DFNSolvers: Algorithmic resolution of the Dissipative Flow Network Problem

Robin Delabays

Center for Control, Dynamical Systems, and Computation,
UC Santa Barbara,
Santa Barbara, CA-93106, USA.

robindelabays@ucsb.edu

December 18, 2021

Contents

We provide the theoretical grounds for the algorithms resolving the Dissipative Flow Network Prolem, both on acyclic and cyclic graphs, as well as a description of the relevant routines. A illustrating example is also discussed.

The source code for the algorithm and the toolbox of routines can be found on the online repository: `https://github.com/DissipativeNetworkFlows/DFNSolvers`. The algorithm is a companion to Ref. [1].

1 System requirements

The code has been developed under **Julia 1.6** [2] and should work with any version older than 1.0. The following Julia packages are needed:

- `DelimitedFiles`
- `LinearAlgebra`
- `SparseArrays`
- `Statistics`

2 The Dissipative Flow Network Problem

Let us define the Dissipative Flow Network (DFN) problem [1]. Given an undirected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, let us denote the (i,j)th component of its adjacency matrix by $a_{ij} \in \{0,1\}$. Let associate to each edge $\{i,j\}$ of $\mathcal{G}$ a pair of coupling functions $h_{ij}, h_{ji} \colon \mathbb{R} \to \mathbb{R}$, one for each orientation. We aim at solving

$$\varphi = \omega_i - \sum_{j=1}^{n} a_{ij} h_{ij}(\theta_i - \theta_j)\,, \qquad\qquad i \in \{1, ..., n\}\,, \tag{1}$$

in the variables $\theta_1, ..., \theta_n$, and φ. In Eq. (3), $\omega_i \in \mathbb{R}$ is the natural frequency of oscillator i, which can be interpreted as a commodity injection or withdrawal. The coupling functions h_{ij} are assumed continuous, 2π-periodic, and $h_{ij}(0) = 0$. Furthermore, we assume that they are strictly increasing in a neighborhood $[\gamma_{ij}^-, \gamma_{ij}^+]$ of the origin, $\gamma_{ij}^- < 0 < \gamma_{ij}^+$. We emphasize that we do not require any relation between h_{ij} and h_{ji}. From now on, we will index the edge-dependent object with edge indices, e.g., if $e = (i,j)$, then $h_e = h_{ij}$.

To put the problem in a more tractable form, we define the proxy *flow variable*

$$f_e = h_{ij}(\theta_i - \theta_j)\,, \qquad\qquad e = (i,j)\,. \tag{2}$$

Strict monotonicity of the coupling functions implies the they are invertible. Therefore angular differences can be unequivocally recovered from the flow variables. Eq. (1) can be reformulated as the DFN problem

$$\varphi = \omega_i - \sum_{\substack{e:\\ s_e = i}} f_e\,, \qquad\qquad i \in \{1, ..., n\}\,, \tag{3a}$$

$$h_e^- = h_e(\gamma_e^-) \leq f_e \leq h_e(\gamma_e^+) = h_e^+\,, \qquad\qquad e \in \mathcal{E}\,, \tag{3b}$$

to be solved in $\boldsymbol{f} \in \mathbb{R}^{2m}$ and $\varphi \in \mathbb{R}$.

We refer to Ref. [1] for a extended discussion of this problem.

3 The acyclic graph algorithm

The case of an acyclic graph $\mathcal{G}$ is completely characterized algorithmically. We discuss this case here and provide the associated algorithm.

3.1 Theoretical background

It has been shown in Ref. [1] that, if the interaction graph is acyclic, then the DFN problem has at most a unique solution. It is possible to decide algorithmically whether a given DFN problem on an acyclic graph has a solution. We first explain how this algorithm works and then formulate it. At each step, we indicate in square brackets the corresponding lines of the algorithm.

Suppose we are given a DFN problem Eq. (3) associated to an acyclic graph $\mathcal{G}$. Without loss of generality, let us renumber the nodes such that node 1 is a leaf, and such that, for all $i \in \{2, ..., n-1\}$, node i is a leaf of the graph $\mathcal{G}$ where nodes with index up to $i-1$ are pruned.

In our algorithm, we will need to distiguish the two orientation of each edge. Let us then denote by $e_i \in \mathcal{E}_\mathrm{b}$ the (directed) edge that connects node $i \in \{1, ..., n-1\}$ to the set of nodes $\{i+1, ..., n\}$. This edge is unique by the way we defined the node numbering. We denote by e_{i+m} the reversed edge $\bar{e}_i$, so all directed edges are indexed.

For convenience, we define the *reciprocal flow funcions*,

$$H_e \colon [h_{\bar{e}}(\gamma_{\bar{e}}^-), h_{\bar{e}}(\gamma_{\bar{e}}^+)] \to [h_e(\gamma_e^-), h_e(\gamma_e^+)] \tag{4}$$

$$f \mapsto h_e[-h_{\bar{e}}^{-1}(f)]\,, \tag{5}$$

which relates the flow on an edge $(\bar{e})$ to the flow on the same edge with opposite orientation (e).

3.2 The algorithm

We now recursively define a sequence of problems that are all equivalent to Eq. (3). Our construction will emphasize conditions that allow to decide if the problem can be solved or not. The initial problem is precisely Eq. (3), which we denote by (P0), that needs to be solved for $\boldsymbol{f} \in \mathbb{R}^{2m}$ and $\varphi \in \mathbb{R}$.

Initial step, $i = 1$. Following our choice of numbering, Eq. (3a) yields

$$f_{e_1} = \omega_1 - \varphi\,, \tag{6}$$

and the bounds of Eq. (3b) require [1-2]

$$\varphi \in [\omega_1 - h_{e_1}^+, \omega_1 - h_{e_1}^-] =: [\ell_1, u_1]\,. \tag{7}$$

Defining the functions $F_{e_1}(\varphi) = \omega_1 - \varphi$ and $F_{\bar{e}_1}(\varphi) = H_{\bar{e}_1}[F_{e_1}(\varphi)]$ [3-4], we can express the relations between f_{e_1}, $f_{\bar{e}_i}$, and φ,

$$f_{e_1} = F_{e_1}(\varphi)\,, \qquad\qquad f_{\bar{e}_1} = F_{\bar{e}_1}(\varphi)\,. \tag{8}$$

Note that F_{e_1} (resp. $F_{\bar{e}_1}$) is striclty decreasing (resp. increasing) in φ. This allows to define the following problem, to be solved for $\boldsymbol{f} \in \mathbb{R}^{2m}$ and $\varphi \in \mathbb{R}$,

$$\begin{aligned}
&\varphi = \omega_j - \sum_{\substack{e\colon s_e = j \\ t_e = 1}} F_e(\varphi) - \sum_{\substack{e\colon s_e = j \\ t_e > 1}} f_e\,, \quad j \in \{1, ..., n\} \\
&f_{e_1} = F_{e_1}(\varphi)\,, \\
&f_{\bar{e}_1} = F_{\bar{e}_1}(\varphi)\,, \\
&\text{st.} \quad \varphi \in [\ell_1, u_1]\,, \\
&\qquad f_e \in [h_e^-, h_e^+]\,, \qquad e \in \{e_2, ..., e_m, \bar{e}_2, ..., \bar{e}_m\}\,, \\
&\qquad f_{\bar{e}} = H_{\bar{e}}(f_e)\,, \qquad e \in \{e_2, ..., e_m, \bar{e}_2, ..., \bar{e}_m\}\,,
\end{aligned} \tag{P1}$$

which is equivalent to (P0).

At this point, if $\ell_1 > u_1$, no solution of (P1) can be found. Otherwise we continue to the next step.

Recursion step, $2 \le i \le n-1$. Up to this point, we have defined a problem (P$(i-1)$), bounds $\ell_{i-1} \le u_{i-1}$, and a sequence of striclty decreasing (resp. strictly increasing) functions F_{e_j} (resp. $F_{\bar{e}_j}$) for $j \in \{1, ..., i-1\}$. By our choice of numbering, node i has only one neighbor with index larger than i. The i-th equation of problem (P$(i-1)$) then yields [6]

$$f_{e_i} = \omega_i - \sum_{\substack{e\colon s_e = i \\ t_e < i}} F_e(\varphi) - \varphi =: F_{e_i}(\varphi)\,, \tag{9}$$

whose components have been explicitly defined by the previous steps and that is strictly decreasing. We also define $F_{\bar{e}_i} = H_{\bar{e}_i} \circ F_{e_i}$ [21]. Furthermore, in order to satisfy Eq. (3b), we may need to restrict the allowed values of φ to a subinterval $[\ell_i, u_i] \subseteq [\ell_{i-1}, u_{i-1}]$, such that for any $\varphi \in [\ell_i, u_i]$, $h^-_{e_i} \leq F_{e_i}(\varphi) \leq h^+_{e_i}$. In details [7],

I. For the lower bound ℓ_i:

(i) If $F_{e_i}(\ell_{i-1}) < h^-_{e_i}$, then, by monotonicity of F_{e_i}, Eq. (3b) cannot be satisfied and there is no solution to Eq. (3) ;

(ii) If $h^-_{e_i} \leq F_{e_i}(\ell_{i-1}) \leq h^+_{e_i}$, then $\ell_i = \ell_{i-1}$;

(iii) If $F_{e_i}(\ell_{i-1}) > h^+_{e_i}$, then ℓ_i is the unique solution of $F_{e_i}(\ell_i) = h^+_{e_i}$ in the interval $[\ell_{i-1}, u_{i-1}]$, if it exists;

II. For the upper bound u_i:

(i) If $F_{e_i}(u_{i-1}) > h^+_{e_i}$, then, by monotonicity of F_{e_i}, Eq. (3b) cannot be satisfied and there is no solution to Eq. (3) ;

(ii) If $h^-_{e_i} \leq F_{e_i}(u_{i-1}) \leq h^+_{e_i}$, then $u_i = u_{i-1}$;

(iii) If $F_{e_i}(u_{i-1}) < h^-_{e_i}$, then u_i is the unique solution of $F_{e_i}(u_i) = h^-_{e_i}$ in the interval $[\ell_{i-1}, u_{i-1}]$, if it exists.

If we are in case either (I.i) or (II.i), there is no solution to Eq. (3). Note that, in case (I.iii), if there is not solution of the equation in the desired interval, this implies that we are simultaneously in case (II.i), and there is then no solution to our problem [and vice versa with cases (II.iii) and (I.i)]. Otherwise, we have defined a new interval $[\ell_i, u_i] \subset [\ell_{i-1}, u_{i-1}]$ for φ. We now define the problem, wich ought to be solved in $\boldsymbol{f} \in \mathbb{R}^{2m}$ and $\varphi \in \mathbb{R}$,

$$
\begin{aligned}
&\varphi = \omega_j - \sum_{\substack{e\colon s_e=j \\ t_e\leq i}} F_e(\varphi) - \sum_{\substack{e\colon s_e=j \\ t_e> i}} f_e \,, \quad j \in \{1, ..., n\} \\
&f_e = F_e(\varphi)\,, \qquad e \in \{e_1, ..., e_i\} \\
&f_{\bar{e}} = F_{\bar{e}}(\varphi)\,, \qquad e \in \{e_1, ..., e_i\} \\
&\text{st.} \quad \varphi \in [\ell_i, u_i]\,, \\
&\qquad f_e \in [h^-_e, h^+_e]\,, \qquad e \in \{e_{i+1}, ..., e_m, \bar{e}_{i+1}, ..., \bar{e}_m\}\,, \\
&\qquad f_{\bar{e}} = H_{\bar{e}}(f_e)\,, \qquad e \in \{e_{i+1}, ..., e_m, \bar{e}_{i+1}, ..., \bar{e}_m\}\,,
\end{aligned}
\tag{Pi}
$$

which is equivalent to all the previous versions of the problem. Again, remark that F_{e_i} (resp. $F_{\bar{e}_i}$) is strictly decreasing (resp. strictly increasing).

Final step, $i = n$. The n-th equation of (P$(n-1)$) yields [22]

$$
\omega_n = \sum_{e\colon s_e=n} F_e(\varphi) + \varphi\,, \tag{10}
$$

whose right-hand-side is striclty increasing. If Eq. (10) has no solution in $[\ell_{n-1}, u_{n-1}]$, then Eq. (3) has no solution [26]. If Eq. (10) has a solution φ^* in $[\ell_{n-1}, u_{n-1}]$, then problem (P$(n-1)$) can be solved as well as all the previous equivalent problems, and in particular Eq. (3), with [24, 28]

$$
\varphi = \varphi^*\,, \tag{11}
$$

$$
f_e = F_e(\varphi^*)\,, \qquad e \in \mathcal{E}_\mathrm{b}\,, \tag{12}
$$

which concludes the algorithm.

Remark. *We note that each time we need to determine if a solution exists in a given interval, by monotonicity of each member of the equation, it suffice to evaluate these members at both ends of the interval of interest to decide of a solution exists in it. If so, we solve the equation, otherwise we can stop the algorithm.*

As we defined a sequence of equivalent problems, the at most uniqueness of the solution of the last one (P$(n-1)$), implies the at most uniqueness of all the others, and in particular at most uniqueness of the solution of Eqs. (3).

Input: A DFN Problem $(\mathcal{G}_\mathrm{b}, \{h_e\}_{e\in\mathcal{E}_\mathrm{d}}, \boldsymbol{\omega}, \boldsymbol{\gamma}^-, \boldsymbol{\gamma}^+)$ with nodes and edges numbered as in [1].

Output: Either "`NO SOLUTION`" or the unique solution $(\boldsymbol{f}^*, \varphi^*)$.

$\ell_1 \longleftarrow \omega_1 - h^+_{e_1}$
$u_1 \longleftarrow \omega_1 - h^-_{e_1}$
$F_{e_1} \longleftarrow (\varphi \mapsto \omega_1 - \varphi)$
$F_{e_{1+m}} \longleftarrow \left(\varphi \mapsto H_{e_{1+m}}[F_{e_1}(\varphi)]\right)$
for $i \in \{2, ..., n-1\}$ **do**
  $F_{e_i} \leftarrow \left(\varphi \mapsto \omega_i - \sum_{e:\ s(e)=i, t(e)<i} F_e(\varphi) - \varphi\right)$
  switch $F_{e_i}(\ell_{i-1})$ *and* $F_{e_i}(u_{i-1})$ **do**
    case $F_{e_i}(\ell_{i-1}) > h^+_{e_i}$ **and** $F_{e_i}(u_{i-1}) < h^-_{e_i}$ **do**
      $\ell_i \leftarrow \ell_{i-1}$
      $u_i \leftarrow u_{i-1}$
    case $h^-_{e_i} \le F_{e_i}(\ell_{i-1}) \le h^+_{e_i}$ **and** $F_{e_i}(u_{i-1}) < h^-_{e_i}$ **do**
      $\ell_i \leftarrow \ell_{i-1}$
      $u_i \leftarrow F^{-1}_{e_i}(h^-_{e_i})$
    case $F_{e_i}(\ell_{i-1}) > h^+_{e_i}$ **and** $h^-_{e_i}1 \le F_{e_i}(u_{i-1}) \le h^+_{e_i}$ **do**
      $\ell_i \leftarrow F^{-1}_{e_i}(h^+_{e_i})$
      $u_i \leftarrow u_{i-1}$
    case $h^-_{e_i} \le F_{e_i}(\ell_{i-1}) \le h^+_{e_i}$ **and** $h^-_{e_i}1 \le F_{e_i}(u_{i-1}) \le h^+_{e_i}$ **do**
      $\ell_i \leftarrow F^{-1}_{e_i}(h^+_{e_i})$
      $u_i \leftarrow F^{-1}_{e_i}(h^-_{e_i})$
    otherwise do return `NO SOLUTION` ;
  $F_{e_{i+m}} \longleftarrow \left(\varphi \mapsto H_{e_{i+m}}[F_{e_i}(\varphi)]\right)$
$F_n \longleftarrow \left(\varphi \mapsto \sum_{e:\ s(e)=n} F_e(\varphi) + \varphi\right)$
if $F_n(\ell_{n-1}) > \omega_n$ **or** $F_n(u_{n-1}) < \omega_n$ **then**
  $\varphi^* \longleftarrow F_n^{-1}(\omega_n)$
else
  return `NO SOLUTION`
for $e \in \mathcal{E}_\mathrm{d}$ **do**
  $f^*_e \longleftarrow F_e(\varphi*)$
return $\boldsymbol{f}^*$, φ^*

3.3 Running the acyclic algorithm

The algorithm is ran by the routine `run_acyclic_algorithm`, which itself calls the actual iteration `acyclic_algorithm`. Both routines are found in the file `acyclic_algorithm.jl`.

3.3.1 `run_acyclic_algorithm`

Prepares the arguments to be appropriate for the actual algorithm and checks for errors in the inputs. Namely, for n and m the number of vertices and of (undirected) edges respectively, it verifies that:

- The graph is acyclic and connected, i.e., $m = n - 1$;
- There are n natural frequencies, i.e., $\boldsymbol{\omega}$ has dimension n;
- There are $2m$ coupling functions.

Then it re-indexes the node and edge indices such that the assumptions of [1, Theorem 2] are satisfied, runs the algorithm (`acyclic_algorithm`), and retieves the original indexing of nodes and edges.

The unique solution of the Asymmetric Flow Network problem is returned, if it exists.

Inputs:

`B::Union{Matrix{Float64}, SparseMatrixCSC{Float64,Int64}}` : Incidence matrix of the (undirected) interaction network. It can be dense or sparse.

`ω::Vector{Float64}` : Vector of natural frequencies at each node.

`h::Union{Tuple{Function,Function},Vector{Tuple{Function,Function}}}` : Vector of 2-tuples composed of the coupling functions (1st component) and their inverse (2nd component). If a single 2-tuple is given, it is assumed that coupling are homogeneous and the same coupling is taken over each (directed) edge.

`γ::Union{Tuple{Float64,Float64}, Vector{Tuple{Float64,Float64}´}}` : Vector of 2-tuples composed of the low (1st component) and upper (2nd component) bounds of the domain of each coupling function, guaranteeing them to be stricly increasing. If a single 2-tuple is given, it is assumed that coupling are homogeneous and the same bounds are taken over each (directed) edge.

Outputs:

`exist::Bool` : Boolean variable that is `true` if and only if the solution exists, `false` otherwise.

`θ::Vector{Float64}` : Vector of angles corresponding to the unique solution. Recall that angles are defined up to a constant shift.

`ff::Vector{Float64}` : Vector of flows on the (directed) edges of the graph for the solution, i.e., it is a vector with $2m$ componentswill. Indices $\{1, ..., m\}$ correspond to the edges with orientation matching the definition of the incidence matrix `B`. Reversed edges have indices $\{m + 1, ..., 2m\}$.

`φ::Float64` : Synchronous frequency of the solution.

3.3.2 `acyclic_algorithm`

This is the actual implementation of the algorithm described in Sec. 3.2.

Inputs:

`B::Union{Matrix{Float64}, SparseMatrixCSC{Float64,Int64}}` : Incidence matrix of the (undirected) interaction network. It can be dense or sparse, and components are `Float64`'s. The node and edge indexing needs to satisfy the assumption of [1, Theorem 2].

`ω::Vector{Float64}` : Vector of natural frequencies at each node.

`H::Vector{Function}` : Vector of the reciprocal flow functions, defined in Eq. (4). This a $2m$-vector.

`hγ::Vector{Float64}` : Vector of 2-tuples composed of the low (1st component) and upper (2nd component) bounds on the values taken by the coupling functions over their domain. Namely, the first (resp. second) component of `hγ[e]` is $h_e(\gamma_e^-)$ [resp. $h_e(\gamma_e^+)$].

Outputs:

`exist::Bool` : Boolean variable that is `true` if and only if the solution exists, `false` otherwise.

`ff::Vector{Float64}` : Vector of flows on the (directed) edges of the graph for the solution, i.e., it is a vector with $2m$ components. Indices $\{1, ..., m\}$ correspond to the edges with orientation matching the definition of the incidence matrix `B`. Reversed edges have indices $\{m + 1, ..., 2m\}$.

`φ::Float64` : Synchronous frequency of the solution.

---

4 The general, cyclic graph iterations

Solutions of the DFN problem for a general, cyclic graph $\mathcal{G}$ is characterized, under technical assumptions. The results presented in Ref. [1] rely on an iteration map, that we implement here.

4.1 Theoretical background

Given an undirected graph $\mathcal{G}$, with incidence matrix B, we define its *bidirected counterpart* as the directed graph with same set of nodes and with each edge doubled, one with each orientation. The incidence matrix of the bidirected graph is $B_\mathrm{b} = [B \ -B]$, and we define its *out-incidence matrix* B_o as the positive part of B_b.

Now, using the angular difference variable $\boldsymbol{\Delta} \in \mathbb{R}^m$, such that $\boldsymbol{f} = \boldsymbol{h}(\boldsymbol{\Delta})$, Eq. 3 can be phrased in vectorial form

$$\varphi \mathbf{1}_n = \boldsymbol{\omega} - B_\mathrm{o} \boldsymbol{h}\left(\boldsymbol{\Delta}\right) \tag{13}$$

$$\boldsymbol{\Delta} \in \prod_e [\gamma_e^-, \gamma_e^+] \,. \tag{14}$$

It is shown in [1], that there is at most one solution to Eq. (13) in each winding cell associated to the interaction graph (we refer to Refs. [3, 1] for an extended discussion about winding numbers, cells, and partitions). This is shown by proving that winding cells are invariant under the iteration map

$$S_{\boldsymbol{\omega}} \tag{15}$$

and that $S_{\boldsymbol{\omega}}$ is contracting. By iterating $S_{\boldsymbol{\omega}}$ from initial conditions with winding vector $\mathbf{u}$, one reaches a fixed point, which is solution of Eq. (13), with the same winding vector $\mathbf{u}$.

4.2 Running the cyclic graph iterations

The iterations are run through the routine `iterations`.

4.2.1 `iterations`

Prepares the argument to be sent to the map $S_{\boldsymbol{\omega}}$ and runs the iterations.

Inputs:

`Δ0::Vector{Float64}` **:** Initial conditions of the iterations. Each components should ideally be bounded by the components of γ.

`B::Matrix{Float64}` **:** Incidence matrix of the (undirected) graph.

`C::Matrix{Float64}` **:** Cycle-edge incidence matrix associated to the cycle basis of the graph (see [1]).

`u::Vector{Int64}` **:** Winding vector of the winding cell where the solution is looked for.

`ω::Vector{Float64}` **:** Vector of natural frequencies.

`h::Union{Function,Vector{Function}}` Vector of coupling functions over the (bidirected) edges of the graph. If a single function is given, the couplings are assumed homogeneous.

`γ::Union{Tuple{Float64,Float64},Vector{Tuple{Float64,Float64}}}` **:** Vector of tuples, whose components are the lower (1st comp.) and upper (2nd comp.) bounds on the domains of the coupling functions. Should have the same dimension as `h`.

`δ::Float64` **:** Scaling parameter which, if sufficiently small, ensures contractivity of the iteration.

`s::Union{Float64,Vector{Float64}}` : Vector of the slopes of the extended coupling functions (see [1]). If a single value is given, all couplings are extended with the same slope.

`max_iter::Int64=100` : Maximal number of iterations.

`tol::Float64=1e-6` : Minimal correction allowed between two iterations.

`verb::Bool=false` : If `true`, enumerates the iterations.

Output:

`Δ::Vector{Float64}` : Final state at the end of the iterations.

`Δs::Matrix{Float64}` : Sequence of states along the iterations.

4.2.2 `Sδ`

Iteration function whose fixed points are solutions of the DFN problem [1].

Inputs:

`Δ::Vector{Float64}` : Argument of the iteration function.

`ω::Vector{Float64}` : Vector of natural frequencies.

`B::Matrix{Float64}` : Incidence matrix of the (undirected) graph.

`Bout::Matrix{Float64}` : Out-incidence matrix of the bidirected graph.

`P::Matrix{Float64}` : Cycle projection matrix (see [3, 1]).

`W::Matrix{Float64}` : Weight matrix to be tuned. In Ref. [1], it is taken as the pseudoinverse of the Laplacian matrix of the graph.

`δ::Float64` : Scaling parameter which, if sufficiently small, ensures contractivity of the iteration.

`h::Union{Function,Vector{Function}}` Vector of coupling functions over the (bidirected) edges of the graph. If a single function is given, the couplings are assumed homogeneous.

`γ::Union{Tuple{Float64,Float64},Vector{Tuple{Float64,Float64}}}` : Vector of tuples, whose components are the lower (1st comp.) and upper (2nd comp.) bounds on the domains of the coupling functions. Should have the same dimension as `h`.

`s::Union{Float64,Vector{Float64}}` : Vector of the slopes of the extended coupling functions (see [1]). If a single value is given, all couplings are extended with the same slope.

Output:

`Δ2::Vector{Float64}` : Updated value of the state Δ.

4.2.3 `hs`

Extended coupling function, extendedin `h` to the whole real axis. Matches `h` on `[γ[1],γ[2]]`, is continuous, and has slope `s` outside of `[γ[1],γ[2]]`. Each input can be given as a vector, in which case, `h` is applied elementwise to each argument.

Inputs:

`x::Union{Float64,Vector{Float64}}` : Argument of the extended coupling function. If a vector is given, the funtion is evaluated elementwise.

`h::Function` : Initial coupling function to be extended.

`γ::Tuple{Float64,Float64}` : Tuple of the two bounds of the domain of `h`.

`s::Float64` : Slope of the extended coupling function outside of the domain of `h`.

Output:

`hx` : Value of the extended coupling function at `x`.

---

5 Illustrative example

The script `examples_rts96.jl` illustrates how the codes `acyclic_algorithm` and `iterations` are run. It is composed of two parts:

1. Resolution of the Dissipative Flow Network problem on an arbitrarily chosen spanning tree of the IEEE RTS-96 test case [4], illustrated in the right panel of Fig. 1. Parameters are loaded from the network admittance matrix through `load_ksakaguchi`. Natural frequencies are taken as the nodal balance of load and generation at each bus in the test case.

2. Application of the iteration alogrithm on the network structure of the IEEE RTS-96 test case [4], illustrated in the left panel of Fig. 1. Compling weights are loaded from teh network admittance matrix through `load_ksakaguchi`, except the line $(71, 73)$ to which we substracted 100 units to the coupling. Coupling frustrations are 70% of what is given by `load_ksakaguchi`. Natural frequencies are chosen in order to properly illustrate how the algorithm works. Over the three winding cells investigated, only two contain a solution, namely those with winding vectors `u1` and `u3`.

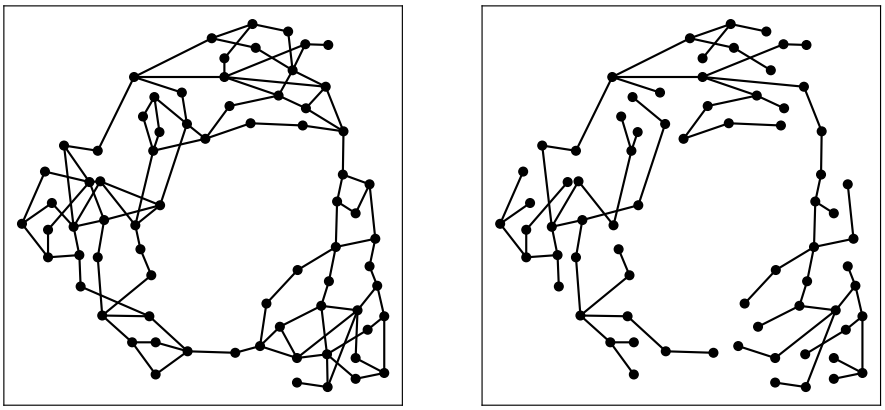

Figure 1: Illustration of the network structure of the IEEE RTS-96 test case (left) of the spanning tree considered in the first part of `examples_rts96.jl` (right) [4].

6 Toolbox

The algorithms of Secs. 3 and 4 rely on a series of routines. We detail them here for sake of completeness and because some of them might be of interest in other contexts.

6.1 cohesiveness_adj

Computes the maximal angular difference (in absolute value) over the edges, using the adjacency (or Laplacian) matrix of the graph.

Inputs:

`θ::Vector{Float64}` **:** Vector of angles.

`A::Matrix{Float64}` **:** Adjacency or Laplacian matrix of the graph.

Outputs:

`dθ::Float64` **:** Maximal angular difference.

6.2 cohesiveness_inc

Computes the maximal angular difference (in absolute value) over the edges, using the incidence matrix of the graph.

Inputs:

`θ::Vector{Float64}` **:** Vector of angles.

`B::Matrix{Float64}` **:** Incidence matrix of the graph.

Outputs:

`dθ::Float64` **:** Maximal angular difference.

6.3 cycle_proj

Computes the cycle projection matrix $\mathcal{P}$, possibly weighted with the weight vector `w`. See [3, 1] for details.

Inputs:

`B::Matrix{Float64}` **:** Incidence matrix of the (undirected) graph.

`w::Vector{Float64}=Float64[]` **:** Weight vector. If none is given, unity weights are considered.

Outputs:

`P::Matrix{Float64}` **:** Cycle projection matrix, possibly weighted.

6.4 dcc

Takes the modulo 2π of `x`, in the interval $[-\pi, \pi)$. Is applied elementwise.

Intputs:

`x::Union{Float64,Vector{Float64},Matrix{Float64}}` **:** Value(s) whose module is to be taken.

Outputs:

`d::Union{Float64,Vector{Float64},Matrix{Float64}}` **:** Result(s) of the modulo.

6.5 deindex

Reverses the action of `reindex` (Sec. 6.10).

Inputs:

`id::Vector{Int64}`: New order of the old node indices (output `id2` of `reindex`).

`ed::Vector{Int64}`: New order of the old edge indices (output `ed2` of `reindex`).

Outputs:

`di::Vector{Int64}`: Ordering of the new node indices in the old ordering of nodes.

`de::Vector{Int64}`: Ordering of the new edge indices in the old ordering of edges.

6.6 dichot

Computes the zero of the function `F` in the interval `[l,u]` with tolerance `tol`, if it exits. The function `F` is assumed strictly monotone and well-defined on `[l,u]`. The algorithm proceeds by dichotomy.

Intputs:

`F::Function`: Function whose zero needs to be found. It is assumed strictly monotone on `[l,u]`.

`l::Float64`: Lower bound of the interval where the zero needs to be found.

`u::Float64`: Upper bound of the interval where the zero needs to be found.

`tol::Float64 = 1e-6`: Tolerance of the algorithm. Set to 10^{-6} if not specified.

Outputs:

`z::Float64`: Zero of the function `F` over `[l,u]`, if it exists.

6.7 L2B

Computes the incidence matrix `B` from the adjacency matrix `L`.

Intputs:

`L::Union{Matrix{Any}, SparseMatrixCSC{Any,Int64}}`: Laplacian matrix of the graph, which can be dense or sparse, weighted or not.

Outputs:

`B::Union{Matrix{Float64}, SparseMatrixCSC{Float64,Int64}}`: Incidence matrix of the graph.

`w::Vector{Any}`: Vector of edge weights, composed of ones if the graph is unweighted.

`Bt::Union{Matrix{Float64}, SparseMatrixCSC{Float64,Int64}}`: Transpose of `B`. Mostly usefull when the input matrix is sparse.

6.8 `load_ksakaguchi`

The Kuramoto-Sakaguchi model [5] is our main example of asymmetric couplings. We provide a convenient way to implement it for our algorithm.

Namely, `load_ksakaguchi` return the vectors of `Function`'s pairs `h` and of `Float64`'s pairs γ, needed to run `acyclic_algorithm`. If it is called with the admittance matrix `Y`, it also returns the incidence matrix `B`.

Inputs 1:

`as::Vector{Float64}` **:** List of coupling weights. Indexing must match the edge indexing in the incidence matrix `B`. The coupling over edge $e \in \{1, ..., m\}$ has weight `as[e]` and the coupling over the edge with opposite orientation has weight `as[e+m]`.

ϕ`s::Vector{Float64}` **:** List of phase frustrations. Indexing must match the edge indexing in the incidence matrix `B`. The frustration over edge $e \in \{1, ..., m\}$ has weight ϕ`s[e]` and the coupling over the edge with opposite orientation has weight ϕ`s[e+m]`.

Outputs 1:

`h::Vector{Function}` **:** Vector of the coupling functions $h_e(x) = a_e \sin(x - \phi_e) + \sin(\phi_e)$.

`hi::Vector{Function}` **:** Vector of the inverses of the coupling functions $h_e^{-1}(f) = \arcsin[f/a_e - \sin(\phi_e)] + \phi_e$.

γ`::Vector{Tuple{Float64,Float64}}` **:** Vector of 2-tuples, composed of the lower and upper bounds of the domains where the coupling functions are strictly increasing. Namely, $\gamma_e = (-\pi/2 + \phi_e, \pi/2 + \phi_e)$.

6.9 `load_ksakaguchi` (bis)

The Kuramoto-Sakaguchi model is typically used to describe the short-term response of voltages in power grids. Given an admittance matrix `Y`, we can define the associated Kuramoto-Sakaguchi model.

Intputs 2:

`Y::Union{Matrix{ComplexF64}, SparseMatrixCSC{ComplexF64,Int64}}` **:** Admittance matrix (dense or sparse) of the power grid. Components are complex numbers: `Y[j,k]` $= g_{jk} + ib_{jk}$. The edge weights are given by $a_{jk} = \sqrt{g_{jk}^2 + b_{jk}^2}$ and the phase frustrations by $\phi_{jk} = \arctan(g_{jk}/b_{jk})$.

Outputs 2:

`h::Vector{Function}` **:** Same as in Outputs 1.

`hi::Vector{Function}` **:** Same as in Outputs 1.

γ`::Vector{Tuple{Float64,Float64}}` **:** Same as Outputs 1.

`B::Union{Matrix{Float64}, SparseMatrixCSC{Float64,Int64}}` **:** Incidence matrix of the network, with edge indexing matching the indexing of `h` and γ.

`as::Vector{Float64}` **:** List of coupling weights.

ϕ`s::Vector{Float64}` **:** List of coupling frustrations.

6.10 `reindex`

Re-indexes the nodes and edges of the graph so it matches the assumptions of [1, Theorem 2].

Intputs:

`B::Union{Matrix{Float64}, SparseMatrixCSC{Float64,Int64}}`**:** Incidence matrix of the **bidirected** graph (dense or sparse). If B' is the incidence matrix of the undirected graph, then `B` is given by $[B' \ -B']$.

`id::Vector{Int64} = Int[]`**:** List of indices of the nodes. If nothing or an empty vector is given, the nodes are assumed to be indexed sequentially `1:n`.

`ed::Vector{Int64} = Int[]`**:** List of edge indices. If nothing is given, the edges are assumed to be indexed sequentially `1:n`.

Outputs:

`B2::Union{Matrix{Float64}, SparseMatrixCSC{Float64,Int64}}`**:** Incidence matrix of the **bidireced** graph with the new indexing of nodes and edges.

`id2::Vector{Int64}`**:** New order of the old node indices.

`ed2::Vector{Int64}`**:** New order of the old edge indices.

6.11 `retro_function`

Recursively defines the flow function on an edge, as a function of the synchronous frequency φ.

Inputs:

`i::Int64`**:** Index of the edge whose flow function has to be defined. Note that, according to the indexing, `i` is also the index of the source node of edge `i`.

`ω::Vector{Float64}`**:** Vector of natural frequencies of the oscillators.

`H::Vector{Function}`**:** Vector of the reciprocal flow functions over all (directed) edges.

`et::Dict{Int64,Vector{Int64}}`**:** Dictionary associating to each node `i`, the list of its in-neighbors. Namely, this is the second output of `targets`.

Outputs:

`F::Function`**:** Flow function over the edge `i`, defined with respect ot the synchronous frequency φ.

6.12 `targets`

In a graph with indexings satisfying the assumption of [1, Theorem 2], computes the unique outgoing edge for each node $i \in \{1, ..., n-1\}$ and the set of nodes that are *in-neighbors* of i.

Inputs:

`B::Union{Matrix{Float64}, SparseMatrixCSC{Float64,Int64}}`**:** Incidence matrix of the undirected graph.

Outputs:

`te::Dict{Int64,Int64}` **:** Dictionary associating to each node $i \in \{1, ..., n-1\}$ the unique node $j \in \{i+1, ..., n\}$ such that the edge $\{i, j\}$ exists.

`et:Dict{Int64,Vector{Int64}}` **:** Dictionary associating to each node $i \in \{1, ..., n\}$ the set of nodes $j \in \{1, ..., i-1\}$ such that the edge $\{j, i\}$ exists.

6.13 `winding`

Computes the winding number(s) associated to the angles θ, around the cycle(s) σ (Σ).

Intputs:

`θ::Vector{Float64}` **:** Vector of angles.

`σ::Vector{Int64}` **or** `Σ::Vector{Vector{Int64}}` **:** σ: Sequence of indices of the nodes of a cycle. Σ: Sequence of σ's.

Outputs:

`q` **:** Winding number or vector.

References

[1] R. Delabays, S. Jafarpour, and F. Bullo. Multistability in the dissipative network flow problem. *under preparation*, 2022.

[2] `https://julialang.org`.

[3] S. Jafarpour, E. Y. Huang, K. D. Smith, and F. Bullo. Flow and elastic networks on the n-torus: Geometry, analysis, and computation. *arXiv preprint: 1901.11189, to appear in SIAM Review*, 2020.

[4] C. Grigg, P. Wong, P. Albrecht, R. Allan, M. Bhavaraju, R. Billinton, Q. Chen, C. Fong, S. Haddad, S. Kuruganty, W. Li, R. Mukerji, D. Patton, N. Rau, D. Reppen, A. Schneider, M. Shahidehpour, and C. Singh. The IEEE Reliability Test System-1996. A report prepared by the Reliability Test System Task Force of the Application of Probability Methods Subcommittee. *IEEE Trans. Power Syst.*, 14(3):1010–1020, 1999.

[5] H. Sakaguchi and Y. Kuramoto. A soluble active rotater model showing phase transitions via mutual entertainment. *Prog. Theor. Phys.*, 76(3):576–581, 1986.