GridCal

Santiago Peñate Vera

GRIDCAL

Research oriented power systems software.
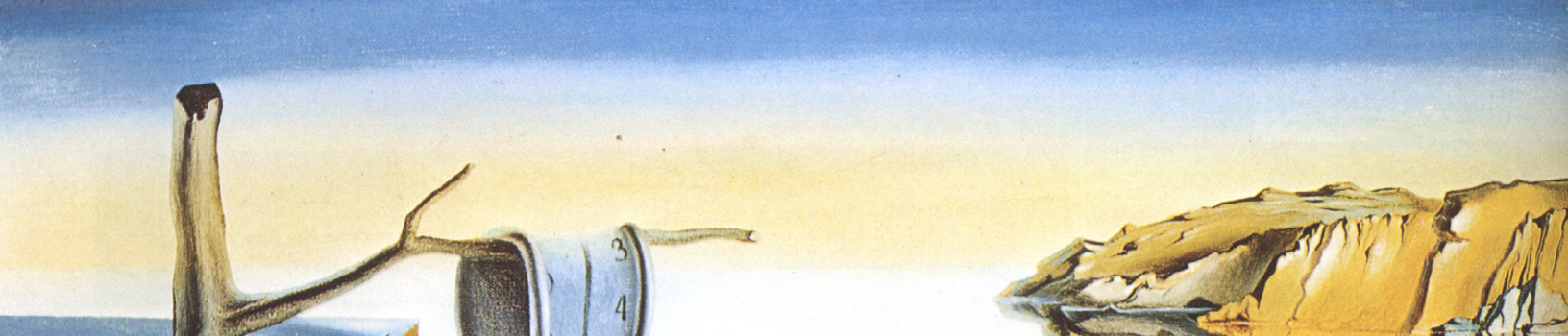
Started writing this document in Madrid the 9th of October of 2016

Table of content

1. Introduction

GridCal is a research oriented power systems software.

Research oriented? How? Well, it is a fruit of research. It is designed to be modular. As a researcher I found that the available software (not even talking about commercial options) are hard to expand or adapt to achieve complex simulations. GridCal is designed to allow you to build and reuse modules, which eventually will boost your productivity and the possibilities that are at hand.

I have made other projects (even another open source one: fPotencia in C++). I believe that this project encapsulates my half life of programming experience and the curiosity I have developed for power systems.

So, I do really hope you enjoy it, and if you have comments, suggestions or just want to collaborate, do not hesitate to contact.

Cheers,
Santiago

2. Structure

GridCal uses an object oriented approach for all the data and simulation management. However the object orientation is very inefficient when used in numerical computation, that is why there are `compile()` functions that extract the information out of the objects and turn this information into vectors, matrices and DataFrames in order to have efficient numerical computations. I have found this approach to be the best compromise between efficientcy and code escalability and maintainability after having been involved in quite some number-crunching software developments.

The whole idea can be summarized as:

```
Object oriented structures -> intermediate objects holding arrays -> Numerical modules
```

2.1 Circuit and MultiCircuit

The concept of circuit should be easy enough to understand. It represents a set of nodes (buses) and branches (lines, transformers or other impedances)

The `MultiCircuit` class is the main object in GridCal. It represents a circuit that may contain islands. It is important to understand that a circuit split in two or more islands cannot be simulated as is, because the admittance matrix would be singular. The solution to this is to split the circuit in island-circuits. Therefore `MultiCircuit` identifies the islands and creates individual `Circuit` objects for each of them.

As I said before GridCal uses an object oriented approach for the data management. This allows to group the data in a smart way. In GridCal I have decided to have only two types of object directly declared in a `Circuit` or `MultiCircuit` object. These are the `Bus` and the `Branch`. The branches connect the buses and the buses contain all the other possible devices like loads, generators, batteries, etc. This simplifies enormously the management of element when adding, associating and deleting.

2.2 The Bus object

The Bus object is the container of all the possible devices that can be attached to a bus bar or substation. Such objects can be loads, voltage controlled generators, static generators, batteries, shunt elements, etc.

2.3 The branch object

2.4 Class diagram

Here the API class diagram is sketched. All the classes are included but only the most fundamental properties and functions od each class are included to keep the diagram simple.
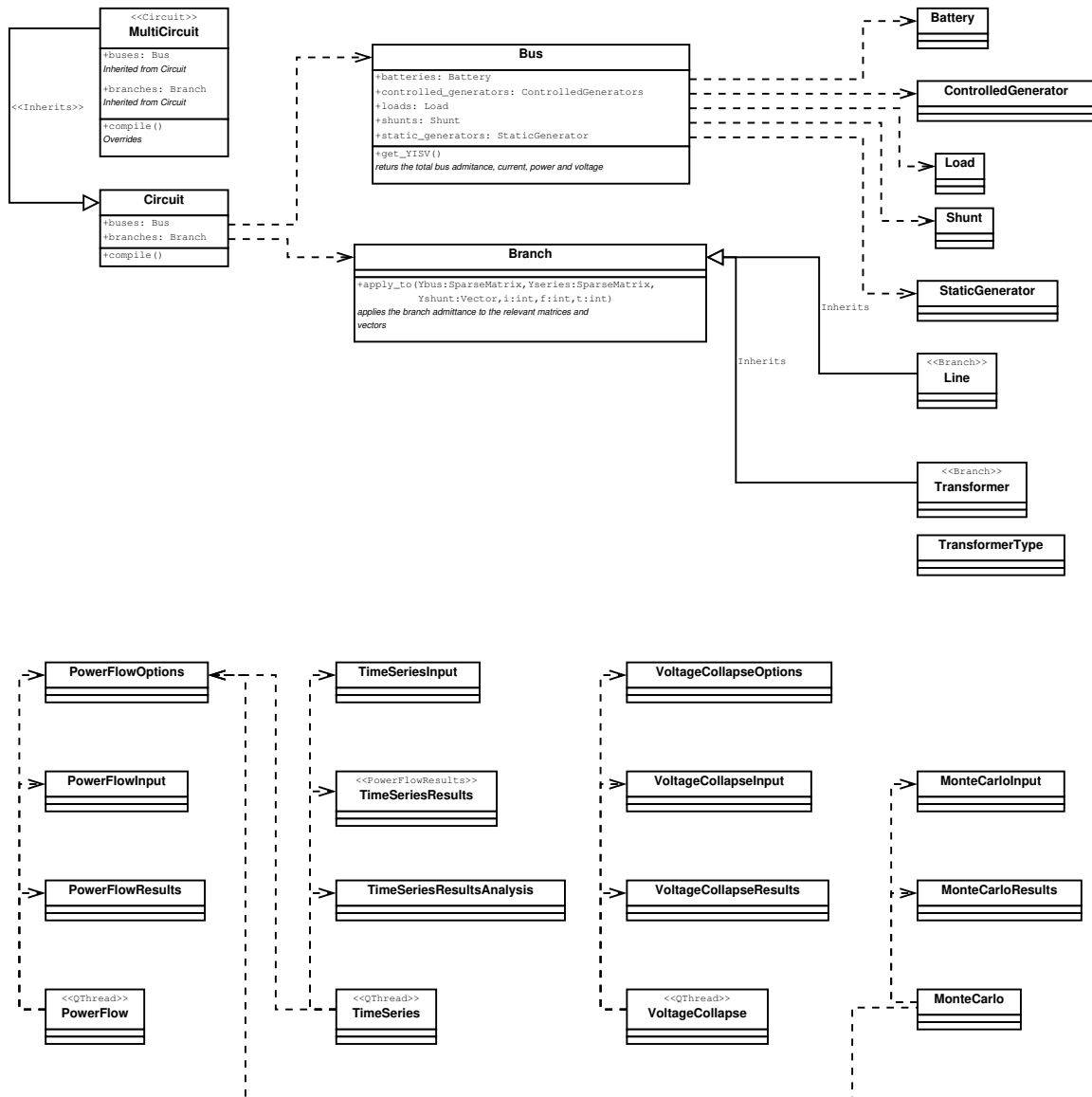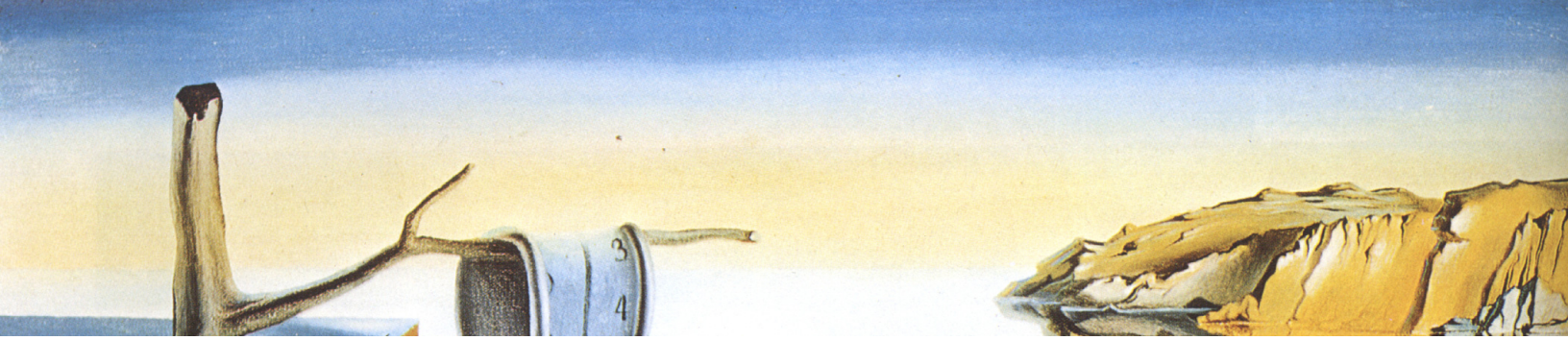
Figure 2.1: Simplified class diagram of GridCal's API

3. Models

3.1 Building the admittance matrices

This operation occurs in the `Compile()` function of the `Circuit` object. This function compiles many other magnitudes and among them `Ybus`, `Yseries` and `Yshunt`.

- `Ybus`: Complete admittance matrix.
  It is a sparse matrix of size $n \times n$
- `Yseries`: Admittance matrix of the series elements. It contains no value comming from shunt elements or the shunt parts of the branch model.
  It is a sparse matrix of size $n \times n$
- `Yshunt`: Admittance vector of the shunt elements and the shunt parts of the branch model.
  It is a vector of size n
- `Yf`: Admittance matrix of the banches with their *from* bus.
  It is a sparse matrix of size $m \times n$
- `Yt`: Admittance matrix of the banches with their *to* bus.
  It is a sparse matrix of size $m \times n$

Where n is the number of buses and m is the number of branches.

The relation between the admittance matrix and the series and shunt admittance matrices is the following:

$$Y_{bus} = Y_{series} + Y_{shunt} \tag{3.1}$$

The algorithmic logic to build the matrices in pseudo code is the following:

```
n = number of buses in the circuit
m = number of branches in the circuit
For i=0 to n:
    bus_shunt_admittance, bus_current, bus_power, bus_voltage = buses[i].get_YISV()
```

```
    Yshunt[i] = bus_shunt_admittance
end

For i=0 to m:
    f = get_bus_inde(branches[i].bus_from)
    t = get_bus_inde(branches[i].bus_to)

    // the matrices are modified by the branch object itself
    branches[i].apply_to(Ybus,Yseries,Yshunt,Yf,Yt,i,f,t)
end
```

3.2 The universal branch model

The following describes the model that is applied to each type of admittance matrix in the `apply_to()`
function inside the `Branch` object seen before.

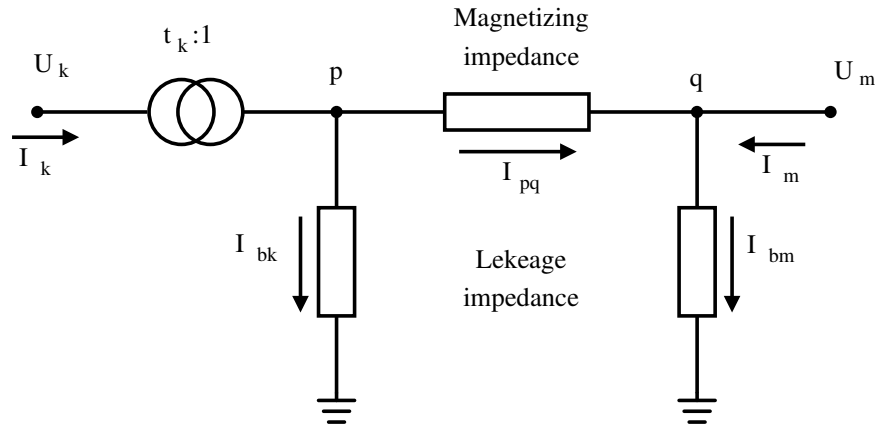The model implemented to describe the behavior of the transformers and lines is the π (pi)
model.

Figure 3.1: π model of a branch

To define the π branch model we need to specify the following magnitudes:
- z_{series}: Magnetizing impedance or simply series impedance. It is given in p.u.
- y_{shunt}: Leakage impedance or simply shunt impedance. It is given in p.u.
- tap_module: Module of the tap changer. It is a magnitude around 1.
- tap_angle: Angle of the tap changer. Angle in radians.

In order to apply the effect of a branch to the admittance matrices, first we compute the complex
tap value.

$$tap = tap\_module \cdot e^{-j \cdot tap\_angle}$$

Then we compose the equivalent series and shunt admittance values of the branch. Both values
are complex.

$$Y_s = \frac{1}{z_{series}}$$

$$Y_{sh} = \frac{y_{shunt}}{2}$$

- z_{series}: Series impedance of the branch composed by the line resistance and its inductance. $z_{series} = r + jl$
- y_{shunt}: Shunt admittance of the line composed by the conductance and the susceptance. $y_{shunt} = c + jb$

The general branch model is represented by a 2×2 matrix. $Y_{branch} = \begin{pmatrix} Y_{ff} & Y_{ft} \\ Y_{tf} & Y_{tt} \end{pmatrix}$

In this matrix, the elements are the following:

$$Y_{ff} = \frac{Y_s + Y_{sh}}{tap \cdot conj(tap)}$$

$$Y_{ft} = -Y_s/conj(tap)$$

$$Y_{tf} = -Y_s/tap$$

$$Y_{tt} = Y_s + Y_{sh}$$

Ybus

The branch admittance values are applied to the complete admittance matrix as follows:

$$Y_{bus\,f,f} = Y_{bus\,f,f} + Y_{ff}$$

$$Y_{bus\,f,t} = Y_{bus\,f,t} + Y_{ft}$$

$$Y_{bus\,t,f} = Y_{bus\,t,f} + Y_{tf}$$

$$Y_{bus\,t,t} = Y_{bus\,t,t} + Y_{tt}$$

These formulas assume that there might be something already in Y_{bus}, therefore the right way to modify these values is to add the own branch values.

3.2.1 Yshunt

$$Y_{shunt\,f} = Y_{shunt\,f} + Y_{sh}$$

$$Y_{shunt\,t} = Y_{shunt\,t} + \frac{Y_{sh}}{tap \cdot conj(tap)}$$

3.2.2 Yseries

$$Y_{series\,f,f} = Y_{series\,f,f} + \frac{Y_s}{tap \cdot conj(tap)}$$

$$Y_{series\,f,t} = Y_{series\,f,t} + Y_{ft}$$

$$Y_{series\,t,f} = Y_{series\,t,f} + Y_{tf}$$

$$Y_{series\,t,t} = Y_{series\,t,t} + Y_s$$

3.2.3 Yf and Yt

$$Y_{f\,i,f} = Y_{f\,i,f} + Y_{ff}$$
$$Y_{f\,i,t} = Y_{f\,i,t} + Y_{ft}$$

$$Y_{t\,i,f} = Y_{t\,i,f} + Y_{tf}$$
$$Y_{t\,i,t} = Y_{t\,i,t} + Y_{tt}$$

3.3 The transformer definition from the short circuit test values

The transformers are modeled as π branches too. In order to get the series impedance and shunt admittance of the transformer to match the branch model, it is advised to transform the specification sheet values of the device into the desired values. The values to take from the specs sheet are:

- S_n: Nominal power in MVA.
- U_{hv}: Voltage at the high-voltage side in kV.
- U_{lv}: Voltage at the low-voltage side in kV.
- U_{sc}: Short circuit voltage in %.
- P_{cu}: Copper losses in kW.
- I_0: No load current in %.
- GX_{hv1}: Reactance contribution to the HV side. Value from 0 to 1.
- GR_{hv1}: Resistance contribution to the HV side Value from 0 to 1.

Then, the series and shunt impedances are computed as follows:
Nominal impedance HV (Ohm): $Zn_{hv} = U_{hv}^2/S_n$

Nominal impedance LV (Ohm): $Zn_{lv} = U_{lv}^2/S_n$

Short circuit impedance (p.u.): $z_{sc} = U_{sc}/100$

Short circuit resistance (p.u.): $r_{sc} = \frac{P_{cu}/1000}{S_n}$

Short circuit reactance (p.u.): $x_{sc} = \sqrt{z_{sc}^2 - r_{sc}^2}$

HV resistance (p.u.): $r_{cu,hv} = r_{sc} \cdot GR_{hv1}$

LV resistance (p.u.): $r_{cu,lv} = r_{sc} \cdot (1 - GR_{hv1})$

HV shunt reactance (p.u.): $xs_{hv} = x_{sc} \cdot GX_{hv1}$

LV shunt reactance (p.u.): $xs_{lv} = x_{sc} \cdot (1 - GX_{hv1})$

Shunt resistance (p.u.): $r_{fe} = \frac{Sn}{P_{fe}/1000}$

Magnetization impedance (p.u.): $z_m = \frac{1}{I_0/100}$

Magnetization reactance (p.u.): $x_m = \dfrac{1}{\sqrt{\frac{1}{z_m^2} - \frac{1}{r_{fe}^2}}}$
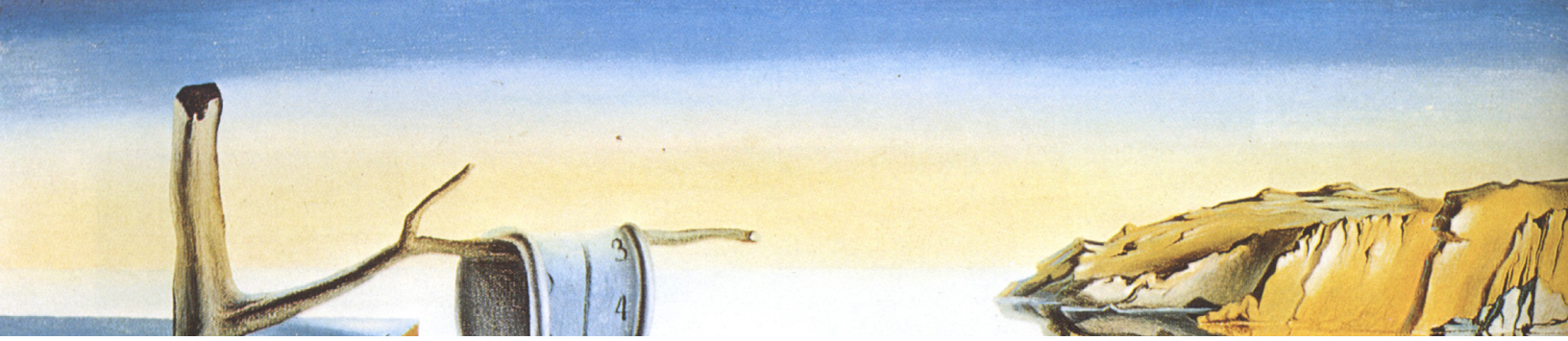
If the content of the square root is negative, set the magnetization impedance to zero.

The final complex calculated parameters in per unit are:
Magnetizing impedance (or series impedance): $z_{series} = Z_m = r_{fe} + j \cdot x_m$
Leakage impedance (or shunt impedance): $Z_l = r_{sc} + j \cdot x_{sc}$
Shunt admittance: $y_{shunt} = 1/Z_l$

4. Power flow methods

4.1 Newton-Raphson-Iwamoto

4.2 Holomorphic Embedding (ASU)

First introduced by Antonio Trias in 2012 [2], promises to be a non-divergent power flow method. Trias originally developed a version with no voltage controlled nodes (PV), in which the convergence properties are excellent (With this software try to solve any grid without PV nodes to check this affirmation).

The version programmed in the file `HelmVect.py` has been adapted from the master thesis of Muthu Kumar Subramanian at the Arizona State University [1]. This version includes a formulation of the voltage controlled nodes. My experience indicates that the introduction of the PV control deteriorates the convergence properties of the holomorphic embedding method. However, in many cases, it is the best approximation to a solution. especially when Newton-Raphson does not provide one.

The `HelmVect.py` file contains a vectorized version of the algorithm. This means that the execution in python is much faster than a previous version that uses loops.

4.2.1 Concepts

All the power flow algorithms until the HELM method was introduced were iterative and recursive. The helm method is iterative but not recursive. A simple way to think of this is that traditional power flow methods are exploratory, while the HELM method is a planned journey. In theory the HELM method is superior, but in practice the numerical degeneration makes it less ideal.

The fundamental idea of the recursive algorithms is that given a voltage initial point (1 p.u. at every node, usually) the algorithm explores the surroundings of the initial point until a suitable voltage solution is reached or no solution at all is found because the initial point is supposed to be "far" from the solution.

On the HELM methods, we form a "curve" that departures from a known mathematically exact solution that is obtained from solving the grid with no power injections. This is possible because with no power injections, the grid equations become linear and straight forward to solve. The arriving point of the "curve" is the solution that we want to achieve. That "curve" is best approximated by a Padè approximation. To compute the Padè approximation we need to compute coefficient of the unknown variables, in our case the voltage (and possibly the reactive power).

The HELM formulation consists in the derivation of formulas that enable the calculation of the coefficients of the series that describes the "curve" from the mathematically know solution to the unknown solution. Once the coefficients are obtained, the Padè approximation computes the voltage solution at the "end of the curve", providing the desired voltage solution. The more coefficients we compute the more exact the solution is (this is true until the numerical precision limit is reached).

All this sounds very strange, but it works ;)

If you want to get familiar with this concept, you should read about the homotopy concept. In practice the continuation power flow does the same as the HELM algorithm, it takes a known solution and changes the loading factors until a solution for another state is reached.

4.2.2 Fundamentals

The fundamental equation that defines the power flow problem is:

$$S = V \times (Y \times V + I)^* \tag{4.1}$$

Most usefully represented like this:

$$\mathbf{Y} \times \mathbf{V} = \left(\frac{\mathbf{S}}{\mathbf{V}}\right)^* \tag{4.2}$$

The holomorphic embedding is to insert a "travelling" parameter α, such that for $\alpha = 0$ we have an mathematically exact solution of the problem (but not the solution we're looking for...), and for $\alpha = 1$ we have the solution we're looking for. The other thing to do is to represent the variables to be computes as McLaurin series. We'll go step by step.

For $\alpha = 0$ we say that $S = 0$, in this way the equation 4.2 becomes linear, and its solution is mathematically exact. But for that to be useful in our case we need to split the admittance matrix Y into Y_{series} and Y_{shunt}. Y_{shunt} is a diagonal matrix, so it can be expressed as a vector instead (no need for matrix-vector product).

$$\mathbf{Y}_{series} \times \mathbf{V} = \left(\frac{\mathbf{S}}{\mathbf{V}}\right)^* - \mathbf{Y}_{shunt}\mathbf{V} \tag{4.3}$$

This is what will allow us to find the zero "state" in the holomorphic series calculation. For $\alpha = 1$ we say that $S = S$, so we don't know the voltage solution, however we can determine a path to get there:

$$\mathbf{Y} \times \mathbf{V}(\alpha) = \left(\frac{\alpha\mathbf{S}}{\mathbf{V}(\alpha)}\right)^* - \alpha\mathbf{Y}_{shunt} \times \mathbf{V}(\alpha) = \frac{\alpha\mathbf{S}^*}{\mathbf{V}(\alpha)^*} - \alpha\mathbf{Y}_{shunt}\mathbf{V}(\alpha) \tag{4.4}$$

Wait, what?? did you just made this stuff up??, well so far my reasoning is:
- The voltage (V) is what I have to convert into a series, and the series depend of α, so it makes sense to say that V is now $V(\alpha)$.
- Regarding the α that multiplies the S, the power (S) is what I vary during the *travel* from $\alpha = 0$ to $\alpha = 1$, so that is why it has to be accompanied by the *traveling* parameter α.
- In my opinion the α $\mathbf{Y}_{shunt}$ is to provoke the first voltage coefficients to be one. $\mathbf{Y}_{series} \times \mathbf{V}[0] = 0$, makes $V[0] = 1$, and this is essential for later steps (to be able to use Padè).

The series are expressed as McLaurin equations::

$$V(\alpha) = \sum_{n}^{\infty} V_n \alpha^n \tag{4.5}$$

There's still something to do. The magnitude $(\mathbf{V}(\alpha))^*$ has to be converted into $(\mathbf{V}(\alpha^*))^*$. This is done in order to make the function be actually holomorphic. The holomorphicity condition is tested by the Cauchy-Riemann condition, this is $\partial V/\partial\alpha^* = 0$, let's check that:

$$\partial\left(\mathbf{V}(\alpha)^*\right)/\partial\alpha^* = \partial\left(\sum_{n}^{\infty} V_n^*(\alpha^n)^*\right)/\partial\alpha^* = \sum_{n}^{\infty} \alpha^n V_n^*(\alpha^{n-1})^* \tag{4.6}$$

Which is not zero, obviously. Now with the proposed change:

$$\partial \left(\mathbf{V}(\alpha^*) \right)^* / \partial \alpha^* = \partial \left(\sum_n^\infty \mathbf{V}_n^* \alpha^n \right) / \partial \alpha^* = 0 \tag{4.7}$$

Yay!, now we're mathematically happy, since this stuff has no effect in practice because our α is not going to be a complex parameter, but for sake of being correct the equation is now:

$$\mathbf{Y}_{series} \times \mathbf{V}(\alpha) = \frac{\alpha \mathbf{S}^*}{\mathbf{V}^*(\alpha^*)} - \alpha \mathbf{Y}_{shunt} \mathbf{V}(\alpha) \tag{4.8}$$

The fact that $\mathbf{V}^*(\alpha^*)$ is dividing is problematic. We need to express it as its inverse so it multiplies instead of divide.

$$\frac{1}{\mathbf{V}(\alpha)} = \mathbf{W}(\alpha) \longrightarrow \mathbf{W}(\alpha)\mathbf{V}(\alpha) = 1 \longrightarrow \sum_{n=0}^\infty \mathbf{W}_n \alpha^n \sum_{n=0}^\infty \mathbf{V}_n \alpha^n = 1 \tag{4.9}$$

Expanding the series and identifying terms of α we obtain the expression to compute the inverse voltage series coefficients:

$$\mathbf{W}_n = \begin{cases} \dfrac{1}{\mathbf{V}_0}, & n = 0 \\ \dfrac{\sum_{k=0}^{n-1} \mathbf{W}_k \mathbf{V}_{n-k}}{\mathbf{V}_0}, & n > 0 \end{cases} \tag{4.10}$$

Now, the equation 4.8 is:

$$\mathbf{Y}_{series} \times \mathbf{V}(\alpha) = \alpha \mathbf{S}^* \mathbf{W}(\alpha)^* - \alpha \mathbf{Y}_{shunt} \mathbf{V}(\alpha) \tag{4.11}$$

Substituting the series by their McLaurin expressions:

$$\mathbf{Y}_{series} \times \sum_{n=0}^\infty \mathbf{V}_n \alpha^n = \alpha \mathbf{S}^* \left(\sum_{n=0}^\infty \mathbf{W}_n \alpha^n \right)^* - \alpha \mathbf{Y}_{shunt} \sum_{n=0}^\infty \mathbf{V}_n \alpha^n \tag{4.12}$$

Expanding the series an identifying terms of α we obtain the expression for the voltage coefficients:

$$\mathbf{V}_n = \begin{cases} 0, & n = 0 \\ \mathbf{S}^* \mathbf{W}_{n-1}^* - Y_{shunt} \mathbf{V}_{n-1}, & n > 0 \end{cases} \tag{4.13}$$

This is the HELM fundamental formula derivation for a grid with no voltage controlled nodes (no PV nodes). Once a sufficient number of coefficients are obtained, we still need to use the Padè approximation to get voltage values out of the series.

In the previous formulas, the number of the bus has not been explicitly detailed. All the $\mathbf{V}$ and the $\mathbf{W}$ are matrices of dimension $n \times nbus$ (number of coefficients by number of buses in the grid) This structures are depicted in the figure 4.1.
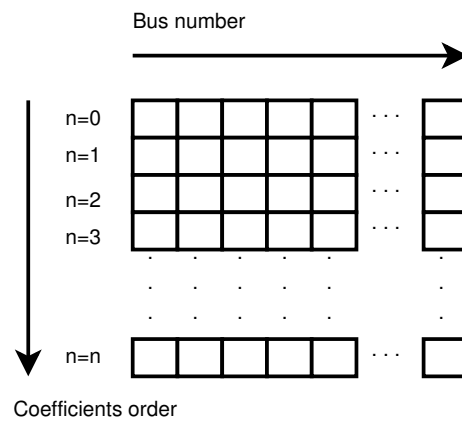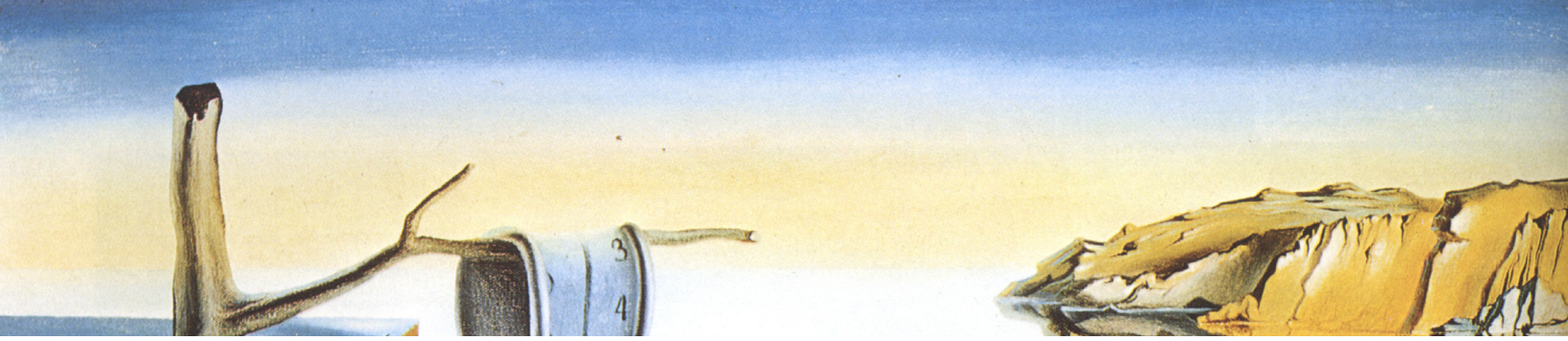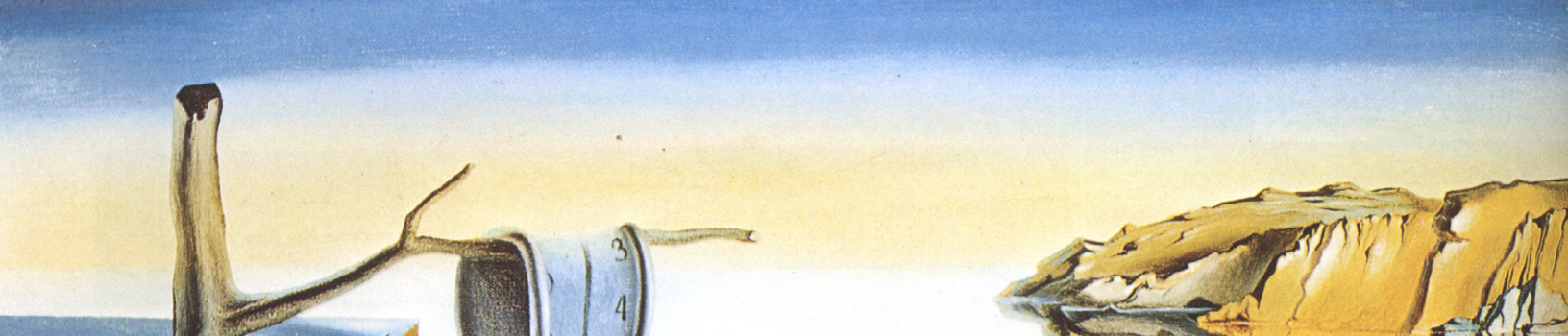
Bus number

n=0
n=1
n=2
n=3

n=n

Coefficients order

Figure 4.1: Structure of the coefficients

5. Graphical User Interface

Bibliography

[1] M. K. Subramanian. Application of holomorphic embedding to the power-flow problem, 2014.

[2] A. Trias. The holomorphic embedding load flow method. pages 1–8, July 2012.