H/m)	0.5e-6	Length of download (m)	18
Counterpoise depth (m)	1	Counterpoise length (m)	122
Soil relative permittivity	9	Pairs	4 0
		Poles	all
		Soil Breakdown Gradient (V.m)	300e3
		Counterpoise radius (m)	0.01
		Number of segments	10

Figure 4.4: OpenEtran GUI - Component tab

Finally, once the user has finished entering all parameters, he needs to press the *Simulate* button in the Simulation tab. The GUI then opens a file selection window for the user to save his current project, and launches OpenEtran. The output files, CSV plot file for the "one-shot" mode or text file for the critical current mode, are written in the same folder as the input file.



## 4.2.2 CODE ORGANIZATION AND INTERNAL CALCULATIONS

The OpenEtran GUI is coded using PyQt5, which is a bridge between Python and the Qt C++ graphical library. In this framework, all components are widgets which can be linked to callback functions and organized using layouts.

The GUI is based around a Tab Widget. All tabs in this main widget are organized in the same fashion using a grid layout. With a grid layout, the user can access widgets by using a line and column number, which makes it very easy to organize a window and access the different components.

**4.2.2.1 INITIALIZATION OF THE APPLICATION:** The graphical application itself is based around a Python class, called *GUI\_Tab*. During the initialization of this class, each tabs of the GUI are created, with their corresponding text fields. Since all components tabs are organized in the same way, a generic function (shown in Fig. 4.5) was designed to add the widgets, as seen in Fig. 4.5. This function takes as arguments the widget's layout, the list of names for all the labels and the number of lines and columns in the layout.

```
# Function to add widgets on the tabs for the 1st time
def addWidgets(grid, names, rowEnd, ColEnd):
    positions = [(i,j) for i in range(rowEnd) for j in range(ColEnd)]
    for position, name in zip(positions, names):
        if name == '':
            widget = QLineEdit()

        elif name == 'New' or name == 'Delete' or name == 'Get Counterpoise R60':
            widget = QPushButton(name)

        elif name == '/':
            widget = QLabel()

        else:
            widget = QLabel(name)

        grid.addWidget(widget, *position)
```

Figure 4.5: Function to add widgets for the first time

Once all components are created in the tab, the different Push-Buttons are linked to their respective callback functions using the *pyqtSlot* framework. PyqtSlots are functions that can be linked to an event for a component, for example when a button is pressed. The appropriate syntax for this action is shown in Fig. 4.6.



```

@pyqtSlot()
def loadProject():
    Project.loadProject(self, guiNormal)
    print('Project loaded')

# Connect buttons to each actions
save.pressed.connect(saveProject)
load.pressed.connect(loadProject)
simulate.pressed.connect(simulateProject)

```

Figure 4.6: PyQt Slots and link to button events

Once all these steps are done, the initialization is complete. The application is then event based. This is typical of GUIs, it means that unless the user triggers an event nothing happens and the application is on hold. Next, the report will detail the different actions the user can trigger and how they are executed within the program.

**4.2.2.2 ADDING AND DELETING WIDGETS DYNAMCALLY:** One of the basic events that the user can trigger is the addition or deletion of widgets on a specific tab, by using the Add or Delete buttons. This is useful if different grounding systems are to be added to the line for example. The framework to add and delete widgets is based on two generic functions, shown in Fig. 4.7.

These functions are extremely similar to the one described in Fig. 4.5 for the GUI initialization. The only difference is that the program needs to keep track of the total number of elements in the tab, in order to add them in the right place and to always keep all the elements of the first component in the tab.

**4.2.2.3 SAVING AND LOADING A PROJECT:** When saving a project, the GUI needs to read every text field in the tabs and store them in a structure. In this project, the main structure used to store all parameters is a Python dictionary. They are the few basic data types in Python with lists. Dictionaries contain elements that are sorted using string-type keys. For example, conductor data is stored under the *conductor* key, ground data is stored under the key *ground* etc. The main dictionary for this project is called



```

# General function to add widgets
def addWidgets(grid, names, rowOffset, numCol):
    count = grid.count()
    rowStart = int(count / numCol) # Number of columns and number of elements are
                                    # always even, so the result will always be an int.
                                    # Still needs to be converted because the result is
                                    # typed as a float by default
    rowEnd = rowStart + rowOffset
    positions = [(i,j) for i in range(rowStart, rowEnd) for j in range(numCol)]

    for position, name in zip(positions, names):
        if name == '':
            widget = QLineEdit()

        elif name == '/':
            widget = QLabel()

        else:
            widget = QLabel(name)

        grid.addWidget(widget, *position)

# General function to remove widgets
def removeWidgets(grid, initCount, rowOffset, numCol):
    count = grid.count()

    # No widgets left to delete
    if count == initCount:
        return 1

    rowEnd = int(count / numCol)
    rowStart = rowEnd - rowOffset
    positions = [(i,j) for i in range(rowStart, rowEnd) for j in range(numCol)]

    # Removing a widget's parents removes the widget in PyQt
    for position in positions:
        widget = grid.itemAtPosition(position[0], position[1]).widget()
        widget.setParent(None)

    return 0

```

Figure 4.7: Functions to dynamically add/delete widgets

*openetran* and contains a key for each type of component, along with a key for the project name and simulation data. Since there are several parameters for each type of components, the data stored under each key is represented as a list of numerical values. If there are several components of the same type in the design, there will be a list of lists under the corresponding key in the dictionary. A visual representation of the main structure is given in Fig. 4.8.

Again, since each tab is organized in the same way, a generic function was written to read each parameter and store them in the main dictionary. It is shown in Fig. 4.9.

This function goes through every widget in the tab but with a step of 2. Indeed, the



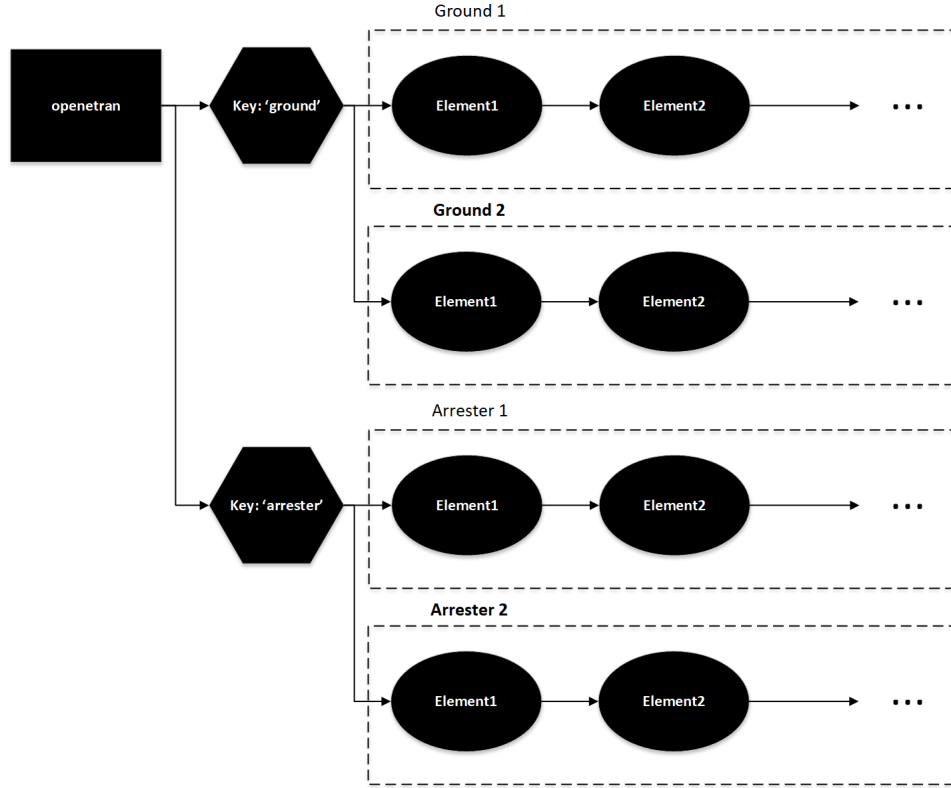


Figure 4.8: Visual representation of the main dictionary

widget configuration is always a label followed by a text field. Only the text field is relevant. The program keeps track of the current widget by updating an index of the current element being read. This index, called *count* in the function, is the product of the row number by the column number. Once the index is above the number of elements in the tab *countTotal*, then the reading is complete. Finally, when the argument *notEven* is 1, this means that the two last widgets of the tab are labels so they are discarded. This is used when the number of parameters is odd and the bottom right widget is not a textfield, like *Meter* or *LPM*. If it is 0, the number of parameters is even and the last widget on the tab is a text field, so it is read like all the others.

Once all parameters on every tab are read and stored in the main dictionary, the GUI writes them in a JSON file. JSON is a file format, specified in the RDC 7159 standard [17],



```

# General function to read widget on the grid layout
# Particular case if the number of text fields to read is not even
def readWidgets(openetran, grid, name, rowOffset, numCol, notEven):
    countTotal = grid.count()
    count = 0 # current count of the elements

    rowStart = 2
    rowEnd = rowStart + rowOffset

    while count < countTotal:
        # Positions of all text fields in each element of the tab
        positions = [(i,j) for i in range(rowStart, rowEnd) for j in range(1, numCol, 2)]
        listParam = list()

        # If the number of text fields is not even, the widget at the bottom right of the read
        # element will be a label, so it's ignored.
        if notEven == 1:
            for position in positions:
                count = (position[0] + 1) * (position[1] + 1)

                # Bottom right widget is a label, so we ignore it
                if position[0] == (rowEnd - 1) and position[1] == (numCol-1):
                    continue

            else:
                # Get the text and append it in the list
                widget = grid.itemAtPosition(position[0], position[1]).widget()
                listParam.append(widget.text())
        else:
            for position in positions:
                count = (position[0] + 1) * (position[1] + 1)

                widget = grid.itemAtPosition(position[0], position[1]).widget()
                listParam.append(widget.text())

        # Append the list of values (which makes for one whole component) in the key list
        openetran[name].append(listParam)
        rowStart = rowEnd + 1
        rowEnd = rowStart + rowOffset

```

Figure 4.9: Generic function to read text fields

used to store arrays in a human readable way. JSON is specified in the standard RDC 7159 [17]. There is a built-in JSON parser in Python, so writing files in this format is extremely simple and is achieved with the following instruction: *json.dump(openetran, f, indent=2)*, with *openetran* the array type element and *f* the file handle. To read a JSON file, a unique instruction is needed: *json.load(f)*. Since the structure is saved as a dictionary in JSON, it is also read as a dictionary so there it is directly usable after being read by the *load* instruction.



**4.2.2.4 CALCULATING THE COUNTERPOISE LOW CURRENT RESISTANCE** The ground low current resistance (R60) is used in OpenEtran for the admittance adjustment between the pole and ground impedances. The GUI adds the possibility to approximate the counterpoise R60 value to have more realistic results.

As shown in Fig. 3.2, the counterpoise is modeled as a ladder network. Since the current is considered low in this calculation, only the resistances and conductances defined in the counterpoise theoretical model are used (the capacitance is however still needed to calculate the conductance). The resistance of the counterpoise is then defined as the ladder network input impedance [18]:

$$R_{60} = r_i \cdot \frac{\sum_{j=0}^N [b[N][j] \cdot K^j]}{\sum_{j=0}^N [c[N][j] \cdot K^j]} \quad (4.1)$$

In (4.1),  $b$  and  $c$  are the matrices of coefficients of the DFF and DFFz triangles, which were introduced in [18]. The coefficients for these triangles are calculated with the following recurrent relationships:

$$b(i, j) = \begin{cases} 1, & \text{for } j = 0 \text{ or } j = i = 1 \\ 0, & \text{for } j > i \\ 2b(i-1, j) + b(i-1, j-1) - b(i-2, j), & \text{for } j \geq 1, i \geq 2, j \leq i \end{cases} \quad (4.2)$$

$$c(i, j) = \begin{cases} i, & \text{for } j = 0 \\ 0, & \text{for } j \geq i \\ 2c(i-1, j) + c(i-1, j-1) - c(i-2, j), & \text{for } j < i, j \neq 0 \end{cases} \quad (4.3)$$

**4.2.2.5 SIMULATING A PROJECT** When the user presses the simulation button, the first thing the program does is save the project in a JSON file. The operating system's selection window comes up so the user can select the appropriate file.

Then, the GUI parses the DAT input file needed to execute OpenEtran. The function that executes this operation is called in a loop for each key in the main dictionary, and it is shown in Fig. 4.10.

In this function, the program starts to check if all parameters are specified for each component. This means that each text field must contain something else than an empty



```

# General function to write the lists for each key in the input file
# The field at each key is expected to be a list of lists
def writeKey(f, openetran, key):
    err = 0

    for v in openetran[key]:
        # We first check if every field has been written in (counterpoise fields
        # in "ground" are optional so it's a special case for that key)
        if key == 'ground':
            for i in range(5):
                if v[i] == '':
                    err = 1
                    break

            # Last 2 fields are Pairs and Poles, which are mandatory
            i = len(v)
            if v[i-1] == '' or v[i-2] == '':
                err = 1
                break

        else:
            for i in range(len(v)):
                if v[i] == '':
                    err = 1
                    break

        # If there's an error (missing text field), we go to the next list
        if err == 1:
            # print('Error, missing field in ' + k)
            continue

        # We start by writing which field it is (same name as the key in the dict)
        f.write(key + ' ')

        # Write separate values, except for the last two.
        for i in range(len(v) - 2):
            f.write(v[i] + ' ')

        f.write('\n')

        # The last 2 strings are for pairs and poles
        i = len(v) - 2
        f.write('pairs ' + v[i] + '\n')
        f.write('poles ' + v[i+1] + '\n')

        f.write('\n')

```

Figure 4.10: Function to parse the OpenEtran input file

string. The only exception to that rule is for the ground component, because the counterpoise parameters are optional. This means only the first five components, plus "Pairs" and "Poles" are mandatory. *No check is made on the content of the actual text field.* Once the verification is complete, the program writes the name of the key first, which corresponds to the type of component, then the parameters are written using the standard *write* function for text files.



Finally, once the input file is parsed the GUI launches OpenEtran. The routine used to launch another program from Python is executed using the *subprocess* framework and the *run* function. This is shown in Fig. 4.11.

```
args = [execName, '-plot', 'csv', inputFileName]
completedProcess = subprocess.run(args, stderr=subprocess.PIPE, stdout=subprocess.PIPE,
                                  universal_newlines=True)
```

Figure 4.11: Function to call OpenEtran from Python

The only mandatory argument in this function is *args*, which is a list of strings regrouping the different arguments necessary when calling OpenEtran: executable name, type of simulation and parameters, input file. The *args* list shown in Fig. 4.11 is for a plot-mode simulation. The two following arguments mean that the function will capture in a buffer the output or error return message of OpenEtran. The last argument means that these outputs are sent as strings. If it is *False*, they are returned as uninterpreted binary outputs.

When simulating in critical current mode, the GUI reads the pole and wire sequences, saves them as lists, then calls OpenEtran in a loop for each pole. The arguments for this simulation mode are shown in Fig. 4.12.

```
# We call OpenEtran with wire1 = wire2 (see OpenEtran doc) in a loop
# for each selected pole
for i in poleSeq:
    if len(wireSeq) == 1:
        args = [execName, '-icrit', i, i, wireSeq[0], inputFileName]

    elif len(wireSeq) == 2:
        args = [execName, '-icrit', i, i, wireSeq[0], wireSeq[1], inputFileName]

    elif len(wireSeq) == 3:
        args = [execName, '-icrit', i, i, wireSeq[0], wireSeq[1], wireSeq[2], inputFileName]

    elif len(wireSeq) == 4:
        args = [execName, '-icrit', i, i, wireSeq[0], wireSeq[1], wireSeq[2], \
                wireSeq[3], inputFileName]

    elif len(wireSeq) == 5:
        args = [execName, '-icrit', i, i, wireSeq[0], wireSeq[1], wireSeq[2], \
                wireSeq[3], wireSeq[4], inputFileName]

    else:
        print('Invalid wire sequence')
        f.close()
        return
```

Figure 4.12: OpenEtran call arguments in critical current mode



## 4.3 VISUALIZATION TOOL

### 4.3.1 USER LEVEL DESCRIPTION

Fig. 4.13 shows the line visualization tool. It can only be used for five-wire transmission lines, with three phases and two shield wires. In this window, the arcs represent each wire's vulnerability zone, meaning that if lightning were to cross that arc it would hit the conductor. The thick green line below the conductors is the ground plane. The thin green lines crossing the arcs represents the striking distances to ground and objects, meaning that if a lightning bolt crosses these lines, it will hit the ground or an object. These striking distances are defined as:

$$r_c = 10I^{0.65} \quad (4.4)$$

$$r_g = \frac{\beta r_c}{\cos \alpha} \quad (4.5)$$

$r_c$  is the radius of the arcs,  $r_g$  is the striking distance to ground,  $\alpha$  is the ground slope, which can be between 0 and 45 degrees, and  $I$  the current going through the conductor. The term  $\beta$  is defined as:

$$\beta = \begin{cases} 0.37 + 0.17 \cdot \log_{10}(43 - h_{max}), & \text{for } h_{max} < 43 \\ 0.55, & \text{for } h_{max} > 43 \end{cases} \quad (4.6)$$

The green arcs outline the vulnerability zone for the shield wires and the red arcs for the phase wires. If a red arc is contained by two green arcs or is positioned below the thin green line, then the phase conductor is protected. If part of a wire is exposed, the exposure width is defined as the horizontal length of the exposed region of the arc. More details about the exposure width calculation are given in the next section.

In order to analyze the system properly, the user can enter the conductor's coordinates, either manually or by using the button *Update coordinates*. By pressing this button, the GUI copies the conductor coordinates that are written in the *Conductor* tab of the OpenEtran window. Then, it is possible to add objects to the design. The user also can specify the ground slope, total line length and flash density for the region the line will be installed.



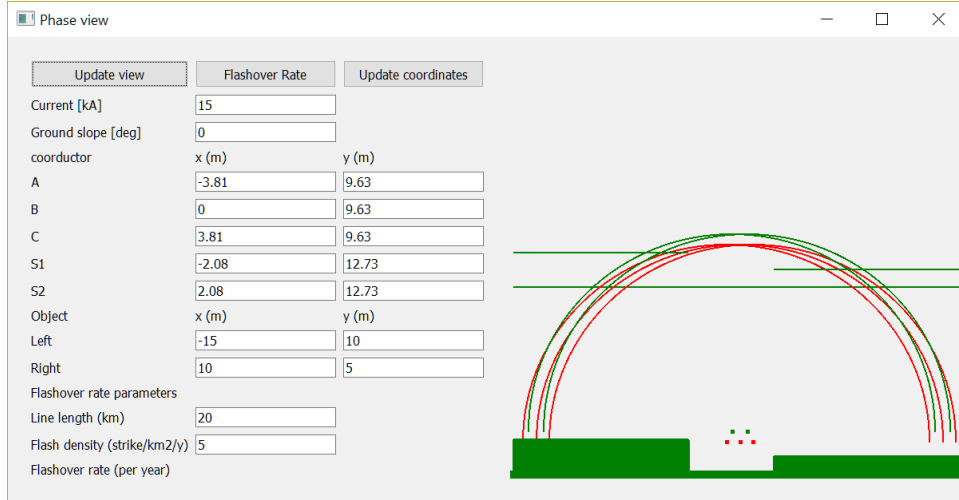


Figure 4.13: Phase visualization tool

Finally, the striking distances for the wires depend on the current going through them, so by changing the value in the textfield *Current*, the user changes the radii of the arcs. This setting is not used when calculating the flashover rate, it is solely used for display purposes to aid the user in the analysis.

The most important feature of the visualization tool is the calculation of the flashover rate. It is executed by pressing the *Flashover Rate* button, and the result is displayed in the bottom label of the window. The result is the yearly flashover rate on the whole length of the line. In order to work, the program needs the line geometry, along with the flash density and the critical current values for each conductor. These values of critical currents are calculated by OpenEtran in a *critical current iteration* simulation, and stored in a text file as an output (refer to section 4.2.2). When the flashover button is pressed, the GUI prompts the operating system's file section window, in order for the user to select the critical current file.



## 4.3.2 CODE ORGANIZATION AND INTERNAL CALCULATIONS

**4.3.2.1 INITIALIZATION OF THE APPLICATION:** The visualization tool is, just like the OpenEtran window, based on a Python class. This one is called *SysView*. When it is created it first needs to initialize. During the initialization of the application, the program places the different widgets in their position: the left half of the window is the parameter area, with the different labels and text fields for conductors coordinates to display the flashover rate. A grid layout is used to organize the widgets in that part of the window. The right half is a simple widget, which is used for the drawing area.

**4.3.2.2 PAINTING ELEMENTS ON THE DRAW VIEW:** The main function of the visualization tool is the callback function for a painting event. This function is called automatically to redraw the view each time something changes, for example if the user wants to change the window size or moves it on the screen. It can also be called manually when the user clicks on the *update view* button. This is useful when the value in the *Current* textfield is changed, because the window will not update automatically.

The different tasks that are executed in that function are:

- Translate the physical coordinates to screen coordinates
- Draw the ground plane
- Draw the vulnerability arcs
- Draw the objects and their strike line

The most time-consuming part is to adapt the coordinates to keep the scales true on screen. This is done with the function *calcCoordinates*. In this function, the GUI starts by reading all the physical coordinates from the text fields: ground slope, wires height and horizontal positions and the striking distances, are calculated using (4.4) and (4.5). In the rest of the description, the right half of the window, where the system is drawn, is called the *DrawView* widget.



After reading all the physical coordinates and striking distances, a horizontal and vertical scale is defined. The first value of this scale is:

$$vScale = H_w/50 \quad (4.7)$$

$$hScale = w/50 \quad (4.8)$$

With  $H_w$  the window's height and  $w$  the *DrawView*'s width.

Now that the scales are defined, the program translates the physical coordinates of the components into screen coordinates. The coordinates in a PyQt window are defined as pixel indexes, starting from the top left corner where the coordinates are (0,0). The phases' screen coordinates are:

$$x_{sc} = W_w - \frac{w}{2} + x \cdot hScale \quad (4.9)$$

$$y_{sc} = H_w - y \cdot vScale \quad (4.10)$$

With  $W_w$  the width of the whole window and (x,y) the wire's physical coordinates.

The arcs' radii and starting points' coordinates are calculated in the following way:

$$w_a = 2 \cdot r_c \cdot hScale \quad (4.11)$$

$$h_a = 2 \cdot r_g \cdot vScale \quad (4.12)$$

$$x_a = x_{sc} - \frac{w_a}{2} \quad (4.13)$$

$$y_a = y_{sc} - \frac{h_a}{2} \quad (4.14)$$

$w_a$  and  $h_a$  are the width and height of the arc respectively (they can be different since the window is not always a square),  $(x_a, y_a)$  are the coordinates of the arc's bottom left point. It is defined this way because the function used to paint the arcs takes this point's coordinates, along with the height and width, as parameters.

Finally, the objects are defined with only a horizontal position and a height. The first object starts from the right side of the *DrawView* and is in contact with the ground. The



two coordinates define the object length from the right side of the window and its height. These two metrics are calculated as:

$$x_{sc,obj} = W_w - \frac{w}{2} + x_{obj} \cdot hScale \quad (4.15)$$

$$y_{sc,obj} = H_w - y_{obj} \cdot vScale \quad (4.16)$$

All these coordinates are stored in a structure called *coord*. This structure is composed of several lists, one list for each type of coordinates:

1. x-component of all phase conductors
2. y-component of all phase conductors
3. x-component of all arcs' left starting point
4. y-component of all arcs' left starting point
5. arcs width
6. arcs height
7. striking distance to ground
8. x-component of all objects' top left/right point
9. y-component of all objects' top left/right point

Now that all the screen coordinates are calculated, it is necessary to verify whether all elements are inbound or not. In order to do that the program calculates one boolean value for each element, if it is true the element is out of bounds. The different conditions to determine if an element is out of bounds are:

$$\begin{aligned} c\_Out &= x < (W_w - w) \textbf{ or } x > W_w \\ &\textbf{ or } y < 0 \textbf{ or } y > H_w \end{aligned} \quad (4.17)$$

$$\begin{aligned} a\_Out &= c\_Out(origin) == True \\ &\textbf{ or } x_{origin} + w_a > W_w \\ &\textbf{ or } y_{origin} + h_a > H_w \end{aligned} \quad (4.18)$$

$$o\_Out = c\_Out(origin) == True \quad (4.19)$$



With  $c\_Out$ ,  $a\_Out$  and  $o\_Out$  the conditions for a conductor, an arc and an object respectively. The origin point for an arc is the bottom left end of the arc. For an object, it is either the top right (object1) or top left (object2) point.

If one or more of the three conditions are true, the program lowers the scale and recalculates the screen coordinates and out-of-bounds conditions. The new scales are defined as:

$$hScale' = \frac{9}{10} \cdot hScale \quad (4.20)$$

$$vScale' = \frac{9}{10} \cdot vScale \quad (4.21)$$

An overview of what the drawing area looks like after rescaling is shown in Fig. 4.14.

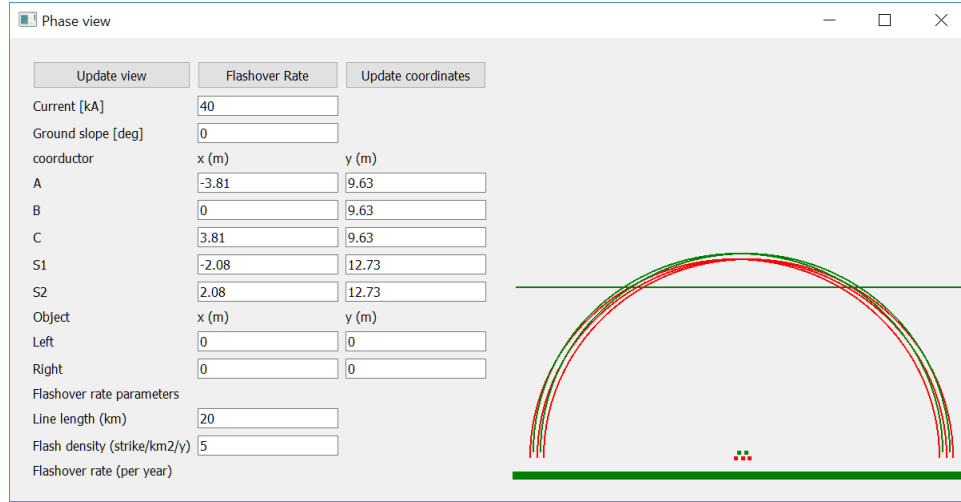


Figure 4.14: Phase visualization tool with rescaled drawing section

Once the scales have been calculated, the next step is to paint the ground line, phases, arcs and striking lines on the screen. In order to do that, several functions are called, one for each type of element to draw.

To draw on a window with PyQt, the user first needs to declare a *QPainter* object. No arguments are needed in the constructor, the *QPainter* object is the canvas on which the other objects will be drawn. Then a *QPen* object needs to be defined. The classic arguments



for this constructor are the colour, type of line (i.e dotted, solid etc.) and thickness. In the case of the ground, the pen is defined as  $QPen(Qt.darkGreen,10,Qt.SolidLine)$ . The program then draws a line, by using the function *drawLine* of the *QPainter* class. This function takes a *QLineF* object as an argument. This *QLineF* object defines a line between two points whose coordinates are floating point numbers. In the case of the ground line the two points to draw the line between have the coordinates (x1,y1) and (x2,y2), which are defined as:

$$x_1 = W_w - w \quad (4.22)$$

$$y_1 = H_w + w \cdot \frac{\tan \alpha}{2} \quad (4.23)$$

$$x_2 = w \quad (4.24)$$

$$y_2 = H_w - w \cdot \frac{\tan \alpha}{2} \quad (4.25)$$

The striking line to ground is drawn using the same functions and the coordinates of its two points are the same, with a vertical offset equal to  $rg$ , the striking distance to ground, times the vertical screen scale.

After drawing the ground line, the program starts drawing the phase conductors. In the visualization tool these conductors are represented as points, so drawing them is straightforward. The standard *drawPoint* function (see Fig. 4.15), which is a method of the *QPainter* class, is used. This function takes a *QPointF* object as an argument. This *QPointF* object only takes the coordinates of the point as floating point numbers. The screen scale coordinates of the phase wires are then simply copied into the function to draw them on the screen.

Once the phases are drawn, the program starts to define the arcs. Since all the necessary coordinates have been calculated already, the only function that needs to be called is *drawArc* (see Fig. 4.16), which is also a method of the *QPainter* class. This function takes as arguments the (x,y) coordinates of the bottom left point of the arc, its width, its height, the start angle (in this case 0) and the span angle in  $1/16^{th}$  of degree. For a total semi-circle the total span angle is then  $180 \cdot 16 = 2880$ .

Finally, the last things to draw are the objects (see Fig. 4.17). They are drawn using *QPolygonF* objects, which behave as a list of *QPointF* objects. To draw a parallelogram like



```

def drawPhases(self, qp, coord):
    # Phase wires, red pen
    pen = QPen(Qt.red, 5, Qt.SolidLine)
    qp.setPen(pen)

    for k in range(len(coord[0])-2):
        qp.drawPoint(coord[0][k], coord[1][k])

    # Shielding wires in green
    pen = QPen(Qt.darkGreen, 5, Qt.SolidLine)
    qp.setPen(pen)

    for k in range(len(coord[0])-2, len(coord[0])):
        qp.drawPoint(coord[0][k], coord[1][k])

```

Figure 4.15: *drawPhases* function

```

def drawArcs(self, qp, coord):
    for k in range(len(coord[0])):
        if k < len(coord[0])-2:
            # Phase wires in red
            qp.setPen(QPen(Qt.red, 2, Qt.SolidLine))
        else:
            # Shielding wires in green
            qp.setPen(QPen(Qt.darkGreen, 2, Qt.SolidLine))

    qp.drawArc(coord[2][k], coord[3][k], coord[4][k], coord[5][k], 0, 180*16)

```

Figure 4.16: *drawArcs* function

the objects, only four points are needed. As stated before the objects on the visualisation tool are defined only with one point and span all the way to a side of the drawing area: top right point and left side for Object1, top left point and right side for Object2.

Let's define  $(x_{tr}, y_{tr})$  the original coordinates of the top-right point of Object1. Since some adjustments need to be made to compensate for the ground slope  $\alpha$ , the final coordinates for each point of Object1 are:

$$p_{11} = \left( W_w - w, H_w + \frac{w \cdot \tan \alpha}{2} \right) \quad (4.26)$$

$$p_{12} = \left( W_w - w, y_{tr} + \frac{w \cdot \tan \alpha}{2} \right) \quad (4.27)$$

$$p_{13} = \left( x_{tr}, y_{tr} + \tan(\alpha) \cdot \left( W_w - \frac{w}{2} - x_{tr} \right) \right) \quad (4.28)$$

$$p_{14} = \left( x_{tr}, H_w + \tan(\alpha) \cdot \left( W_w - \frac{w}{2} - x_{tr} \right) \right) \quad (4.29)$$



With each point being respectively the bottom left, top left, top right and bottom right points. Similarly, with  $(x_{tl}, y_{tl})$  the original coordinates of the top-left point, the final coordinates for each point of Object2 are defined as:

$$p_{21} = \left( x_{tl}, H_w + \tan(\alpha) \cdot \left( W_w - \frac{w}{2} - x_{tl} \right) \right) \quad (4.30)$$

$$p_{22} = \left( x_{tl}, y_{tl} + \tan(\alpha) \cdot \left( W_w - \frac{w}{2} - x_{tl} \right) \right) \quad (4.31)$$

$$p_{23} = \left( W_w, y_{tl} - \frac{w \cdot \tan \alpha}{2} \right) \quad (4.32)$$

$$p_{24} = \left( W_w, H_w - \frac{w \cdot \tan \alpha}{2} \right) \quad (4.33)$$

With each point being respectively the bottom left, top left, top right and bottom right. After appending the points to the polygon structure, the last operation is to fill the geometry. In order to do that, it is necessary to declare a *QPainterPath* object, associate the polygon to the path by using the function *addPolygon* then fill the path with the function *fillPath*. This last function is a member of the *QPainter* class, and takes as arguments the *QPainterPath* object and a color, represented as a *QColor* object.

This finishes the description of all the operations needed to draw the system on the window.

**4.3.2.3 CALCULATING THE FLASHOVER RATE** The other main functionality of the visualization tool is the calculation of the yearly average flashover rate on the total length of the line. This function is by far the heaviest of the program in terms of calculations, and is separated in two main steps:

- Calculating each phase wire's exposure width.
- Calculating the total flashover rate at each pole.

As a reminder, during a critical-current simulation in OpenEtran the results are written in a text file. The user defines the wire and pole sequence for the analysis, so potentially not all phases and not all poles are concerned by the analysis. The first step in the flashover rate function is then to parse this output text file, which is shown in Fig. 4.18. The program starts by creating a list of five other lists. Each list contains the current values at different poles for a specific phase.



```

# If the object has a height of 0 we don't need to draw it
if realCoord[1][0] > 0:
    qp.setPen(QPen(Qt.darkGreen, 5, Qt.SolidLine))

    # Left object (starts from left side until the xCoordinate)
    poly = QPolygonF()

    # Bottom left
    point1 = QPointF(self.width() - self.drawView.width(),
                     self.drawView.height() + self.drawView.width()*tan/2)
    poly.append(point1)

    # Top left
    point2 = QPointF(self.width() - self.drawView.width(),
                     coord[8][0] + self.drawView.width()*tan/2)
    poly.append(point2)

    # Top right
    point3 = QPointF(coord[7][0], coord[8][0] + tan*(self.width() -
        self.drawView.width()/2 - coord[7][0]))
    poly.append(point3)

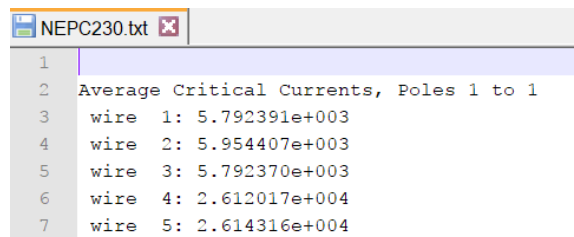
    # Bottom right
    point4 = QPointF(coord[7][0],
                     self.drawView.height() + tan*(self.width() -
        self.drawView.width()/2 - coord[7][0]))
    poly.append(point4)

    path = QPainterPath()
    path.addPolygon(poly)

    qp.drawPolygon(poly)
    qp.fillPath(path, QColor(Qt.darkGreen))

```

Figure 4.17: First half of the *drawObjects* function, for *Object1*



Index	Wire	Critical Current
1	Average	Critical Currents, Poles 1 to 1
2	wire 1	5.792391e+003
3	wire 2	5.954407e+003
4	wire 3	5.792370e+003
5	wire 4	2.612017e+004
6	wire 5	2.614316e+004

Figure 4.18: Critical current file

At each line of the text file, the index of the concerned wire is given, followed by the value of the critical current. The program reads this index first to determine in which list



to store the current, then the actual value is read and converted in kA. Finally, after all currents in the file have been read, the program adds -1 values in unused lists to indicate that the corresponding phases are not concerned by the flashover analysis.

```

fileName = name[0]
with open(fileName, 'r') as f:
    # Reads the whole critical currents file
    output = f.readlines()

    # If the word 'wire' is in a line, there's a critical current value to store
    for line in output:
        if 'wire' in line:
            end = len(line)
            pos = int(line[7]) - 1 # Wire number in the string, needed to know which
                                # position it should take on the list.

            try:
                current = float(line[end-14:end-1]) / 1000 # Actual critical current value [kA]
            except ValueError:
                print('Critical current reading error')
                continue

            icritList[pos].append(current)
        else:
            continue

```

Figure 4.19: Critical current parser

Once the parsing operations are done, the program starts the actual flashover rate calculation for each pole. In order to do that, the program calculates in a loop the exposure widths of each concerned phase for several values of current going through them, from 2.5kA to 300kA with a step of 0.5kA. If the value of current is inferior to the critical current of a specific phase, the probability of causing a back-flashover is 0 so this phase's exposure width is set to 0. If the critical current of a phase is -1, it means it is not considered for the analysis so the exposure width is also set to 0. Finally if the current value is superior to the critical current, the exposure width for a specific arc is calculated as follows:

**Step 1:** The program enumerates and stores all the coordinates of the intersections between the considered arc and the arcs from the other four phases, or between the considered arc and a striking line from the ground or an object. This is done by solving a second degree equation on the arc's equation:

$$(x - x_1)^2 + (y - y_1)^2 = r_c^2 \quad (4.34)$$

$$(x - x_2)^2 + (y - y_2)^2 = r_c^2 \quad (4.35)$$



In the previous system,  $(x,y)$  are the intersection's coordinates,  $(x_1,y_1)$  and  $(x_2,y_2)$  are the coordinates of each arc's center and  $r_c$  is the arcs' radii. The solutions for this system are, if  $x_1 \neq x_2$  and  $y_1 \neq y_2$ :

$$x = ay + b \quad (4.36)$$

$$y = \frac{-B_1 + \sqrt{\Delta_1}}{2A_1}, \text{ if } \Delta \geq 0 \quad (4.37)$$

With:

$$a = \frac{-2(y_2 - y_1)}{2(x_2 - x_1)} \quad (4.38)$$

$$b = \frac{x_2^2 + y_2^2 - x_1^2 - y_1^2}{2(x_2 - x_1)} \quad (4.39)$$

$$A_1 = a^2 + 1 \quad (4.40)$$

$$B_1 = 2ab - 2ax_1 - 2y_1 \quad (4.41)$$

$$C_1 = b^2 - 2bx_1 + x_1^2 + y_1^2 - r_c^2 \quad (4.42)$$

$$\Delta_1 = B_1^2 - 4A_1C_1 \quad (4.43)$$

Note that, even if there are two solutions for  $y$  theoretically, since the only section of interest in the arc is the upper half, the stored solution is the highest. Similarly, if  $x_1 = x_2$ , the solutions are:

$$x = \frac{-B_2 + \sqrt{\Delta_2}}{2A_2} \quad (4.44)$$

$$y = \frac{y_2^2 - y_1^2}{2(y_2 - y_1)} \quad (4.45)$$

With:

$$A_2 = 1 \quad (4.46)$$

$$B_2 = -2x_1 \quad (4.47)$$

$$C_2 = x_1^2 + y^2 - 2yy_1 + y_1^2 - r_c^2 \quad (4.48)$$

$$(4.49)$$



When  $y_1 = y_2$  the solutions are analogous to the previous case:

$$x = \frac{x_2^2 - x_1^2}{2(x_2 - x_1)} \quad (4.50)$$

$$y = \frac{-B_3 + \sqrt{\Delta_3}}{2A_3} \quad (4.51)$$

With:

$$A_3 = 1 \quad (4.52)$$

$$B_3 = -2y_1 \quad (4.53)$$

$$C_3 = x^2 - 2xx_1 + x_1^2 + y_1^2 - r_c^2 \quad (4.54)$$

These solutions are considered valid only if their y-component is above both of the phases of the concerned arcs. If they are not, it means the intersection happens in the lower half of one arc so it is not relevant.

Finally, if there is an intersection with a striking line, the equation to solve is:

$$(x - x_1)^2 + (r_g + y_o + \tan \alpha \cdot x_1 - y_1)^2 = r_c^2 \quad (4.55)$$

With  $(x_1, y_1)$  the arc's center's coordinates,  $y_o$  the object's top right/left point's y-component coordinate. The solutions for this system are:

$$x = \frac{-B_o \pm \sqrt{\Delta_o}}{2A_o} \quad (4.56)$$

$$y = r_g + y_o + \tan \alpha \cdot x_1 \quad (4.57)$$

With:

$$A_o = 1 \quad (4.58)$$

$$B_o = -2x_1 \quad (4.59)$$

$$C_o = x_1^2 + (r_g + y_o + \tan \alpha \cdot x_1 - y_1)^2 - r_c^2 \quad (4.60)$$

In the case of an intersection with a striking distance line, unlike for an arc, there may be two intersections in the upper-half of the circle.



**Step 2:** After having all the intersections for a specific arc, the program isolates the "exposed" intersections. If an intersection is inside another arc or below a ground/object striking line, then it is protected and discarded. This operation is done by the *isContained* function, which returns a boolean value. For an intersection to be contained within an arc, it needs to respect the following inequation:

$$(x - x_i)^2 + (y - y_i)^2 < r_c^2 \quad (4.61)$$

For an intersection to be contained by a ground or object1 striking line, it needs to respect the following conditions:

$$\begin{cases} y < r_g \\ y < r_g + y_o + \tan \alpha \cdot x \text{ and } x < x_o \end{cases} \quad (4.62)$$

In the previous cases,  $(x_o, y_o)$  are the object's top right point's coordinates. To be contained by the Object2 striking line, the conditions are the same, except that  $x > x_o$ .

**Step 3:** The last step when calculating the exposure width is to check whether the portion of the arc between to exposed intersections is also vulnerable to lightning. In order to verify this, the program calculates the coordinates of the point at mid-distance between the two intersections and calls the *isContained* function a second time. If this point is also vulnerable then the horizontal distance between the two exposed intersections is stored as the exposure width. These values for each phase at a specific pole are stored in a list called *expo*.

Now that the exposure widths of each phase at a specific pole are calculated, the program starts the operations to get the yearly flashover rate per pole. This is predicted by:

$$f_y = \frac{L}{N} \cdot D_{flash} \cdot p \cdot \sum_{k=0}^K \frac{w_{expo}(k)}{1000} \quad (4.63)$$

In (4.63),  $f_y$  is the flashover rate per year at a specific pole,  $w_{expo}(k)$  is an exposure width at a specific pole in meters,  $K$  is the size of the exposure width list,  $L$  is the length of the line in km,  $N$  is the number of poles in the line,  $D_{flash}$  is the lightning flash density in



$flash/km^2$  (this can be obtained from geographical data) and  $p$  is the probability that the lightning's first stroke current will be greater or equal to the current value in the loop. This probability is set in [19] and is defined as:

$$p(I \geq i_0) = \frac{1}{1 + \left(\frac{i_0}{31}\right)^{2.6}} \quad (4.64)$$

As a reminder, the program does the previous calculations in a loop for several values of current between 2.5kA and 300kA, so the final flashover rate value for each pole is the sum of all results from (4.63) for each current value. Then finally, the value displayed in the visualization tool is the arithmetical average of the flashover rates at each pole:

$$F_y = \frac{\sum_{n=0}^N f_y(n)}{N} \quad (4.65)$$



## 5.0 CASE STUDIES

### 5.1 NEPC230 - NEW ENGLAND 230KV STEEL

OpenEtran has been executed on two different variations of an IEEE Flash test case NEPC230 [2]. This system represents a 230kV line with a 122m span, 11 towers modeled and 5 conductors: 3 phases and 2 shield wires. In this test case, a -80kA lightning surge strikes tower 6 on one of the shielding wires. The critical flashover voltage (CFO) is estimated at 1350 kV on each phase. Fig. 5.1 shows the results of the transient simulation of the ground current on the struck tower, for two different grounding methods.

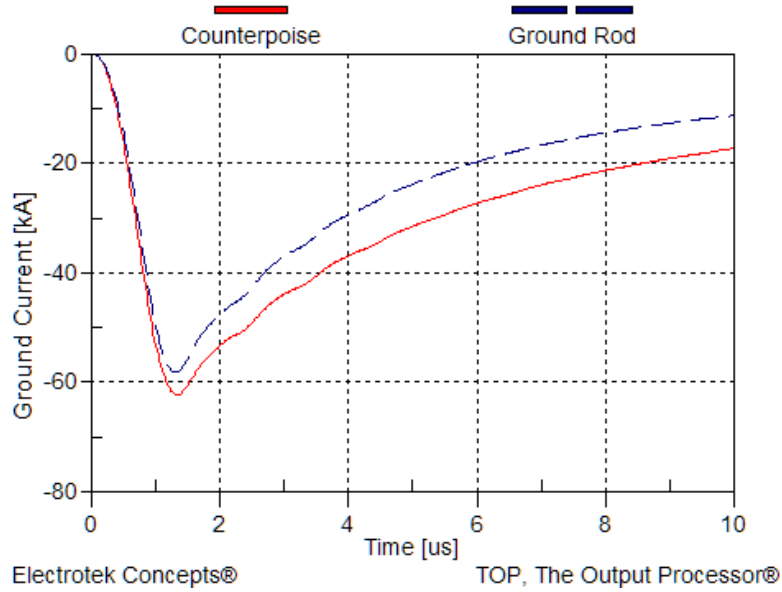


Figure 5.1: Ground currents for -80kA stroke to the NEPC230 tower with counterpoise and ground rod



On the first variation, a 122m long counterpoise conductor with a 1cm radius, divided into 20 segments and buried at 1m beneath ground is used. On the second variation, the tower only has a 20m vertical ground rod but no counterpoise. With  $\rho = 1000\Omega m$ , the 60-Hz low-current ground resistance is  $16.86\Omega$  for the counterpoise and  $63.56\Omega$  for the ground rod. Fig. 5.2 shows that the peak insulation voltage (1457.5kV) with ground rod exceeds the CFO, but with counterpoise the peak (1071.8kV) is less than the CFO. Here, the counterpoise appears to be effective, but higher stroke currents, line arresters and insulator upgrades can also be simulated.

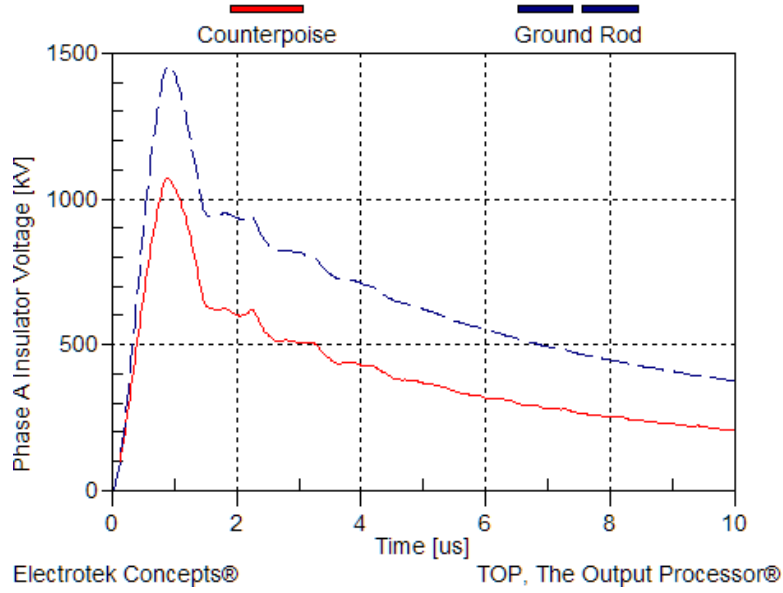


Figure 5.2: Phase A insulator voltages for -80 kA stroke to the NEPC230 tower with counterpoise and ground rod; CFO is 1350 kV.

Note that the voltage peaks at around  $1\mu s$  in both cases, due to the effects of  $L_T$ . The peak counterpoise current in Fig. 5.1 is higher than the peak ground rod current, because of differences in the resistance. From the  $V_T$  waveforms, not plotted here, the apparent ground resistance at  $10\mu s$  is  $15.99\Omega$  for the counterpoise and  $44.38\Omega$  for the ground rod. These values include the effect of  $L_T$ , but that is negligible at  $10\mu s$ . The ground rod has a greater reduction in resistance than the counterpoise, because of heavier soil ionization predicted in (3.1) and (3.2). Even so, the counterpoise performs better.



## 6.0 CONCLUSION

A theoretical model for the counterpoise conductor, inspired from [12], was added to the OpenEtran transient simulation engine in order to better benchmark lightning performance in modern power systems. Results show a coherent behavior of the counterpoise conductor.

A new GUI has been added, in order to increase user efficiency while using OpenEtran. This GUI regroups a new tab window, similar in organization to the previous Excel spreadsheet interface, and a line visualization tool has been introduced to allow the user to better design line shielding to reduce flashover rates. The software will be publicly available for the engineering community under the open source GNU General Public License.

Future work ideas on this project would include adding more counterpoise system architectures (crow foot, parallel counterpoise etc.), and address the frequency-dependent aspect of the soil resistivity, which in turn affects the current leakage rate into the ground and hence modifies the total grounding impedance [11].



## BIBLIOGRAPHY

- [1] T. E. McDermott, T. A. Short, F. G. Velez, and J. S. McDaniel. Open source lightning protection and electromagnetic transients software. In *2013 IEEE Power Energy Society General Meeting*, pages 1–5, July 2013.
- [2] IEEE Flash [Online]. Available: <http://www.sourceforge.net/projects/ieeeflash>.
- [3] H. W. Dommel. Nonlinear and time-varying elements in digital simulation of electromagnetic transients. *IEEE Transactions on Power Apparatus and Systems*, PAS-90(6):2561–2567, Nov 1971.
- [4] H. W. Dommel. Digital computer solution of electromagnetic transients in single-and multiphase networks. *IEEE Transactions on Power Apparatus and Systems*, PAS-88(4):388–399, April 1969.
- [5] IEEE Guide for Improving the Lightning Performance of Transmission Lines. *IEEE Std 1243-1997*, pages 1–44, Dec 1997.
- [6] P. Sebire, D. J. Woodhouse, and W. J. V. Tocher. Effective management of earth potential rise through alternative installation strategies for counterpoise conductors. In *2016 Down to Earth Conference (DTEC)*, pages 1–7, Sept 2016.
- [7] C. L. G. Fortescue. Counterpoises for transmission lines. *Electrical Engineering*, 53(4):598–600, April 1934.
- [8] L. V. Bewley. Theory and tests of the counterpoise. *Electrical Engineering*, 53(8):1163–1172, Aug 1934.
- [9] W. A. Chisholm and W. Janischewskyj. Lightning surge response of ground electrodes. *IEEE Transactions on Power Delivery*, 4(2):1329–1337, Apr 1989.
- [10] A. K. Mishra, N. Nagaoka, and A. Ametani. Frequency-dependent distributed-parameter modelling of counterpoise by time-domain fitting. *IEE Proceedings - Generation, Transmission and Distribution*, 153(4):485–492, July 2006.
- [11] Bok-Hee Lee Jong-Hyuk Choi and Seung-Kwon Paek. Frequency-dependent grounding impedance of the counterpoise based on the dispersed currents. *Journal of Electrical Engineering and Technology*, 7(4):589–595, 2012.



- [12] Jinliang He, Yanqing Gao, Rong Zeng, Jun Zou, Xidong Liang, Bo Zhang, Jaebok Lee, and S. Chang. Effective length of counterpoise wire under lightning current. *IEEE Transactions on Power Delivery*, 20(2):1585–1591, April 2005.
- [13] OpenEtran user manual. [Online]. Available: <http://sourceforge.net/projects/epri-openetran/>.
- [14] M. Kagan and X. Wang. Infinite circuits are easy, how about long ones ?. [Online]. Available: <https://arxiv.org/abs/1507.08221>, 2015.
- [15] Akihiro Ametani, Daigo Soyama, Yu Ishibashi, Naoto Nagaoka, and Shigemitsu Okabe. Modeling of a buried conductor for an electromagnetic transient simulation. *IEEE Transactions on Electrical and Electronic Engineering*, 1(1):45–55, 2006.
- [16] GSL - GNU Scientific library. [Online]. Available: <https://www.gnu.org/software/gsl/>.
- [17] RFC7159 - The JavaScript Object Notation (JSON) Data Interchange Format. [Online]. Available: <https://tools.ietf.org/html/rfc7159#section-1.2>.
- [18] M. Faccio, G. Ferri, and A. D’Amico. A new fast method for ladder networks characterization. *IEEE Transactions on Circuits and Systems*, 38(11):1377–1382, Nov 1991.
- [19] IEEE Guide for Improving the Lightning Performance of Electric Power Overhead Distribution Lines. *IEEE Std 1410-2010 (Revision of IEEE Std 1410-2004)*, pages 1–73, Jan 2011.