GridCal

Santiago Peñate Vera

GRIDCAL

Research oriented power systems software.

Started writing this document in Madrid the 9th of October of 2016

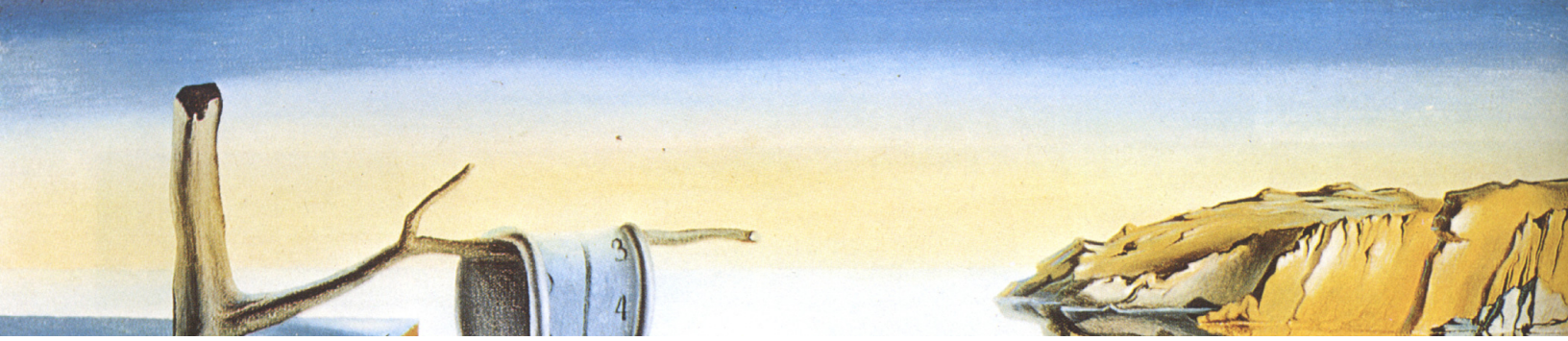
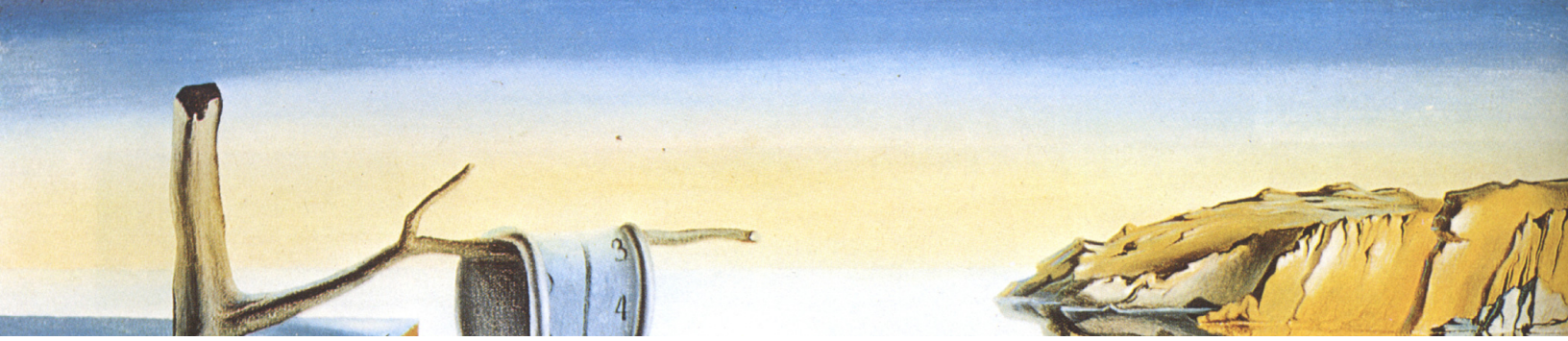


Table of content



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

1.1 installation

GridCal is designed to work with python 3.5 and onwards, hence retro-compatibility is not guaranteed.

1.1.1 Windows

Open a system console (go to the desktop menu and type `cmd`, the console should appear). Then type:

```
pip install GridCal
```

This assumes that your system-wide python is a python 3 distribution, hence you must make sure of this. An easy way to check this, is to open a console, type `python` and if a python 3 console opens within the terminal, then it is allright.

For windows systems I have disabled the installation of PyQt (the user interface technology) because it conflicts with the Qt version provided by Anaconda. This should not be the case, but it happens. Therefore, under windows, you must install python through Anaconda.

1.1.2 Linux / OSX

On Unix systems the python 2 / 3 issue is non existent since the terminal commands for python 2 and python 3 are different. So, simply go to a system terminal and type:

```
pip3 install GridCal
```

This command will install all the dependencies flawlessly unlike under windows.

1.2 Run with user interface

You must have Python 3.5 or higher installed to work with the GUI. From a Python console:

```
from GridCal.ExecuteGridCal import run
run()
```

1.2.1 Windows

To run GridCal with GUI directly from a system console, type:

```
python -c "from GridCal.ExecuteGridCal import run; run()"
```

You can embed this command in a shortcut file, and have a windows shortcut in the desktop for GridCal.

1.2.2 Linux / OSX

To run GridCal with GUI directly from a system console, type:

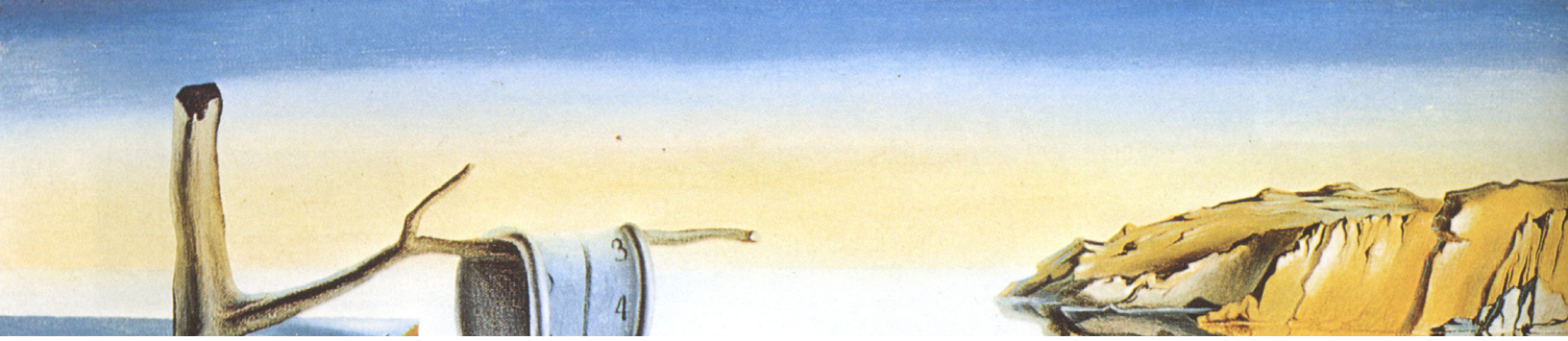
```
python3 -c "from GridCal.ExecuteGridCal import run; run()"
```

1.3 Using GridCal as a library

You can use the calculation engine directly or from other applications:

```
from GridCal.grid.CircuitOO import *
```

This will provide access to all the objects in the calculation engine of GridCal.



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. After having been involved in quite some number-crunching software developments, I have found this approach to be the best compromise between efficiency and code escalability and maintainability .

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.

2.1.1 Circuit

- *Sbase*: Base power to compute the per unit power values (MVA)

- *branch_original_idx*: Array that keeps the index of the branches in the parent MultiCircuit object.
- *branches*: List of Branch objects.
- *bus_original_idx*: Array that keeps the index of the buses in the parent MultiCircuit object.
- *buses*: List of Bus objects.
- *graph*: Graph object from the `networkx` package.
- *mc_time_series*
- *monte_carlo_input*
- *name*: Name of the circuit.
- *power_flow_input*
- *power_flow_results*
- *time_series_input*
- *time_series_results*

2.1.2 MultiCircuit (inherits Circuit)

Since the MultiCircuit inherits the Circuit object, it means that it has by default all the properties of the Circuit plus the ones that are below.

- *branch_dictionary*
- *branch_names*: Array with the name of the branch objects all together.
- *bus_dictionary*
- *bus_names*: Array with the name of the bus objects all together.
- *circuits*: List of the Circuit objects. Each Circuit represents an island.
- *has_time_series*: Are there time series available? (True/False)

2.2 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.

- *Qmax_sum*: Maximum reactive power of this bus (inferred from the devices).
- *Qmin_sum*: Minimum reactive power of this bus (inferred from the devices).
- *Vmax*: Maximum voltage of this bus (p.u.)
- *Vmin*: Minimum voltage of this bus (p.u.)
- *Vnom*: Nominal voltage of the bus (kV)
- *batteries*: List of Battery objects.
- *controlled_generators*: List of ControlledGenerator objects.
- *dispatch_storage*: Shall this bus dispatch its storage devices? (True / False)
- *graphic_object*: Qt graphic object associated to this bus.
- *is_enabled*: Is this bus enabled for calculation? (True / False)
- *is_slack*: Is this bus a slack bus? (True / False)
- *loads*: List of Load objects.
- *name*: Name of the element.
- *shunts*: List of Shunt objects.
- *static_generators*: List of StaticGenerator objects.
- *type*: Type of the bus.
- *x*: x coordinate of the bus for representation.

- y: y coordinate of the bus for representation.

2.2.1 Load object

The load object implements the so-called ZIP model, in which the load can be represented by a combination of power (P), current(I), and impedance (Z).

- I: Current (complex, in kA)
- $Iprof$: Pandas DataFrame with the current profile (complex, in kA)
- S: Power (complex in MVA)
- $Sprof$: Pandas DataFrame with the power profile (complex, in MVA)
- Z: Impedance (complex, in Ohm)
- $Zprof$: Pandas DataFrame with the impedance profile (complex, in Ohm)
- *name*: Name of the load.

The sign convention is: Positive to act as a load, negative to act as a generator.

2.2.2 StaticGenerator object

- S: Power (complex in MVA)
- $Sprof$ Pandas DataFrame with the power profile (complex, in MVA)
- *name*: Name of the StaticGenerator.

2.2.3 Battery object

- $Enom$: Nominal energy capacity (MWh)
- P: Active power being dispatched (MW)
- $Pprof$: Pandas DataFrame with the active power profile (real, in MW)
- $Qmax$: reactive power upper limit (p.u.)
- $Qmin$: reactive power lower limit (p.u.)
- $Snom$: Nominal power (MVA)
- $Vset$: Voltage set point (p.u.)
- $Vset\_prof$ Pandas DataFrame with the voltage set point profile (p.u.)
- *name*: Name of the battery.

2.2.4 ControlledGenerator object

- P: Active power being dispatched (MW)
- $Pprof$: Pandas DataFrame with the active power profile (real, in MW)
- $Qmax$: reactive power upper limit (p.u.)
- $Qmin$: reactive power lower limit (p.u.)
- $Snom$: Nominal power (MVA)
- $Vset$: Voltage set point (p.u.)
- $Vset\_prof$ Pandas DataFrame with the voltage set point profile (p.u.)
- *name*: Name of the ControlledGenerator.

2.2.5 Shunt object

- Y: Admittance of the shunt object (in p.u.)
- $Yprof$: Pandas DataFrame with the admittance profile (p.u.)
- *name*: Name of the shunt object.

2.3 Branch object

- *angle*: Tap angle in radians.
- bus_from: Bus object to which the branch is connected at the "from" end.
- bus_to: Bus object to which the branch is connected at the "to" end.
- *is_enabled*: Is this branch enabled for calculation?
- $mttf$: Mean time to failure (h)
- *mttr*: Mean time to repair (h)
- *name*: Name of the branch
- *rate*: Power rating (MVA)
- tap_module: tap changer module (value around 1)
- *y_shunt*: Total shunt admittance.
- *z_series*: Total series impedance.
- $type\_obj$: Object of type TransformerType or LineType used to set the current impedance values.

Implements the model at 3.2.

2.4 TransformerType object

- *HV_nominal_voltage*: High voltage side nominal voltage (kV)
- *LV_nominal_voltage*: Low voltage side nominal voltage (kV)
- *Nominal_power*: Transformer nominal power (MVA)
- *Copper_losses*: Copper losses (kW)
- *Iron_losses*: Iron Losses (kW)
- *No_load_current*: No load current (%)
- *Short_circuit_voltage*: Short circuit voltage (%)
- *GR_hv*1:
- *GX_hv*1:

Implements the model at 3.3.

2.5 element

-
-
-
-
-

2.6 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.

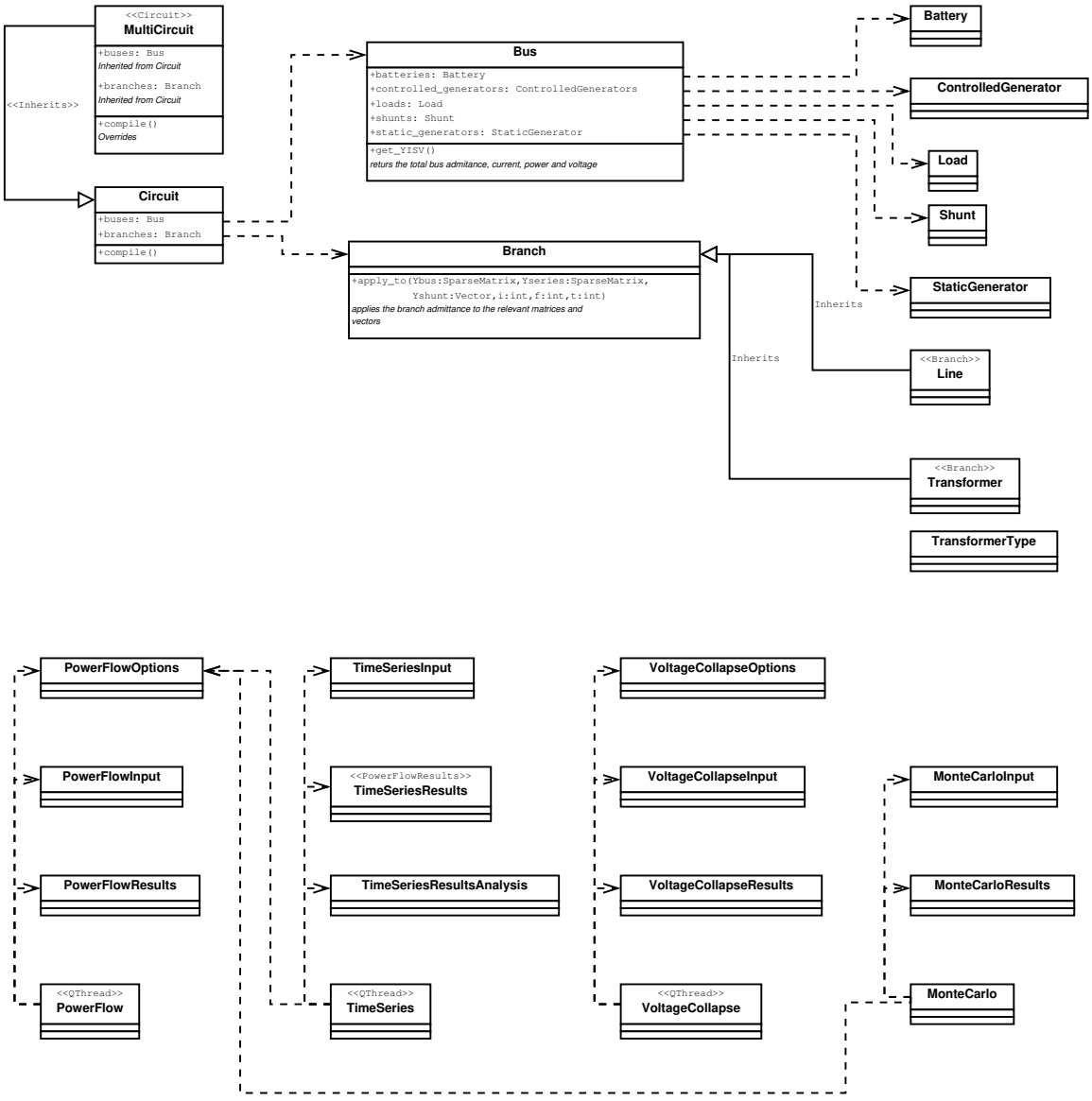
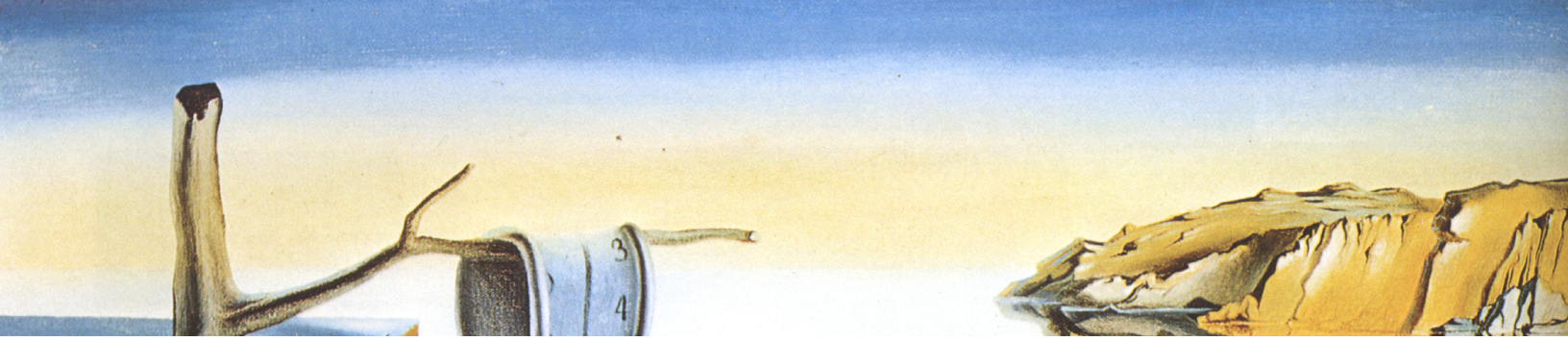


Figure 2.1: Simplified class diagram of GridCal's API



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, the following matrices:

- `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.

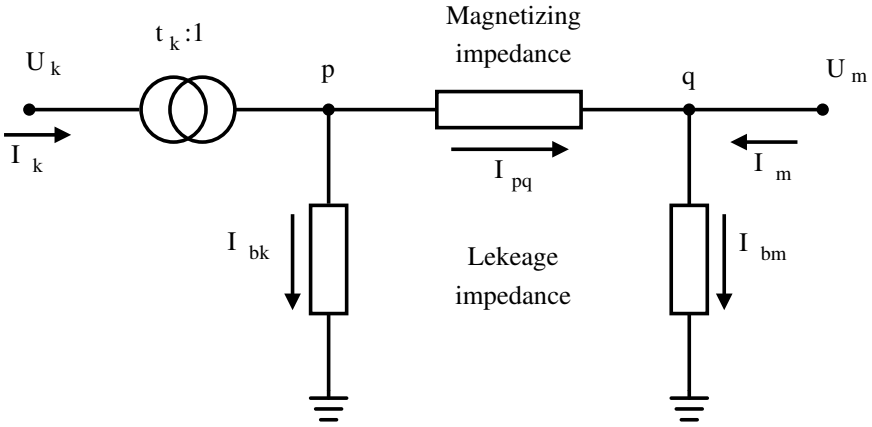


Figure 3.1: π model of a branch

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:

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

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

$$Y_{bust,f} = Y_{bust,f} + Y_{tf}$$

$$Y_{bust,t} = Y_{bust,t} + Y_{tt}$$

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

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

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

$$Y_{seriest,f} = Y_{seriest,f} + Y_{tf}$$

$$Y_{seriest,t} = Y_{seriest,t} + Y_s$$

3.2.3 Yf and Yt

$$Yf_{i,f} = Yf_{i,f} + Yff$$
$$Yf_{i,t} = Yf_{i,t} + Yft$$

$$Yti_{,f} = Yti_{,f} + Ytf$$
$$Yti_{,t} = Yti_{,t} + Ytt$$

Here i is the index of the branch in the circuit and f, t are the corresponding bus indices.

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.): $xs_{lv} = x_{sc} \cdot (1 - GX_{hv1})$

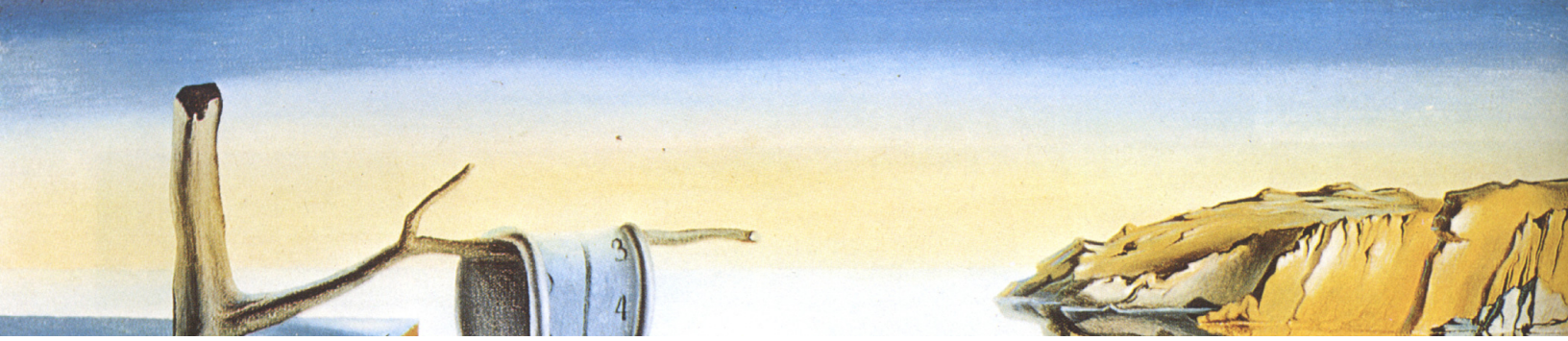
Shunt resistance (p.u.): $r_{fe} = \frac{Sn}{P_{fe}/1000}$

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

Magnetization reactance (p.u.): $x_m = \frac{1}{\sqrt{\frac{1}{z_m^2} - \frac{1}{r_{fe}^2}}}$

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

4.1 Newton-Raphson-Iwamoto

The Newton-Raphson method is the standard power flow method tough at schools. GridCal implements a slight but important modification of this method that turns it into a more robust, industry-standard algorithm. The Newton Raphson method is the first order Taylor approximation of the power flow equation. The method implemented in GridCal is the second order approximation, let's see how.

The expression to update the voltage solution in the Newton-Raphson algorithm is the following:

$$\mathbf{V}_{t+1} = \mathbf{V}_t + \mathbf{J}^{-1}(\mathbf{S}_0 - \mathbf{S}_{calc}) \tag{4.1}$$

Where:

- $\mathbf{V}_t$: Voltage vector at the iteration t (current voltage)
- $\mathbf{V}_{t+1}$: Voltage vector at the iteration $t+1$ (new voltage)
- $\mathbf{J}$: Jacobian matrix.
- $\mathbf{S}_0$: Specified power vector.
- $\mathbf{S}_{calc}$: Calculated power vector using $\mathbf{V}_t$.

The formulation implemented in GridCal is the following:

$$\mathbf{V}_{t+1} = \mathbf{V}_t + \mu\mathbf{J}^{-1}(\mathbf{S}_0 - \mathbf{S}_{calc}) \tag{4.2}$$

Here μ is the Iwamoto optimal step size parameter. In 1982 S.Iwamoto and Y.Tamura present a method [1] where the Jacobian matrix J is only computed at the beginning, and the iteration control parameter μ is computed on every iteration. In GridCal I compute J and μ on every iteration getting a more robust method on the expense of a greater computational effort.

To compute the parameter μ we must do the following:

Theorem 4.1.1 — Computation of μ.

$$\mathbf{J'} = Jacobian(\mathbf{Y}, \mathbf{dV})$$

$$\mathbf{dx} = \mathbf{J}^{-1}(\mathbf{S}_0 - \mathbf{S}_{calc})$$

$$\mathbf{a} = \mathbf{S}_0 - \mathbf{S}_{calc}$$

$$\mathbf{b} = \mathbf{J} \times \mathbf{dx}$$

$$\mathbf{c} = \frac{1}{2}\mathbf{dx} \cdot (\mathbf{J'} \times \mathbf{dx})$$

$$g_0 = -\mathbf{a} \cdot \mathbf{b}$$

$$g_1 = \mathbf{b} \cdot \mathbf{b} + 2\mathbf{a} \cdot \mathbf{c}$$

$$g_2 = -3\mathbf{b} \cdot \mathbf{c}$$

$$g_3 = 2\mathbf{c} \cdot \mathbf{c}$$

$$G(x) = g_0 + g_1 x + g_2 x^2 + g_3 x^3$$

$$\mu = solve(G(x), x_0 = 1)$$

There will be three solutions to the polynomial $G(x)$. Only the last solution will be real, and therefore it is the only valid value for μ. The polynomial can be solved numerically using 1 as the seed.

The matrix **J'** is the Jacobian matrix computed using the voltage derivative numerically computed as the voltage increment $\mathbf{dV} = \mathbf{V}_t - \mathbf{V}_{t-1}$ (voltage difference between the current and the previous iteration).

4.2 Levenberg-Marquardt

The Levenberg-Marquardt iterative method is often used to solve non-linear least squares problems. in those problems one reduces the calculated error by iteratively solving a system of equations that provides the increment to add to the solution in order to decrease the solution error. So conceptually it applies to the power flow problem.

Set the initial values:

- $\nu = 2$
- $f_{prev} = 10^9$
- $ComputeH = true$

In every iteration:

- Compute the jacobian matrix if $ComputeH$ is $true$:

$$\mathbf{H} = Jacobian(\mathbf{Y}, \mathbf{V})$$

- Compute the mismatch in the same order as the jacobian:

$$\mathbf{S}_{calc} = \mathbf{V}(\mathbf{Y} \cdot \mathbf{V} - \mathbf{I})^*$$

$$\mathbf{m} = \mathbf{S}_{calc} - \mathbf{S}$$

$$\mathbf{dz} = [Re(\mathbf{m}_{pv}), Re(\mathbf{m}_{pq}), Im(\mathbf{m}_{pq})]$$

- Compute the auxiliary jacobian transformations:

$$\mathbf{H}_1 = \mathbf{H}^\top$$

$$\mathbf{H}_2 = \mathbf{H}_1 \cdot \mathbf{H}$$

- Compute the first value of λ (only in the first iteration):

$$\lambda = 10^{-3} Max(Diag(\mathbf{H}_2))$$

- Compute the system Matrix:

$$\mathbf{A} = \mathbf{H}_2 + \lambda \cdot Identity$$

- Compute the linear system right hand side:

$$\mathbf{rhs} = \mathbf{H}_1 \cdot \mathbf{dz}$$

- Solve the system increment:

$$\mathbf{dx} = Solve(\mathbf{A}, \mathbf{rhs})$$

- Compute the objective function:

$$f = 0.5 \cdot \mathbf{dz} \cdot \mathbf{dz}^\top$$

- Compute the decision function:

$$\rho = \frac{f_{prev} - f}{0.5 \cdot \mathbf{dx}^\top \cdot (\lambda \mathbf{dx} + \mathbf{rhs})}$$

- Update values:
  If $\rho > 0$
  - $ComputeH = true$
  - $\lambda = \lambda \cdot max(1/3, 1 - (2 \cdot \rho - 1)^3)$
  - $\nu = 2$
  - Update the voltage solution using **dx**.

  Else
  - $ComputeH = false$
  - $\lambda = \lambda \cdot \nu$
  - $\nu = \nu \cdot 2$
- Compute the convergence:

$$converged = ||dx, \infty|| < tolerance$$

- $f_{prev} = f$

As you can see it takes more steps than Newton-Raphson. It is a slower method, but it works better for ill-conditioned and large grids.

4.3 DC approximation

The so called direct current power flow (or just DC power flow) is a convenient oversimplification of the power flow procedure.

It assumes that in any branch the reactive part of the impedance is much larger than the resistive part, hence the resistive part is neglected, and that all the voltages modules are the nominal per unit values. This is, $|v| = 1$ for load nodes and $|v| = v_{set}$ for the generator nodes, where v_{set} is the generator set point value.

In order to compute the DC approximation we must perform a transformation. The slack nodes are removed from the grid, and their influence is maintained by introducing equivalent currents in all the nodes. The equivalent admittance matrix ($\mathbf{Y}_{red}$) is obtained by removing the rows and columns corresponding to the slack nodes. Likewise the removed elements conform the ($\mathbf{Y}_{slack}$) matrix.

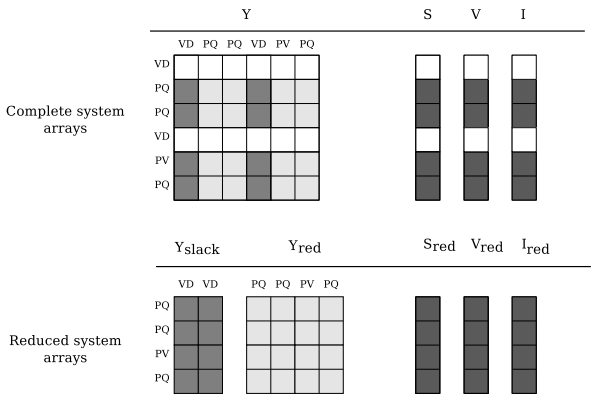


Figure 4.1: Matrix reduction (VD: Slack, PV: Voltage controlled, PQ: Power controlled)

$$\mathbf{P} = real(\mathbf{S}_{red}) + (-imag(\mathbf{Y}_{slack}) \cdot angle(\mathbf{V}_{slack}) + real(\mathbf{I}_{red})) \cdot |\mathbf{V}_{red}| \tag{4.3}$$

The equation 4.3 computes the DC power injections as the sum of the different factors mentioned:

1. $real(\mathbf{S}_{red})$: Real part of the reduced power injections.
2. $imag(\mathbf{Y}_{slack}) \cdot angle(\mathbf{V}_{slack}) \cdot |v_{red}|$: Currents that appear by removing the slack nodes while keeping their influence, multiplied by the voltage module to obtain power.
3. $real(\mathbf{I}_{red}) \cdot |v_{red}|$: Real part of the grid reduced current injections, multiplied by the voltage module to obtain power.

Once the power injections are computed, the new voltage angles are obtained by:

$$\mathbf{V}_{angles} = imag(\mathbf{Y}_{red})^{-1} \times \mathbf{P} \tag{4.4}$$

The new voltage is then:

$$\mathbf{V}_{red} = |\mathbf{V}_{red}| \cdot e^{1j \cdot \mathbf{V}_{angles}} \tag{4.5}$$

This solution does usually produces a large power mismatch. That is to be expected because the method is an oversimplification with no iterative convergence criteria, just a straight forward set of operations.

4.4 Holomorphic Embedding (ASU)

First introduced by Antonio Trias in 2012 [3], 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 (ASU) [2]. 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.

4.4.1 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 the coefficients of the unknown variables, in our case the voltages (and possibly the reactive powers at the PV nodes).

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.

4.4.2 Fundamentals

The fundamental equation that defines the power flow problem is:

$$\mathbf{S} = \mathbf{V} \times (\mathbf{Y} \times \mathbf{V})^* \tag{4.6}$$

Most usefully represented like this:

$$\mathbf{Y} \times \mathbf{V} = \left(\frac{\mathbf{S}}{\mathbf{V}}\right)^* \tag{4.7}$$

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 computed as McLaurin series. Let's go step by step.

For $\alpha = 0$ we say that $S = 0$, in this way the equation 4.7 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).

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

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:

$$\mathbf{Y} \times \mathbf{V}(\alpha) = \left(\frac{\alpha\mathbf{S}}{\mathbf{V}(\alpha)}\right)^* - \alpha\mathbf{Y}_{shunt} \times \mathbf{V}(\alpha) = \frac{\alpha\mathbf{S}^*}{\mathbf{V}(\alpha)^*} - \alpha\mathbf{Y}_{shunt}\mathbf{V}(\alpha) \tag{4.9}$$

Wait, what?? did you just made this stuff up??, well so far my reasoning is:

- The voltage $\mathbf{V}$ is what I have to convert into a series, and the series depend of α, so it makes sense to say that $\mathbf{V}$, as it is dependent of α, becomes $\mathbf{V}(\alpha)$.
- Regarding the α that multiplies $\mathbf{S}$, the amount of power ($\alpha\mathbf{S}$) is what I vary during the *travel* from $\alpha = 0$ to $\alpha = 1$, so that is why $\mathbf{S}$ has to be accompanied by the *traveling* parameter α.
- In my opinion the $\alpha\,\mathbf{Y}_{shunt}$ is to provoke the first voltage coefficients to be one. $\mathbf{Y}_{series} \times \mathbf{V}[0] = 0$, makes $V[0] = 1$. This is essential for later steps (is a condition to be able to use Padè).

The series are expressed as McLaurin equations:

$$V(\alpha) = \sum_n^\infty V_n \alpha^n \tag{4.10}$$

Theorem 4.4.1 — Holomorphicity check. 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 holomorphic. The holomorphicity condition is tested by the Cauchy-Riemann condition, this is $\partial\mathbf{V}/\partial\alpha^* = 0$, let's check that:

$$\partial\left(\mathbf{V}(\alpha)^*\right)/\partial\alpha^* = \partial\left(\sum_n^\infty V_n^*(\alpha^n)^*\right)/\partial\alpha^* = \sum_n^\infty \alpha^n V_n^*(\alpha^{n-1})^* \tag{4.11}$$

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

$$\partial\left(\mathbf{V}(\alpha^*)\right)^*/\partial\alpha^* = \partial\left(\sum_n^{\infty}\mathbf{V}_n^*\alpha^n\right)/\partial\alpha^* = 0 \tag{4.12}$$

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:

$$\mathbf{Y}_{series}\times\mathbf{V}(\alpha) = \frac{\alpha\mathbf{S}^*}{\mathbf{V}^*(\alpha^*)} - \alpha\mathbf{Y}_{shunt}\mathbf{V}(\alpha) \tag{4.13}$$

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

$$\frac{1}{\mathbf{V}(\alpha)} = \mathbf{W}(\alpha) \longrightarrow \mathbf{W}(\alpha)\mathbf{V}(\alpha) = 1 \longrightarrow \sum_{n=0}^{\infty}\mathbf{W}_n\alpha^n\sum_{n=0}^{\infty}\mathbf{V}_n\alpha^n = 1 \tag{4.14}$$

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

$$\mathbf{W}_n = \begin{cases} \dfrac{1}{\mathbf{V}_0}, & n = 0 \\ \dfrac{\sum_{k=0}^{n-1}\mathbf{W}_k\mathbf{V}_{n-k}}{\mathbf{V}_0}, & n > 0 \end{cases} \tag{4.15}$$

Now, the equation 4.13 is:

$$\mathbf{Y}_{series}\times\mathbf{V}(\alpha) = \alpha\mathbf{S}^*\cdot\mathbf{W}(\alpha)^* - \alpha\mathbf{Y}_{shunt}\mathbf{V}(\alpha) \tag{4.16}$$

Substituting the series by their McLaurin expressions:

$$\mathbf{Y}_{series}\times\sum_{n=0}^{\infty}\mathbf{V}_n\alpha^n = \alpha\mathbf{S}^*\left(\sum_{n=0}^{\infty}\mathbf{W}_n\alpha^n\right)^* - \alpha\mathbf{Y}_{shunt}\sum_{n=0}^{\infty}\mathbf{V}_n\alpha^n \tag{4.17}$$

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

$$\mathbf{V}_n = \begin{cases} 0, & n = 0 \\ \mathbf{S}^*\mathbf{W}_{n-1}^* - Y_{shunt}\mathbf{V}_{n-1}, & n > 0 \end{cases} \tag{4.18}$$

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 4.2. For instance $\mathbf{V}_n$ is the n^{th} row of the coefficients structure $\mathbf{V}$.

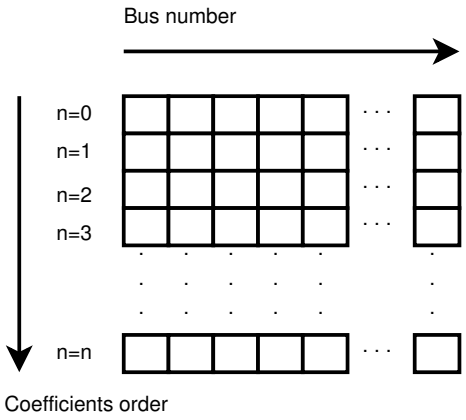


Figure 4.2: Structure of the coefficients

4.4.3 Padè approximation

The equation 4.10 provides us with an expression to obtain the voltage from the coefficients, knowing that for $\alpha = 1$ we get the final voltage results. So, why do we need any further operation?, and what is this Padè thing?

Well, it is true that the equation 4.10 provides an approximation of the voltage by means of a series (this is similar to a Taylor approximation), but in practice, the approximation might provide a wrong value for a given number of coefficients. The Padè approximation accelerates the convergence of any given series, so that you get a more accurate result with less coefficients. This means that for the same series of voltage coefficients, using the equation 4.10 could give a completely wrong result, whereas by applying Padè to those coefficients one could obtain a fairly accurate result.

The Padè approximation is a rational approximation of a function. In our case the function is $\mathbf{V}(\alpha)$, represented by the coefficients structure $\mathbf{V}$. The approximation is valid over a small domain of the function, in our case the domain is $\alpha = [0,1]$. The method requires the function to be continuous and differentiable for $\alpha = 0$. Hence the Cauchy-Riemann condition. And yes, our function meets this condition, we tested it before.

GridCal implements two algorithms that perform the Padè approximation; The Padè canonical algorithm, and Wynn's Padè approximation.

Padè approximation algorithm

The canonical Padè algorithm for our problem is described by:

$$Voltage\_value\_approximation = \frac{P_N(\alpha)}{Q_M(\alpha)} \quad \forall \alpha \in [0,1] \tag{4.19}$$

Here $N = M = n/2$, where n is the number of available voltage coefficients, which has to be an even number to be exactly divisible by 2. P and Q are polynomials which coefficients p_i and q_i must be computed. It turns out that if we make the first term of $Q_M(\alpha)$ be $q_0 = 1$, the function to be approximated is given by the McLaurin expression (What a happy coincidence!)

$$P_N(\alpha) = p_0 + p_1\alpha + p_2\alpha^2 + ... + p_N\alpha^N \tag{4.20}$$

$$Q_M(\alpha) = 1 + q_1\alpha + q_2\alpha^2 + ... + q_M\alpha^M \tag{4.21}$$

The problem now boils down to find the coefficients q_i and p_i. This is done by solving two systems of equations. The first one to find q_i which does not depend on p_i, and the second one to get p_i which does depend on q_i.

First linear system: The only unknowns are the q_i coefficients.

$$\begin{aligned} q_M V_{N-M+1} + q_{M-1} V_{N-M+2} + ... + q_1 V_N &= 0 \\ q_M V_{N-M+2} + q_{M-1} V_{N-M+3} + ... + q_1 V_{N+1} &= 0 \\ &... \\ q_M V_N + q_{M-1} V_{N+1} + ... + q_1 V_{N+M+1} + V_{N+M} &= 0 \end{aligned} \tag{4.22}$$

Second linear System: The only unknowns are the p_i coefficients.

$$\begin{aligned} V_0 - p_0 &= 0 \\ q_1 V_0 + V_1 p_1 &= 0 \\ q_2 V_0 + q_1 V_1 + V_2 - p_2 &= 0 \\ q_3 V_0 + q_2 V_1 + q_1 V_2 + V_3 - p_3 &= 0 \\ &... \\ q_M V_{N-M} + q_{M-1} V_{N-M+1} + ... + +V_N - p_N &= 0 \end{aligned} \tag{4.23}$$

Once the coefficients are there, you would have defined completely the polynomials $P_N(\alpha)$ and $Q_M(\alpha)$, and it is only a matter of evaluating the equation 4.19 for $\alpha = 1$.

This process is done for every column of coefficients $\mathbf{V} = \{V_0, V_1, V_2, V_3, ..., V_n\}$ of the structure depicted in the figure 4.2. This means that we have to perform a Padè approximation for every node, using the one columns of the voltage coefficients per Padé approximation.

Wynn's Padè approximation algorithm

Wynn published a paper in 1969 where he proposed a simple calculation method to obtain the Padè approximation. This method is based on a table. Weniger in 1989 publishes his thesis where a faster version of Wynn's algorithm is provided in Fortran code.

That very Fortran piece of code has been translated into Python and included in GridCal.

One of the advantages of this method over the canonical Padè approximation implementation is that it can be used for every iteration. In the beginning I thought it would be faster but it turns out that it is not faster since the amount of computation increases with the number of coefficients, whereas with the canonical implementation the order of the matrices does not grow dramatically and it is executed the half of the times.

On top of that my experience shows that the canonical implementation provides a more consistent convergence.

Anyway, both implementations are there to be used in the code.

4.4.4 PV nodes formulation

To be done.

4.5 Post power flow (loading and losses)

As we have seen, the power flow routines compute the voltage at every bus of an island grid. However we do not get from those routines the "power flow" values, this is the power that flows through the branches of the grid. In this section I show how the *post power flow* values are computed.

First we compute the branches per unit currents:

$$\mathbf{I}_f = \mathbf{Y}_f \times \mathbf{V} \tag{4.24}$$

$$\mathbf{I}_t = \mathbf{Y}_t \times \mathbf{V} \tag{4.25}$$

These are matrix-vector multiplications. The result is the per unit currents flowing through a branch seen from the *from* bus or from the *to* bus.

Then we compute the power values:

$$\mathbf{S}_f = \mathbf{V}_f \cdot \mathbf{I}_f^* \tag{4.26}$$

$$\mathbf{S}_t = \mathbf{V}_t \cdot \mathbf{I}_t^* \tag{4.27}$$

These are element-wise multiplications, resulting in the per unit power flowing through a branch seen from the *from* bus or from the *to* bus.

Now we can compute the losses in MVA as:

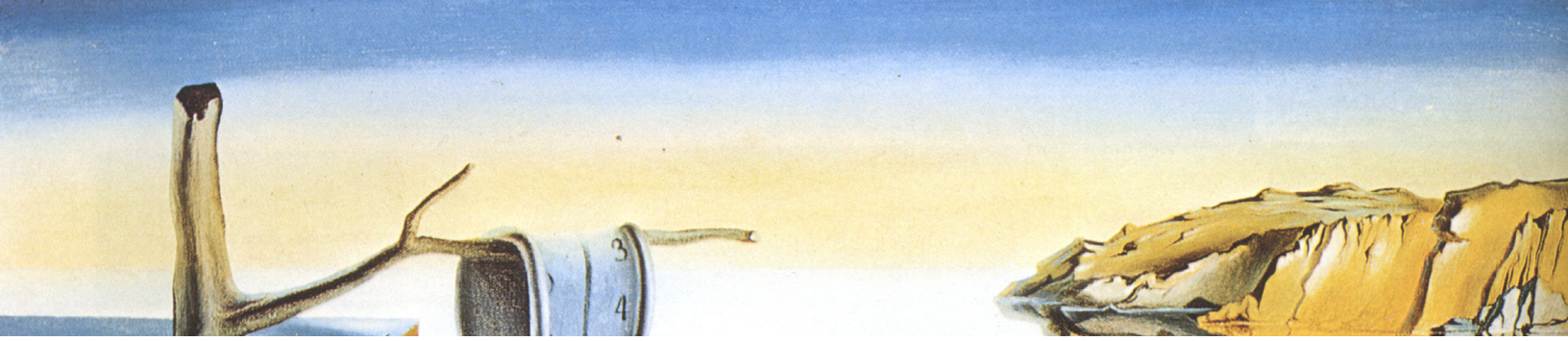
$$\mathbf{losses} = |\mathbf{S}_f - \mathbf{S}_t| \cdot Sbase \tag{4.28}$$

And also the branches loading in per unit as:

$$\mathbf{loading} = \frac{max(|\mathbf{S}_f|, |\mathbf{S}_t|) \cdot Sbase}{\mathbf{rate}} \tag{4.29}$$

The variables are:

- $\mathbf{Y}_f, \mathbf{Y}_t$: *From* and *To* bus-branch admittance matrices, see section 3.1.
- $\mathbf{I}_f$: Array of currents at the *from* buses in p.u.
- $\mathbf{I}_t$: Array of currents at the *to* buses in p.u.
- $\mathbf{S}_f$: Array of powers at the *from* buses in p.u.
- $\mathbf{S}_t$: Array of powers at the *to* buses in p.u.
- $\mathbf{V}_f$: Array of voltages at the *from* buses in p.u.
- $\mathbf{V}_t$: Array of voltages at the *to* buses in p.u.
- **rate**: Array of branch ratings in MVA.



5. Short circuit

5.1 3-phase short circuit

First, declare an array of zeros of size equal to the number of nodes in the circuit.

$$\mathbf{I} = \{0,0,0,0,...,0\}$$

Then, compute the short circuit current at the selected bus i and assign that value in the i^{th} position of the array $\mathbf{I}$.

$$\mathbf{I}_i = -\frac{\mathbf{V}_{pre-failure,i}}{\mathbf{Z}_{i,i} + z_f} \tag{5.1}$$

Then, compute the voltage increment for all the circuit nodes as:

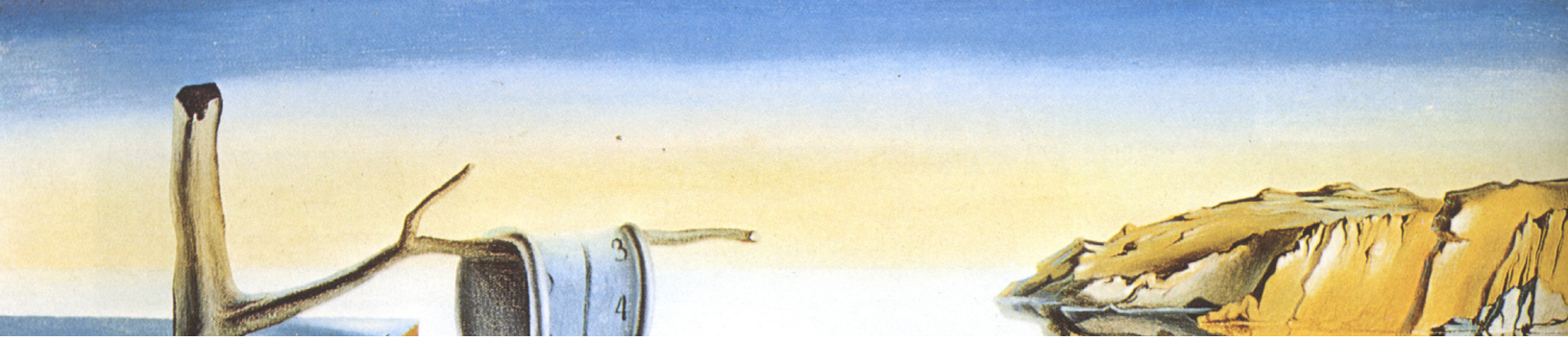
$$\Delta\mathbf{V} = \mathbf{Z} \times \mathbf{I} \tag{5.2}$$

Finally, define the voltage at all the nodes as:

$$\mathbf{V}_{post-failure} = \mathbf{V}_{pre-failure} + \Delta\,\mathbf{V} \tag{5.3}$$

Magnitudes:

- $\mathbf{I}$: Array of fault currents at the system nodes.
- $\mathbf{V}_{pre-failure}$: Array of system voltages prior to the failure. This is obtained from the power flow study.
- z_f: Impedance of the failure itself. This is a given value, although you can set it to zero if you don't know.
- $\mathbf{Z}$: system impedance matrix. Obtained as the inverse of the complete system admittance matrix.



6. Graphical User Interface

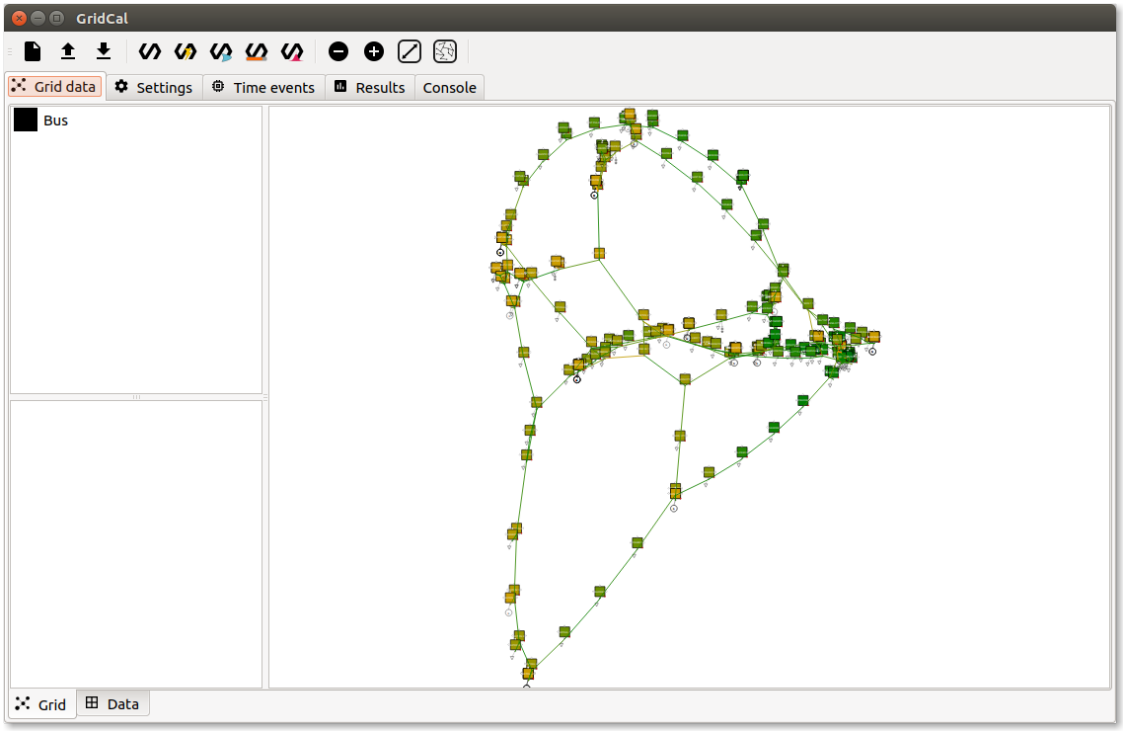
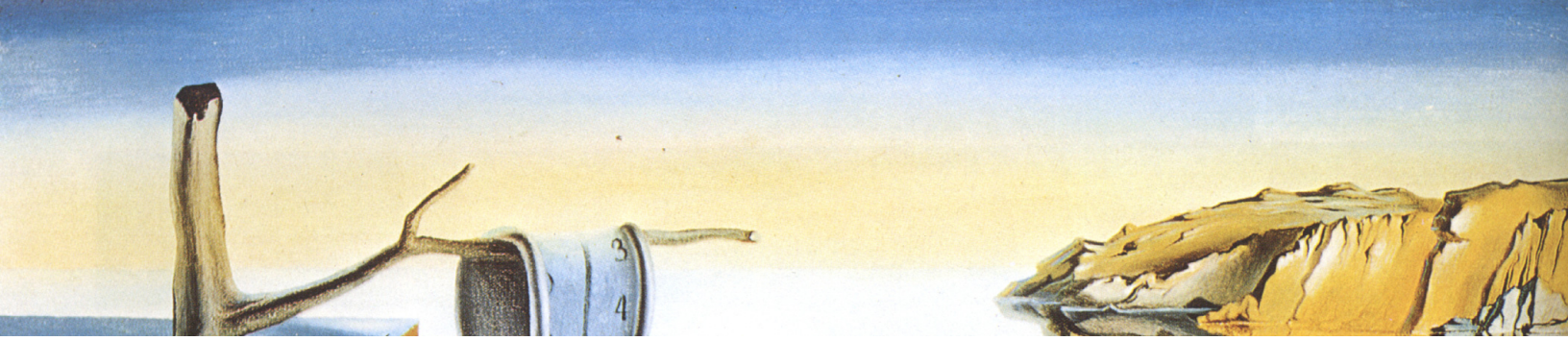


Figure 6.1: GridCal user interface representing the 200-bus Illinois grid.



Bibliography

[1] S. Iwamoto and Y. Tamura. A load flow calculation method for ill-conditioned power systems. *IEEE Transactions on Power Apparatus and Systems*, (4):1736–1743, 1981.

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

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