AURES Documentation and Power Flows— 2012.08

Rolando and Sarah, with help from everyone else

The following document attempts to explain the problem and execution of the constrained DC power flow, as well as technical documentation/User's Guide of the code. Section 1 is unchanged form previous incarnations of the documentation, and deals with the physical interpretation of the DC power flow. Section 2 – also from previous versions – deals with how to define these flows as part of an optimisation problem. Section 3 is a technical explanation of the code and User's Guide, while Section 4 is a Super Short Guide for very impatient people, focused on the case of the 27-Node European Grid. Basically, the less you care about the details, the further on you can start.

1 Modelling Power Flow

1.1 The DC Power Flow as a Simplification of the AC Power Flow

We face the task of describing the flow of power across various nodes in a network. These nodes represent power grids (regions or countries) connected by high voltage transmission lines. For simplicity, we choose to model these flows considering only Kirchhoff's laws. This means that, based on the power generation and consumption (load) in each node, we look for the solution to the problem how power should be transmitted from sources to sinks which minimises the necessary transmission capacities.

This approach neglects market influences, i.e. the fact that power exchanges from country to country are directly controlled using power electronic devices, such as FACTS or HVDC lines. We expect that such economic agreements between countries lead to shortest-path flows, which is not the optimal solution if one is interested in minimising the transmission capacities needed.

In particular, we are interested in a fully renewable scenario, where the overall load is (on average) completely covered by generation of wind and solar energy. For a realistic model, we use actual weather data, which is converted into wind and solar energy generation and then normalised to match the average load. For details about the load and generation time series, see Dominik Heide's PhD thesis. The important parameters in this context are α, which is the percentage of wind energy in the mix, and γ, which is a global factor. Together with the load time series, we get from that a time series of power mismatches for each country, denoted by

$$\begin{aligned} \Delta &= \text{renewable generation} - \text{load} \\ &= \gamma\left(\alpha \cdot \text{wind power} + (1-\alpha) \cdot \text{solar power}\right) - \text{load}. \end{aligned} \tag{1}$$

From the power mismatches, we want to determine the power flows across Europe, subject to various outer conditions like transmission line capacities or different α and γ values.

Power transmission in large networks can be precisely described by AC power flow, which accounts for voltage drops, phase differences, and active and reactive power. Each of the N buses (nodes) of the system is described by four variables: real power mismatch P, reactive power mismatch Q, the voltage V and the voltage phase angle δ. Depending on whether a bus is a generator bus, a load bus, or a reference (slack) bus, different variables are used in engineering. As capacitive lines require energising before being able to transmit power, and inductive loads consume reactive power, the values of P, Q, V, and δ will vary throughout the network. Solving the power flow and finding the power levels of buses and lines requires solving

two sets of equations, one for the active and one for the reactive power.

$$P_i = \sum_{k=1}^{N} |V_i||V_k|(G_{ik}\cos\delta_{ik} + B_{ik}\sin\delta_{ik}) \tag{2}$$

$$Q_i = \sum_{k=1}^{N} |V_i||V_k|(G_{ik}\sin\delta_{ik} - B_{ik}\cos\delta_{ik}) \tag{3}$$

where $\delta_{ik} = \delta_i - \delta_k$ is the difference of the phase angles at nodes i and k, and G_{ik} and B_{ik} are the real and imaginary parts of the admittance of the links connecting nodes i and k, respectively. The consideration of reactive power and voltage is integral for transmission system operators (TSOs) in order to detect instability in transmission and distribution systems, but complicates the solution significantly. These equations are usually solved using the Newton-Raphson method.

For our purposes, we assume a stable system with stable voltage levels and no active or reactive power losses. This means that the power mismatch in each node is equal to its real part, i.e.

$$\Delta_i = P_i, \quad Q_i = 0$$

In this case, we can simplify the AC power flow to a DC power flow: Essentially, our assumptions imply that the voltage levels throughout the grid are constant, so that $V_i = V_k \equiv 1$, that there are no real components in the impedance of the lines (that is, they have resistances $R_{ik} = 0$ such that $G_{ik} = 0$), and the difference between the phase angles is sufficiently small so that $\sin(\delta_{ik}) \approx \delta_{ik}$. This means that we can forget about (**??**), and (**??**) is simplified to

$$\Delta_i = \sum_{k\neq i}^{N} B_{ik}(\delta_i - \delta_k) = \sum_{k\neq i}^{N} B_{ik}\delta_i - \sum_{k\neq i}^{N} B_{ik}\delta_k \equiv \sum L_{ik}\delta_k, \tag{4}$$

if we define

$$L_{ik} = \begin{cases} -B_{ik} & \text{if } i \neq k \\ \sum_{k\neq i} B_{ik} & \text{if } i = k \end{cases}$$

In our case where $B_{ik} = 1$, the L matrix is equal to the Laplace matrix

$$L = KK^T$$

where K is the incidence matrix of the network with the elements

$$K_{nl} = \begin{cases} 1 & \text{if link } l \text{ starts at node } n \\ -1 & \text{if link } l \text{ ends at node } n \\ 0 & \text{otherwise} \end{cases}$$

The values of δ_n are now obtained from (**??**). They define the flows between two nodes, as the power flow F_l along link l that connects nodes i and k is given by

$$F_l = B_{ik}(\delta_i - \delta_k),$$

which, in the case where all the lines have $B_{ik} = 1$, is reduced to

$$F = K^T\delta \tag{5}$$

To sum up, the problem is now to solve for δ in (**??**) in order to find the flows using (**??**) for given Δ_i and network topology. As B does has one eigenvalue zero, this is only possible for Δs that obey the constraint $\sum_i \Delta_i = 0$. Additionally, the solution is only unique up to a global phase, i.e. one can fix e.g. $\delta_0 = 0$. Alternatively, the Moore-Penrose pseudo inverse can be obtained for L, to solve directly, the latter option saving calculation time in very large systems. See Dominik Heide's dissertation for more details.

1.2 The minimum dissipation principle

We now want to include constraints on the flows that model the finite transmission capacity of a power line, i.e. inequalities of the form

$$F = K^T \delta \leq h \tag{6}$$

Additionally, we want to treat cases in which

$$\sum_i \Delta_i \neq 0. \tag{7}$$

This is obviously not possible in our setting so far, since there are no solutions at all if $\sum_i \Delta_i \neq 0$ (no surprise here – that would violate the conservation of energy), and if there are solutions, they are unique up to a global phase and do not allow for further constraints on the flows. What we need here is a reformulation of the problem that allows for the inclusion of (**??**) and (**??**). In this section, we will restate the problem, and in the next section, we will include (**??**) and (**??**).

While we have toyed with the idea of changing the reactances of individual lines to regulate the transmission limits, we found that the problem is easier to solve if we state the power in the nodes directly as a function of the flows, and forget about δ. Eq. (**??**) then simply becomes

$$KF = \Delta \tag{8}$$

We have now passed from a system in N (number of nodes) variables to a system of L (number of links) variables. For a generic power network, L will be larger than N, and therefore, the solution of this problem is no longer unique. Additional solutions arise because Eq. (**??**) can be satisfied by flows that have, in addition to the "net transporting current" that takes power from node to node, "circular components" that flow in a round. Fortunately, it is possible to make it unique and identical to the DC power flow situation by requiring that the square of the flows be minimal. This eliminates the circular flows. It is called the minimum dissipation principle. We show that the solution to Eq. (**??**) together with the minimisation is indeed the same as the one of Eq. (**??**) using a vector of N Lagrange multipliers λ. We are looking for the minimal flows under the constraint that the power is transported from the sources to the sinks.

$$\min_F F^T F - \lambda^T (KF - \Delta). \tag{9}$$

It's easy to see the minimum by completing the squares in F :

$$\left(F - \frac{1}{2}K^T\lambda\right)^T \left(F - \frac{1}{2}K^T\lambda\right) = F^T F - \lambda^T KF + \frac{1}{4}\lambda^T KK^T\lambda$$

so that Eq. (**??**) becomes

$$\min_F \left(F - \frac{1}{2}K^T\lambda\right)^T \left(F - \frac{1}{2}K^T\lambda\right) - \frac{1}{4}\lambda^T KK^T\lambda + \lambda^T\Delta$$

whose minimum evidently lies at $F = 1/2K^T\lambda$, i.e. the minimising flows fulfil an equation of the same form as Eq. (**??**) (the flow is a so-called potential flow). This means that, as long as we minimise the square of the flows $F^T F$ while ensuring that $KF = \Delta$, we are describing the DC power flow, or, in other words, the two formulations are equivalent.

We now turn to the extension of the problem to cases where $\sum_i \Delta \neq 0$. In order to achieve that, we introduce *balancing*, i.e. extra power generation when there is less renewable generation than load, and *curtailment*, i.e. the shedding of overproduction. At the same time, we include the constraints on the flows. Computationally speaking, this is a quadratic programming optimisation problem.

2 Power Flow as a optimisation problem

To calculate balancing needs and occurring flows, we employ the physical DC approximation to the full AC power flow. It is valid as long as the network is in a stable state, i.e. no significant voltage or phase shifts between the nodes, and for a network in which line resistances can be neglected. It has the obvious advantage that it minimises the total dissipation in the network, i.e. the power losses, and the necessary flows.

We first take a look at a situation in which there is no global mismatch between renewable energy generation and load.

$$\sum_{i=1}^{N} \Delta_i = 0 \tag{10}$$

For a network with unconstrained line capacities ("copper flow"), the DC power flow is then given as that set of flows F_l that fulfils

$$KF = \Delta \tag{11}$$

and minimises

$$\sum_{l=1}^{L} F_l^2. \tag{12}$$

Due to condition (**??**), the existence of such a solution is guaranteed, and the minimisation (**??**) makes it unique.

Now we generalise this to the case where we have a global mismatch, which has to be compensated by balancing. Balancing power is needed to cover the residual load after all local resources (Δ_i) and imports/exports ($(K \cdot F)_i$) have been taken into account. For a node n, this can be expressed as[1]

$$B_i = (\Delta_i - (K \cdot F)_i)_- \tag{13}$$

The total balancing is thus

$$B_{\text{tot}} = \sum_{i=1}^{N} (\Delta_i - (K \cdot F)_i)_- = \sum_{i=1}^{N} B_i$$

We use the flows to minimise B_{tot}. In other words, we make the system use as much of the renewable generation as possible. This is our top priority.

Our secondary objective, to minimise dissipation by minimising the square of the flows, is implemented in a second step. Here, we minimise the flows keeping the total balancing at its minimal value found in the first step:

$$\begin{aligned} \min_{F} \quad & \sum_{l=1}^{L} F_l^2 \\ \text{s.t.} \quad & \sum_{i=1}^{N} (\Delta_i - (K \cdot F)_i)_- \leq B_{\min} \end{aligned} \tag{14}$$

This ensures that we arrive at the minimal flows that make optimal sharing of renewables possible.

[1] $(x)_- = \max\{-x, 0\}$ denotes the negative part of a quantity x.

If the line capacities are constrained by (possibly direction dependent) capacities, $f_l^- \leq F_l \leq f_l^+$, these constraints are included in the two minimizations, and we have

$$\begin{aligned}
&\text{Step 1:} && \min_F \quad \sum_{i=1}^{N} (\Delta_i - (K \cdot F)_i)_- && = B_{\min} \\
& && \text{s.t.} \quad f_l^- \leq F_l \leq f_l^+ && \\
&\text{Step 2:} && \min_F \quad \sum_{l=1}^{L} F_l^2 && \\
& && \text{s.t.} \quad f_l^- \leq F_l \leq f_l^+ && \\
& && \quad \sum_{i=1}^{N} (\Delta_i - (K \cdot F)_i)_- \leq B_{\min} &&
\end{aligned} \tag{15}$$

2.1 Energy Storage

Including an Energy Storage System (ESS) into the model simply involves adding a degree of freedom for each node that can provide or absorb power, S, with certain limitations on it's output and input power, s_+ and s_-. These limitations can be used to implement the ESS's power capacities and current state of charge. Assuming that these limitations are known or can be calculated for each time step and for each country, then the residual load can be expressed as

$$B_i = (\Delta_i - (K \cdot F)_i + S_i)_- \tag{16}$$

In minimising this new expression for the residual load, we ensure that the storage S will discharge when needed to reduce the negative mismatch. This assumes that there is some charge inside the storage. To charge the storage with renewables, we wish to add an option to reduce the excess generation, to reduce curtailment. We define the curtailment in the no storage case as

$$C_i = (\Delta_i - (K \cdot F)_i)_+ \tag{17}$$

and, in the storage case, simply as

$$C_i = (\Delta_i - (K \cdot F)_i + S_i)_+ \tag{18}$$

Minimising the curtailment ensures that the storage S will charge when possible to reduce waste of renewables. The optimisation problem would then look as follows

$$
\begin{aligned}
\text{Step 1:} \quad & \min_{F,S} \quad \sum_{i=1}^{N} (\Delta_i - (K \cdot F)_i + S_i)_- && = B_{\min} \\
& \text{s.t.} \quad f_l^- \leq F_l \leq f_l^+ \\
& \qquad s_n^- \leq S_n \leq s_n^+ \\
\text{Step 2:} \quad & \min_{F,S} \quad \sum_{i=1}^{N} (\Delta_i - (K \cdot F)_i + S_i)_+ && = C_{\min} \\
& \text{s.t.} \quad f_l^- \leq F_l \leq f_l^+ \\
& \qquad s_n^- \leq S_n \leq s_n^+ \\
& \qquad \sum_{i=1}^{N} (\Delta_i - (K \cdot F)_i + S_i)_- \leq B_{\min} \\
\text{Step 3:} \quad & \min_{F,S} \quad \sum_{l=1}^{L} F_l^2 \\
& \text{s.t.} \quad f_l^- \leq F_l \leq f_l^+ \\
& \qquad s_n^- \leq S_n \leq s_n^+ \\
& \qquad \sum_{i=1}^{N} (\Delta_i - (K \cdot F)_i + S_i)_- \leq B_{\min} \\
& \qquad \sum_{i=1}^{N} (\Delta_i - (K \cdot F)_i + S_i)_+ \leq C_{\min}
\end{aligned}
\tag{19}
$$

That is, the same as in the no storage case, just with the added secondary objective of reducing curtailed energy. We are convinced[2] that it can be proved – but hasn't yet been so[3] – that the constraint

$$
\sum_{i=1}^{N} (\Delta_i - (K \cdot F)_i + S_i)_- \leq B_{\min} \tag{20}
$$

in Step 2 is never active. That is, that in any given hour, reductions in curtailment never come at an expense for reductions to balancing, and vice versa. Therefore, the order of steps 1 and 2 is irrelevant, and so the problem can be expressed in a more readable structure:

$$
\begin{aligned}
\text{Step 1:} \quad & \min_{F,S} \quad \sum_{i=1}^{N} |\Delta_i - (K \cdot F)_i + S_i| && = \Delta'_{\min} \\
& \text{s.t.} \quad f_l^- \leq F_l \leq f_l^+ \\
& \qquad s_n^- \leq S_n \leq s_n^+ \\
\text{Step 2:} \quad & \min_{F,S} \quad \sum_{l=1}^{L} F_l^2 \\
& \text{s.t.} \quad f_l^- \leq F_l \leq f_l^+ \\
& \qquad s_- \leq S_n \leq s_+ \\
& \qquad \sum_{i=1}^{N} |\Delta_i - (K \cdot F)_i + S_i| \leq \Delta'_{\min}
\end{aligned}
\tag{21}
$$

[2] Rolando and Gorm are convinced

[3] *cough* *cough* Uffe

In matrix notation, where $\min X$ implies a minimisation of the direct internal sum of the elements of X, and $\leq$ implies an elementwise comparison, the problem is

$$\begin{aligned}
\text{Step 1:} \quad & \min_{F,S} \quad |\Delta - (K \cdot F) + S| && = \Delta'_{\min} \\
& \text{s.t.} \quad f_- \leq F \leq f_+ \\
& \qquad\; s_- \leq S \leq s_+ \\
\text{Step 2:} \quad & \min_{F,S} \quad F^2 \\
& \text{s.t.} \quad f_- \leq F \leq f_+ \\
& \qquad\; s_- \leq S \leq s_+ \\
& \qquad\; |\Delta - (KF) + S| \leq \Delta'_{\min}
\end{aligned} \tag{22}$$

Notice also that this is equivalent to the storageless problem, if our assumptions on the "orthogonality" of balancing and curtailment hold and the characteristics of the ESS are set to zero or close to zero.

3 Problem Implementation and Code Structure

A number of software options exist to deal with this kind of problems. Some of them, such as `cvxgen`, can receive problems under this formulation, (eq. 22). For most cases, though, it is better to change this to the standard quadratic programming formulation,

$$\begin{aligned}
\min_{x} \quad & \frac{1}{2} x^T P x + q^T x \\
\text{subject to} \quad & Ax = b \\
& Gx \leq h
\end{aligned} \tag{23}$$

Under this formulation, Step 1 can be expressed as

$$\min_{F,S,X} \quad \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}^T \begin{bmatrix} F \\ S \\ X \end{bmatrix} \equiv \Delta'_{min}$$

$$\begin{aligned}
\text{subject to} \quad & F && \leq && f_+ \\
& -F && \leq && -f_- \\
& S && \leq && s_+ \\
& -S && \leq && -s_- \\
& -KF + S - X && \leq && -\Delta \\
& KF - S - X && \leq && \Delta
\end{aligned} \tag{24}$$

The addition of a new X variable which, as the last two constraints point to, is defined as

$$X \geq |\Delta - KF + S| \tag{25}$$

turns the first step into a purely linear programming problem.

Following the same structure, step 2 can be stated as

$$
\begin{aligned}
\min_{F,S,X} \quad & \tfrac{1}{2} \begin{bmatrix} F \\ S \\ X \end{bmatrix}^T \begin{bmatrix} 2\mathbb{I} & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} F \\ S \\ X \end{bmatrix} \\
\text{subject to} \quad & \begin{aligned} F &\leq f_+ \\ -F &\leq -f_- \\ S &\leq s_+ \\ -S &\leq -s_- \\ -KF + S - X &\leq -\Delta \\ KF - S - X &\leq \Delta \\ X &\leq \Delta'_{min} \end{aligned}
\end{aligned} \tag{26}
$$

with

$$
\begin{aligned}
S, X, \Delta &\in \mathbb{R}^N \\
F &\in \mathbb{R}^L \\
K &\in \mathbb{R}^{N \times L}
\end{aligned}
$$

for all cases

3.1 Code Overview

When we set out to write the scripts that make up our code, our objective was to produce a black box that would receive a graph with N nodes and L links, where each node contained a time series for the mismatch Δ, and which would output a time series for the flow F along each link. This is, essentially, solving the problem stated in (24) and (26). Most of the actual written code is dedicated to receiving the data from a practical storage format, organising it in easily accessible ways, preparing it for the solver and storing it back after the solver has run through the time series. Additionally, each of us has developed their own private utilities, which we use to produce figures, run through different scenarios and make additional calculations.

The core of our code is developed in `python`. We use the interactive shell `ipython`, as it eases the process of playing with results and testing new things. If you have a good reason not to use `python`, other languages can more or less easily interact with the solver structure, though we can offer very little guidance in that matter.

At the centre of the software is the `Nodes` class, which is a collection of `node` objects and some tools to operate on them. The `node` object is essentially a box with information on a given country or region, such as a load time series, potential wind and solar time series, as well as α and γ parameters (see eq. (1)). By default, within each `node`, an `ESS` object (Energy Storage System) is initialised, though this can be set to very small – but not arbitrarily small – values to rule out its influence on the solution. Note, however, that the `Nodes` class holds no information of the properties of the graph itself, that is, no information on the links.

The network structure must be stored within a human-readable text file. It should contain a tab-separated table with link capacities between two nodes, where the value of (i, j) is the transmission capacity from node i to node j in MW. The diagonal is not read and can be set to any value, but for human readability is currently used to show the sum of outgoing capacities. From this table, the K matrix with network structure can be extracted, as well as the flow capacities f_+ and f_-. All of this is performed by the function `AtoKh()`, which returns the K matrix in a decomposed, non-human readable format for sparse matrix composition, the link capacities in a single vector, and a list of the links with their start and end points and their indices.

By looking at equations (24) and (26), we see that we have essentially all the information we need to solve the problem, apart from the mismatch Δ for each country. The main function in the software, `sdcpf()`, takes a `Nodes` object of any size – as long as it has an appropriately dimensioned network structure text file – calculates a Δ time series and runs through the time series of the nodes contained therein. The output of this function is a 'solved' `Nodes` object, which now contains additional information on the storage power and on the balancing and curtailment that each node performs, and an F matrix with the flows at all links at all times.

3.2 Code – Line by Line

A number of `python` modules is required by `AURES`: `numpy`, `pylab`, `scipy`, which you should have anyway; `sqlalchemy`, if you wish to download data form our server: and, most importantly, `gurobipy`, which is the python extension of the solver tool `gurobi`. This last one can only be obtained from the manufacturer's website, and requires an academic license to operate. Additionally, it's a good idea to have `matplotlib`, for all your plotting needs.

A working installation of `AURES` should contain at least two files: `aures.py` and `auresc.py`, and two folders: `data` and `settings`. If any of these folders does not exist after you fork/install/steal the code, you should create them. The `data` folder should contain `*.npz` files, which are uncompressed `numpy` files that contain several arrays, one file per node. `AURES` expects your arrays to be called `L`, `Gw`, `Gs` and `datalabel`, and to contain, respectively, the un-normalised load time series, the normalised wind time series, the normalised solar time series and a name for the node. The `settings` folder should contain the network capacity text file, as described above. In what follows, it is useful to have a copy of the code at hand.

3.2.1 File `auresc.py`

The `auresc.py` file contains the classes for `AURES`. We start by defining the `node` object, which reads the data files described just above. It receives a path and filename for the datafile and an ID number in `int` format. It stores the load in `load` and the normalised wind and solar time series in `normwind` and `normsolar`, respectively. It is only in the normalised states that this information is stored, as the parameters `alpha` and `gamma` are used to calculate the `mismatch`. Use the commands `get_wind()` and `get_solar()` to query the un-normalised wind and solar time series. Furthermore, you should not modify `alpha` and `gamma` directly, instead use the functions `set_alpha()` and `set_gamma()`, as the `mismatch` won't be recalculated automatically otherwise.

Usage Examples:

```
my_node=node(path="./data", filename="data1.npz",0)
print my_node.label, my_node.ID
print my_node.load
print my_node.get_wind()
print my_node.mismatch
my_node.set_gamma(0.75)
my_node.set_alpha(0.95)
print my_node.mismatch
```

Within each node object, an instance of the `ESS` class is initialised. This small object is used to keep track of the State of Charge of an energy storage system, and to identify the limits in power output and input that the storage can accommodate and deliver. Note that the `SOC` field is defined as being one time step longer than the rest of the time series in the code, as we require knowledge of the charge at the beginning and end of each hour. The `SOC` of the ESS, together with the maximum and minimum power `pmax` and `pmin` can be used to determine what

the adequate limits s_+ and s_- should be in the problem formulation. However, none of this is calculated inside the ESS object, and the only function inside is the `update()` command, which uses the power output or input to determine the change in the `SOC`.

The last and most important class is `Nodes`. This is simply a collection of `node` objects and a set of tools to perform operations on them. The simple usage of objects within `python` enables quick queries of all the `node`s' properties. A list of files is provided to the `Nodes` object (though it might be easier to change the default in the script, for working on specific cases), and then a cache of `node`s with those files is built. The `Nodes` object then arranges the `node`s in a list with ID indices starting at zero. It also receives a path to the network table and stores it in `admat`. A number of tools for manipulating the values of the nodes and updating them as the time series progresses is also provided. For example, `get_Ppos(t)` and `get_Pneg(t)` return the possible output and input to the individual `node`s' ESS at time t. It is also possible to change the values for `alpha`, `gamma` and `ESS` to all the nodes contained in `Nodes` by using the corresponding functions. A last important function is the `save_nodes()` function, which writes an uncompressed `*.npz` file containing the whole `Nodes` object (in contrast to the files in `data`, which have contain a single `node` instance). These can be reloaded when calling defining a new instance of `Nodes` by using the parameter `load_filename`. This is especially useful when solving a wide range of scenarios and wanting to go back to a previously 'solved' problem.

Usage Examples:

```
Europe=Nodes(files=["data1.npz", "data2.npz", "data3.npz"...])
Europe[0].set_gamma(0.75)
for n in Europe:
  n.set_gamma(0.75)
Europe.set_gammas(0.75)
for n in Europe:
  print n.label, n.id, n.gamma
Europe.save_nodes("Backup")
New_Europe=Nodes(load_filename="Backup.npz")
```

3.2.2 File `aures.py`

Though its just under 300 lines long, this is the central file of the software. It starts with the definition of two auxiliary functions, `AtoKh()` and `get_quant()`. The first of this uses the defined admittance matrix, as defined by the network capacity in the text file, to extract the incidence matrix K and the upper and lower limits for transmission along links, f_+ and f_- (these were previously referred to as h). The incidence matrix K is expressed in the disassembled vectors of row and column indices and values, `K_row`, `K_col` and `K_val`, respectively. The fourth returned variable is a vector `h` composed of interchanging values of f_+ and f_-, for each link.

The other auxiliary function, `get_quant()`, reads data from a previously existing run. One can define the source file with the parameter `filename`, though it points by default to a file called `copper_flows.npy`, stored in the `results` folder. This file should simply be a previously saved vector of flows. The function builds a cumulative histogram for the flow at each link, and finds the quantile defined by the input parameter `quant`. It returns a large vector of the same size as `h` in `AtoKh()`, both of which can immediately be used in defining new transmission capacities.

Two important functions finish the file: `sdcpf()`[4] , which manages the time series and stores results at every timestep, and `solve_flows`, which actually finds the optimal values witht he help of `gurobi`. The function `sdcpf()`receives a `Nodes` object `N` and several optional

[4]A newer version of the old `zdcpf()`. Historically named thus for 'Sarah DC Power Flow'. Sorry, Sarah, but at some point we should start calling it something fancier. Maybe just `solve()` or `aures()`

parameters. If `copper` is set to 1 or `True`, then transmission capacities are set to very high values. Please, please, please note that they are not ignored, and make sure that the value they are set to (`1e6`) is not too small for your calculations. The parameter `lapse` defines the length of timesteps the solver should run. It is useful for debugging, as smaller runs take considerable less time. If ignored, it defaults to the total timesteps in the first `node` of `N`. The parameters `b` and `h0` can be used to modify the transmission capacities. A new distribution can be set by `h0`, after, for example, obtaining one from `get_quant()`. This new distribution, or the default one, if `h0` is ignored, can be linearly increased or reduced by the parameter `b`.

The first part of the function, helpfully labeled 'Loading Default Values' loads the default values, and assigns new ones if they have been defined in the parameters. This is followed by 'Preparing vars for solver', which basicaly creates two empty variables for receiving the solution at each time step (`flw` and `sto`), and two larger arrays to accumulate the solutions of all timesteps (`F` and `S`). Next, in 'Setting up the model', the problem statement as per (24) and (26) is defined for `gurobi`. This implies the explicit creation of all variables, upper and lower bounds, and constraints. The upper and lower bounds are essentially the first four constraints in both problems, but are treated differently by `gurobi`. A special `gurobi.Model()` object is required for this. The K matrix, the first four constraints, and all the variables are added by name to `network`, which is the `gurobi` object. Objective functions and the last constraints of each problem are left for the solver function to define. Next comes the section 'Run the time series', where the mismatch is calculated at each timestep and sent, together with the `network` object, to the solver function `solve_flows`. Results are received by `flw` and `sto` and then stored in `F` and `S`. This could be done more efficiently, but I think this is more human readable. A small `,` is printed sequentially during the calculation[5] and a timer counter shows the elapsed time at the end of it. The final part of the function, 'Assignment to Objects', collects the `F` and `S` and uses them to calculate actual balancing, curtailment, and saving them to the object `N`. The function returns the nodes `N` and the flows `F`.

Finally, the function `solve_flows()` takes care of actually using the `gurobi` solver to get the minimum from problem (24), and then reformulating the problem to problem (26). The first part of the function builds up the variable list, in order to 'know' where to add constraints, etc. Then it defined the cost function as the simple linear sum of all X. After the step has converged, the minimum balancing is stored and a slack variable of size `1.0` is added to help the second step converge. The constraints according to (26) are added, and the new qudratic objective function is defined.

4 Super Short User's Guide

As mentioned before, a working installation of `AURES` should contain at least two files: `aures.py` and `auresc.py`, and two folders: `data` and `settings`. For the case of the 27-node European network, you need to download the data files from the group's server, Pepsi. This data is currently only available to group collaborators. Alternatively, your own data can be used if it is organised as previously described. Additionally to the main `AURES` files, a script originally by Gorm Andresen, `datamgmt.py`, fetches de data files and saves them in the correct format. First thing to do is to open a connection to Pepsi by inputting the following command in your terminal:

```
ssh -L5432:localhost:5432 USERNAME@pepsi.imf.au.dk
```

Then, in another terminal window, simply run the `datamgmt.py` script directly by typing

```
python datamgmt.py
```

[5] Actualy, every 2073 timesteps, so that 33 comas will be drawn over 70128 timesteps. For more information as to why, ask Sarah

or, in `ipython`

```
%run datamgmt.py
```

This script will attempt to access the file for Denmark[6], and failing to find it in `./data/DK.npz` will download all the data files to that folder. I imagine a similar script should be made by someone[7] to get data for whatever new regions we are going to work on in the future. After the data has been downloaded, the simplest case can be run in `ipython` by typing

```
%run aures.py
N=Nodes()
N,F=sdcpf(N,copper=1)
N.save_nodes("copper")
np.save("./results/copper_flows",F)
```

These lines load the functions from `aures.py`, then generate a instance of `Nodes()`. This instance is then sent to `sdcpf()`for the unconstrained power flow (`copper=1`). This takes around 240 s. We save the nodes and then the flows. It is important to run an unconstrained flow at some point, as the file `opper_flows.npy` is needed by some functions.

I now explain how to make a simple sweep with different transmission sizes. Let's say you want to calculate the power flow for the 99% quantiles and several fractions of those. you can then type

```
X=[0.10, 0.25, 0.50, 0.75, 0.90, 1.0]
H=get_quant(quant=0.99)
for x in X:
  N,F=sdcpf(N,h0=H,q=x)
  N.save_nodes("Sweep_"+str(x))
  np.save("./results/Sweep_"+str(x)+"_flows",F)
```

This will not only cycle through the fractions in `X`, but also save a copy of all the nodes and flows as you are done calculating them. You can afterwards open any of the files, for further analysis, by typing

```
M=Nodes(load_filename("Sweep_0.75.npz"))
G=np.load("./results/Sweep_0.25_flows.npy")
```

And that is, essentially, all that you need to know!

For doubts, questions, comments, complaints and threats, email me @ `rar@imf.au.dk`.

[6]Thanks to the dano-centric script author
[7]*cough* *cough* Timo