

Numerical resolution of the wave equation on a network of slots

Adrien Semin

▶ **To cite this version:**

Adrien Semin. Numerical resolution of the wave equation on a network of slots. [Technical Report] RT-369, INRIA. 2009, pp.35. inria-00411825

HAL Id: inria-00411825
https://inria.hal.science/inria-00411825

Submitted on 29 Aug 2009

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Numerical resolution of the wave equation on a network of slots

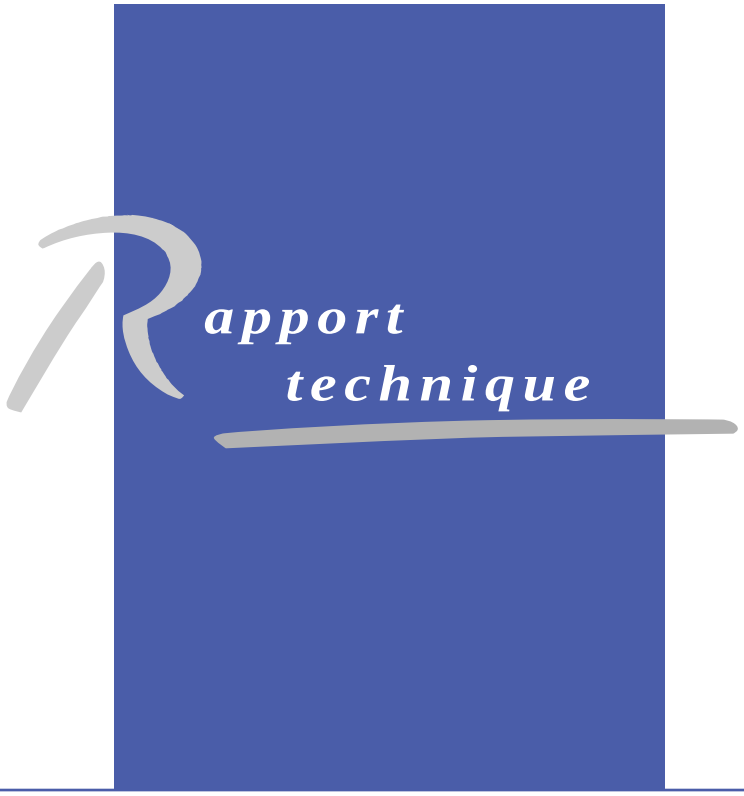
Adrien Semin

Based on a work in collaboration with K. Boxberger

N° 0369

August 2009

Thème NUM



ISSN 0249-0803 ISRN INRIA/RT--0369--FR+ENG

Numerical resolution of the wave equation on a network of slots

Adrien Semin*
Based on a work in collaboration with K. Boxberger

Thème NUM — Systèmes numériques
Projet POems

Rapport technique n° 0369 — August 2009 — 32 pages

Abstract: In this technical report, we present a theoretical and numerical model to simulate wave propagation in finite networks of rods with both classical Kirchhoff conditions and Improved Kirchhoff conditions at the nodes of the networks. One starts with the continuous framework, then we discretize the problem using finite elements with the mass lumping technic introduced by G. Cohen and P. Joly (see [3]). Finally, we show an implementation of the obtained numeric scheme in a homemade code written in C++ in collaboration with K. Boxberger, some results and some error estimates.

Key-words: wave propagation, mass lumping, numeric scheme, network, tree, Kirchhoff

* EPI POems - mail: adrien.semin@inria.fr

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 Le Chesnay Cedex (France)
Téléphone : +33 1 39 63 55 11 — Télécopie : +33 1 39 63 53 30

Résolution numérique de l'équation des ondes sur un réseau de fentes

Résumé : Dans ce rapport technique, nous présentons un modèle théorique et numérique pour simuler la propagation des ondes dans des réseaux finis de fentes avec des conditions de Kirchhoff classiques et améliorées aux nœuds des réseaux. Nous commençons par décrire le cadre continu, puis nous discrétisons le problème en utilisant la technique des éléments finis avec condensation de masse, introduite par G. Cohen et P. Joly (voir [3]). Finalement, nous montrons une implémentation du schéma numérique dans un code écrit en C++ en collaboration avec K. Boxberger, quelques résultats et des estimations d'erreurs.

Mots-clés : propagation d'ondes, condensation de masse, schéma numérique, réseau, arbre, Kirchhoff

Introduction

In this technical report, we extend the results about propagation of acousic wave in a general network of thin slots. Study of waves in networks is not so recent: one can check for example the works of B. Dekoninck and S. Nicaise [5], and in a more recent case the works of B. Maury, D. Salort and C. Vannier [8]. It's also quite easy to do some numerical simulations in some particular case of networks (one see for example the work of Y. Achdou, C. Sabot and N. Tchou [1]). But actually, it is not possible to find an efficient numeric code which solves wave propagation problem on a general network. To avoid this, a subject was proposed for an internship during second trimester 2009. During this internship which was supervised by P. Joly and A. Semin, K. Boxberger studied the propagation of acoustic wave in "semi-homogeneous" networks, i.e. networks whose geometrical and physical parameters are constant functions on each edge of those networks, but are not constant on the whole network. The approach used in this stage was with a finite differences method. The approach allowed us to get some good numeric scheme, but we can see that we lost one order of error estimate when dealing with boundary conditions. The idea is then to use a finite element approach. One could say that numerically, finite element method is slower than finite differences method, that's why we will use the mass-lumping method.

The aim of this technical report is the following: in section 1, we present the continuous problem we want to solve. In section 2, we present the discrete problem associated to the continuous problem. We first use space discretisation, then time discretisation. In section 3, we present an implementation of the numeric code `netwaves` (whose writing has during the previously cited internship - the code can be soonly found at `http://gforge.inria.fr/projects/netwaves/`). In section 4, we give some error estimates for both problems introduced in sections 1 and 2. Finally, in section 5,

1 Setting of the continuous problem

1.1 Geometry of the problem

In this part, we recall some notations about the geometry we consider, in the spirit of [8], but taking the geometry in $\mathbb{R}^d$, with $d = 2$ or $d = 3$. Reader can find more information in [2] or in [7] about the choice of the conductances.

Let $\mathcal{G} = (\mathcal{V}, \mathcal{E}, c)$ denote a graph: $\mathcal{V}$ is a finite set of vertices in $\mathbb{R}^d$, $\mathcal{E}$ subset of $\mathcal{V} \times \mathcal{V}$ such that, for $((A, B), (C, D)) \in \mathcal{E} \times \mathcal{E}$ with $(A, B) \neq (C, D)$, the open segments (AB) and (CD) do not intersect in $\mathbb{R}^d$ and the the intersection of the closed segments $[AB]$ and $[CD]$ is a subset (possibly empty) of $\mathcal{V}$, and $c \in (0, +\infty)^{\mathcal{E}}$ a conductance field. Note that the condition we put on $\mathcal{E}$ let us allow say that edges are counted only once in $\mathcal{E}$, that is to say $(x, y) \in \mathcal{E} \Rightarrow (y, x) \notin \mathcal{E}$. Such example of sets are given by figure 1.1.

We now have to introduce some definitions to describe more precisely the particularities of graph $\mathcal{G}$:

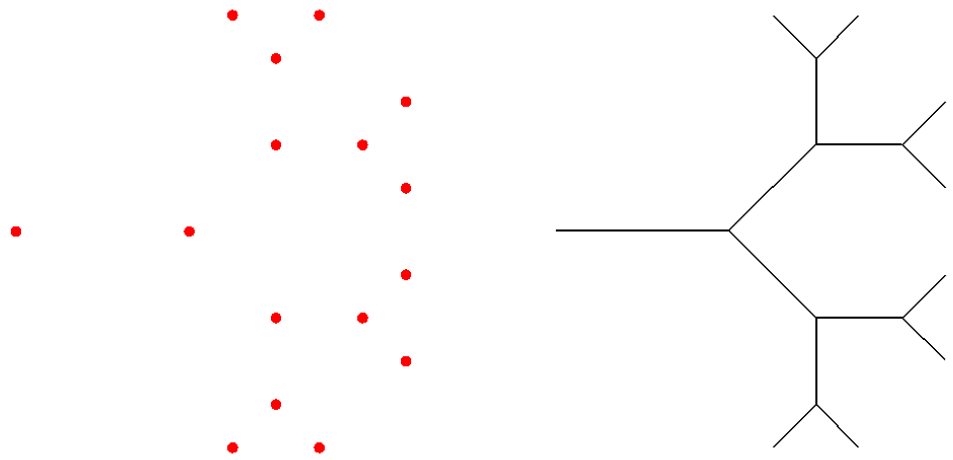


Figure 1.1: On the left: an example of set $\mathcal{V}$ in $\mathbb{R}^2$. On the right: an example of set $\mathcal{E}$ associated to $\mathcal{V}$.

Definition 1.1 (Edges connected to a vertex, interior and exterior vertices)**.** Here we introduce some definitions to describe more precisely the structure of our network.

- *Edges connected to a vertex.* Given $x \in \mathcal{V}$, the set of edges $E(x)$ connected to x is the subset of $\mathcal{E}$ given by elements whose one of the extremities is x. In other words,

$$E(x) := \{(A, B) \in \mathcal{E} \ / \ A = x \text{ or } B = x\}$$

- *Inner and outer vertices.* A vertex $x \in \mathcal{V}$ will be called an outer vertex if the number of edges connected to x is equal to 1. Otherwise, it will be called an inner vertex. We call $\mathcal{V}_e$ the set of outer vertices, and $\mathcal{V}_i$ the set of inner vertices. In other words,

$$\begin{aligned} \mathcal{V}_i &= \{x \in \mathcal{V} \ / \ \#E(x) \geqslant 2\} \\ \mathcal{V}_o &= \{x \in \mathcal{V} \ / \ \#E(x) = 1\} \end{aligned}$$

Remark. In this definition, one assume that there's not any non-connected vertex (i.e. a vertex x satisfying $\#E(x) = 0$).

Now one wants to define the wave equation on this kind of structure: we have to consider parametrization on each edge of the graph, and to define precisely the equation we consider on each edge $e \in \mathcal{E}$ and conditions we put on each vertex $x \in \mathcal{V}$.

Definition 1.2 (Parametrization)**.** Given $e = (A, B) \in \mathcal{E}$, we introduce the parametrisation $\varphi_e :]0, L_e[\to e$, where L_e is the length (in the sense of the Euclidian norm) of e, by

$$\varphi_e : s_e \mapsto A + \frac{B - A}{L_e} s_e$$

Definition 1.3 (Sobolev spaces on a edge)**.** Given $n \in \mathbb{N}$ and $e \in \mathcal{E}$, a function u defined on e will be in the Sobolev space $\mathrm{H}^n(e)$ if and only if the function $u \circ \varphi_e^{-1}$ belongs to $\mathrm{H}^n(]0, L_e[)$.

Definition 1.4 (Discontinuous Sobolev spaces on a graph)**.** Let $\mathcal{G} = (\mathcal{V}, \mathcal{E}, c)$ be a graph and $n \in \mathbb{N}$ given. A function u defined on $\mathcal{G}$ will be in the Discontinuous Sobolev space $\mathrm{H}^n_{\mathrm{d}}(\mathcal{G})$ if and only if the restriction of u to each edge $e \in \mathcal{E}$ belongs to $\mathrm{H}^n(e)$.

1.2 Continuous problem

We want to solve the following problem: find $u \in \mathrm{C}^0(\mathbb{R}_+, \mathrm{H}^1(\mathcal{G})) \cap \mathrm{C}^1(\mathbb{R}_+, \mathrm{H}^0(\mathcal{G}))$ such that:

$$\left\{ \begin{array}{rcll} \dfrac{\partial^2 u}{\partial t^2} - \Delta u & = & 0 & \text{in } e, \forall e \in \mathcal{E} \text{ and } \forall t \in \mathbb{R}_+^* \\ \displaystyle\sum_{e \in E(x)} c(e) \frac{\partial u}{\partial n_{e,i}} & = & 0 & \text{for each } x \in \mathcal{V}_i \text{ and } \forall t \in \mathbb{R}_+^* \\ u & = & f & \text{for } t = 0 \text{ and } \forall e \in \mathcal{E} \\ \dfrac{\partial u}{\partial t} & = & g & \text{for } t = 0 \text{ and } \forall e \in \mathcal{E} \end{array} \right. \tag{1.1}$$

where:

- Δu means the second derivative of u with respect to the curvilinear abscissa, *i.e.*

$$\Delta u = \frac{\partial^2 (u \circ \varphi_e)}{\partial s_e^2} \circ \varphi_e^{-1} \tag{1.2}$$

- $\frac{\partial u}{\partial n_{e,i}}$ means the inner derivative of u with respect to the curvilinear abscissa on an edge $e \in E(x)$, for $x \in \mathcal{V}$. To be more precise, let $x \in \mathcal{V}_i$ and $e \in E(x)$. Then we define $\frac{\partial u}{\partial n_{e,i}}$ by

$$\frac{\partial u}{\partial n_{e,i}}(x) = \left\{ \begin{array}{ll} \dfrac{\partial (u \circ \varphi_e)}{\partial s_e} \circ \varphi_e^{-1}(x) & \text{if } \varphi_e^{-1}(x) = 0, \\ -\dfrac{\partial (u \circ \varphi_e)}{\partial s_e} \circ \varphi_e^{-1}(x) & \text{otherwise.} \end{array} \right.$$

Remark. One can see that the sign of this quantity does not depend of the direction of the parametrization φ_e.

Of course, one can define the outer derivative of u on an edge $e \in E(x)$, for $x \in \mathcal{V}$, as the opposite of the inner derivative. More precisely,

$$\frac{\partial u}{\partial n_{e,o}}(x) = -\frac{\partial u}{\partial n_{e,i}}(x)$$

- $f \in \mathrm{H}^1(\mathcal{E})$ and $g \in \mathrm{H}^1(\mathcal{E})$ vanish near any vertex.

The first line of this system is the "classical" wave equation (one can see that, thanks to (1.2), this line becomes the classical 1D wave equaion in $\mathbb{R}_+^* \times (0, L_e)$). The second line, associated to the fact that U is continuous at the nodes, is the so-called Kirchhoff conditions, named in this way by analogy with the conditions Kirchhoff wrote in 1906 for electrical networks. We will see in section 5 what changes if we use other type of conditions.

To complete the system (1.1), one has to add boundary conditions on the outer vertices:

- (In)homogeneous Dirichlet condition: on a given $x_0 \in \mathcal{V}_o$, one has

$$u(t, x_0) = d_{x_0}(t) \quad \text{where } d_{x_0} \in \mathrm{C}^1(\mathbb{R}_+^*)$$

- (In)homogeneous Neumann condition: on a given $x_0 \in \mathcal{V}_o$, one has

$$\frac{\partial u}{\partial n}(t, x_0) = n_{x_0}(t) \quad \text{where } n_{x_0} \in \mathrm{C}^0(\mathbb{R}_+^*)$$

Remark. One denotes $\frac{\partial u}{\partial n}$ instead of $\frac{\partial u}{\partial n_{e,o}}$. This is due to the fact that on outer vertices, by definition, there's only one edge connected (there's no ambiguity about e). We omit o also for convenience (the convention for normal derivatives is to consider outer normal derivatives).

- Outgoing condition - this is the particular case of the Sommerfield radiation condition: on a given $x_0 \in \mathcal{V}_o$, one has

$$\frac{\partial u}{\partial t}(t, x_0) + \frac{\partial u}{\partial n}(t, x_0) = 0$$

We call respectively $\mathcal{V}_{o,d}$, $\mathcal{V}_{o,n}$ and $\mathcal{V}_{o,o}$ the set of outer vertices with Dirichlet conditions, Neumann conditions and Outgoing conditions. With these conditions, multiplying the first line of (1.1) with the product of $c(e)$ and a test function $v \in \mathrm{H}^1(\mathcal{G})$ which vanishes on each vertex $x \in \mathcal{V}_{o,d}$, and making some integration over the term $(-\Delta u)v$, by noting

$$\nabla u = \frac{\partial \left(u \circ \varphi_e\right)}{\partial s_e} \circ \varphi_e^{-1},$$

one gets, by summing the integrals over all edges:

$$\sum_{e \in \mathcal{E}} \int_e c(e) \frac{\partial^2 u}{\partial t^2}(t, \cdot)v + c(e)\nabla u(t, \cdot)\nabla v + \sum_{x \in \mathcal{V}_{o,n}} c(x) n_x(t) v(x) + \sum_{x \in \mathcal{V}_{o,o}} c(x) \frac{\partial u}{\partial t}(t, x) v(x) = 0 \tag{1.3}$$

Remark. The notation $c(x)$ is an abusive notation. We should rather write $c(e(x))$, where $e(x)$ is the sole edge connected to x.

2 Discretization of the continuous problem

2.1 Discretization in space

Once the continuous problem under its variationnal form (1.3) is defined, one can discretize this problem by using $\mathbb{P}^k$ finite elements with mass lumping. Moreover the fact that mass lumping allows lesser storage and improved computation time, one will see that this method also allows us to uncouple computation on open edges and computation on vertices. In the following we will take $k = 1$.

On a given edge $e \in \mathcal{E}$, we introduce a local space step Δx_e such that $L_e/\Delta x_e$ is a (great) positive integer, denoted N_e. We introduce then, for $0 \leqslant k \leqslant N_e$, the point $x_{e,k}$ given by

$$x_{e,k} = \varphi_e(k\Delta x_e)$$

Note that there are some couples (e, k) which give the same point $x \in \mathbb{R}^d$: these points correspond to inner vertices of the graph.

Once we introduced discretization points, we can introduce on each edge $e \in \mathcal{E}$ the basis functions of $\mathbb{P}^1$ called $(\Phi_{e,k})_{0\leqslant k\leqslant N_e}$ such that:

- for any $l \in \{1, \dots, N_e\}$, $\Phi_{e,k} \circ \varphi_e$ is affine on the open segment $\left(\varphi_e^{-1}(x_{e,l-1}), \varphi_e^{-1}(x_{e,l})\right)$,
- for any $l \in \{0, \dots, N_e\}$, $\Phi_{e,k}(x_{e,l}) = \delta_{kl}$ where δ_{kl} is the Kronecker symbol.

We approximate our functions u and v by writing, on each edge $e \in \mathcal{E}$,

$$u_{|e}(t, x) = \sum_{k=0}^{N_e} U_{e,k}(t)\Phi_{e,k}(x) \quad \text{and} \quad v_{|e}(x) = \sum_{k=0}^{N_e} V_{e,k}\Phi_{e,k}(x)$$

which can be rewritten under the condensed form, by denoting $X_e = (x_{e,0} \dots, x_{e,N_e})^T$ for $X \in \{U, V, \Phi\}$:

$$u(t, x) = \sum_{e\in\mathcal{E}} \Phi_e^t(x)U_e(t) \quad \text{and} \quad v(x) = \sum_{e\in\mathcal{E}} \Phi_e^t(x)V = \sum_{e\in\mathcal{E}} V^t\Phi_e(x) \tag{2.1}$$

We put up the vectorial approximations (2.1) in the variationnal formulation (1.3), and one can see that two natural set of matrices appear:

- rigidity matrices $(\mathfrak{K}_e)_{e\in\mathcal{E}}$ are given by

$$\mathfrak{K}_e = \int_e \nabla\Phi_e \nabla\Phi_e^t = \frac{1}{\Delta x_e} \begin{pmatrix} 1 & -1 & 0 & \dots & 0 \\ -1 & 2 & \ddots & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & 2 & -1 \\ 0 & \dots & 0 & -1 & 1 \end{pmatrix} \tag{2.2}$$

- mass matrices $(\mathfrak{M}_e)_{e\in\mathcal{E}}$ are given by

$$\mathfrak{M}_e = \int_e \Phi_e \Phi_e^t = \frac{\Delta x_e}{6} \begin{pmatrix} 2 & 1 & 0 & \dots & 0 \\ 1 & 4 & \ddots & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & 4 & 1 \\ 0 & \dots & 0 & 1 & 2 \end{pmatrix} \tag{2.3}$$

Instead of using the mass matrices (2.3), we use mass lumping technic (we give here only for the order 1, the reader may see [3] for mass lumped matrices for any $\mathbb{P}^k$). We set that

$$\mathfrak{M}_e = \Delta x_e \begin{pmatrix} \frac{1}{2} & 0 & \dots & \dots & 0 \\ 0 & 1 & \ddots & & \vdots \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & \ddots & 1 & 0 \\ 0 & \dots & \dots & 0 & \frac{1}{2} \end{pmatrix} \tag{2.4}$$

Moreover, for any point $x \in \mathcal{V}_o$, we introduce:

- $e(x)$ the sole element belonging to $E(x)$,
- $\mathfrak{D}_x$ the $N_{e(x)} + 1$ square matrix whose is identically equal to zero, except for:
  - the coefficient on the first line and first column if $\varphi_e^{-1}(x) = 0$,
  - the coefficient on the last line and last column otherwise.

And finally, for any point $x \in \mathcal{V}$ (yes, even for inner vertices), for any $e \in E(x)$, we introduce the two following functions:

- $\mathbf{0}$ defined as $\mathbf{0}(x, e) = 0$ if $\varphi_e^{-1}(x) = 0$ and $\mathbf{0}(x, e) = N_e$ otherwise,
- $\mathbf{1}$ defined as $\mathbf{1}(x, e) = 1$ if $\varphi_e^{-1}(x) = \Delta x_e$ and $\mathbf{1}(x, e) = N_e - 1$ otherwise.

Remark. For these two functions, we will omit the index e when $x \in \mathcal{V}_o$.

2.2 Discretization in time

For the time discretization, we introduce here a global time step Δt, and one denotes U_e^n the approximation of $U_e(t)$ at time $t = n\Delta t$. We discretize first order time derivative by using a centered scheme:

$$\frac{\partial U_e}{\partial t}(n\Delta t) \quad \Rightarrow \quad \frac{U_e^{n+1} - U_e^{n-1}}{2\Delta t} \tag{2.5}$$

and second order time derivative by using classical scheme:

$$\frac{\partial^2 U_e}{\partial t^2}(n\Delta t) \quad \Rightarrow \quad \frac{U_e^{n+1} - 2U_e^n + U_e^{n-1}}{\Delta t^2} \tag{2.6}$$

2.3 Writing the discete problem and resolution

Finally, using notations and matrices of section 2.1 with time discretizations (2.5) and (2.6) leads to the numerical scheme:

$$\sum_{e\in\mathcal{E}} c(e)\left(\mathfrak{M}_e\frac{U_e^{n+1}-2U_e^n+U_e^{n-1}}{\Delta t^2}+\mathfrak{K}_e U_e^n\right)+\sum_{x\in\mathcal{V}_{o,n}} c(x)\mathfrak{D}_x n_x(n\Delta t)+\sum_{x\in\mathcal{V}_{o,o}} c(x)\mathfrak{D}_x\frac{U_{e(x)}^{n+1}-U_{e(x)}^{n-1}}{2\Delta t}=0 \tag{2.7}$$

under the two following constraints:

- continuity of the solution at the inner vertices - for any $x\in\mathcal{V}_i$, for any $e,e'\in E(x)$, one has
$$U_{e,\mathbf{0}(x,e)}^{n+1}=U_{e',\mathbf{0}(x,e')}^{n+1} \tag{2.8}$$
- Dirichlet condition on some outer vertices: for any $x\in\mathcal{V}_{o,d}$, one has
$$U_{e(x),\mathbf{0}(x)}^{n+1}=d_x((n+1)\Delta t) \tag{2.9}$$

2.3.1 Computation at initial time

One has to compute the solution at iterations $n=0$ and $n=1$ (corresponding respectively to times $t=0$ and $t=\Delta t$). Te be more precise about the "near node vanishing" Cauchy datas of the continuous problem (1.1), we suppose that:

Hypothesis 2.1. For any $x\in\mathcal{V}$, for any $e\in E(x)$, for any $V\in\{F,G\}$, one has

$$V_{e,\mathbf{0}(x,e)}=V_{e,\mathbf{1}(x,e)}=0 \tag{2.10}$$

On the following, we treat separately the contribution from the Cauchy data and the contribution from the boundary conditions.

Contribution from the Cauchy data For U^0, we simply write that $U_e^0=F_e$ for each edge $e\in\mathcal{E}$. For U^1, one starts for the D'Alembert formula for the solution of the 1D time-wave equation:

$$u(t,x)=\frac{1}{2}\left(f(x-t)+f(x+t)\right)+\frac{1}{2}\int_{x-t}^{x+t} g(s)ds \tag{2.11}$$

We can use this formula for any inner discretization point on each edge - one can see easily that $U^1(x)=0$ for any $x\in\mathcal{V}$. By using our interpolation formula in (2.11), one gets (assuming that $\Delta t\leqslant\Delta x_e$ for any edge e):

$$\begin{aligned}\forall e\in\mathcal{E},\forall 1\leqslant k\leqslant N_e-1,\quad U_{e,k}^1 &= \frac{\Delta x_e-\Delta_t}{\Delta x_e}F_{e,k}+\frac{\Delta t}{2\Delta x_e}\left(F_{e,k-1}+F_{e,k+1}\right)\\ &+ \frac{\Delta t}{2\Delta x_e}\left(\frac{\Delta t}{2}G_{e,k-1}+\left(2-\frac{\Delta t}{\Delta x_e}\right)G_{e,k}+\frac{\Delta t}{2}G_{e,k+1}\right)\end{aligned} \tag{2.12}$$

Contribution from the boundary conditions For U^0, we simply take into account the Dirichlet part:

$$\forall x \in \mathcal{V}_{o,d}, \quad U^0_{e(x),\mathbf{0}(x)} = d_x(0)$$

For U^1, we have a little more work to do. We have three parts to consider:

- Dirichlet part:
$$\forall x \in \mathcal{V}_{o,d}, \quad U^1_{e(x),\mathbf{0}(x)} = d_x(\Delta t)$$
- propagation of the Dirichlet part: using (2.7) for $n = 0$ (assuming that U^{-1} is identically equal to 0 near vertices) gives that
$$\forall x \in \mathcal{V}_{o,d}, \quad U^1_{e(x),\mathbf{1}(x)} = \frac{\Delta t^2}{\Delta x_e^2} d_x(0)$$
Remark. Saying that U^{-1} is identically equal to 0 is available if and only if the functions d_x and n_x are support in $\mathbb{R}_+$ and if the Cauchy data vanish near vertices.
- Neumann part: still using (2.7) for $n = 0$, one gets that
$$\forall x \in \mathcal{V}_{o,n}, \quad U^1_{e(x),\mathbf{0}(x)} = -\frac{2\Delta_t^2}{\Delta_x} n_x(0)$$

2.3.2 Computation over time

For computation at other times, one simply starts from the numeric scheme (2.7). Knowing U^{n-1} and U^n, it will be very simple, to compute U^{n+1}. The fact that each matric $\mathfrak{M}_e$ is a diagonal matrix let us allow to compute each discretization point independantly of the others.

Computation of inner edges discretization points One simply has

$$\forall e \in \mathcal{E}, \forall 1 \leqslant k \leqslant N_e - 1, \quad \frac{U^{n+1}_{e,k} - 2U^n_{e,k} + U^{n-1}_{e,k}}{\Delta t^2} - \frac{U^n_{e,k+1} - 2U^n_{e,k} + U^n_{e,k-1}}{\Delta x_e^2} = 0$$

Computation of inner vertices Given $x_0 \in \mathcal{V}_i$. Because of condition (2.8), we denote $\mathfrak{U}^{n+1}_{x_0}$ the common value of U^{n+1} at vertex x_0. One has, using the numeric scheme (2.7), and extracting parts corresponding to this vertex,

$$\sum_{e \in E(x)} \frac{c(e)\Delta x_e}{2\Delta t^2} \left(\mathfrak{U}^{n+1}_{x_0} - 2\mathfrak{U}^n_{x_0} + \mathfrak{U}^{n-1}_{x_0}\right) + \frac{c(e)}{\Delta x_e} \left(\mathfrak{U}^n_{x_0} - U^n_{e,\mathbf{1}(x_0)}\right) = 0 \tag{2.13}$$

which gives

$$\mathfrak{U}^{n+1}_{x_0} = 2\mathfrak{U}^n_{x_0} - \mathfrak{U}^{n-1}_{x_0} + 2\Delta t^2 \frac{\sum_{e \in E(x_0)} \frac{c_e}{\Delta x_e} (U^n_{e,\mathbf{1}(x_0)} - \mathfrak{U}^n_{x_0})}{\sum_{e \in E(x_0)} c_e \Delta x_e} \tag{2.14}$$

Computation of outer vertices We have to treat Neumann and Outgoing conditions. Dirichlet condition is given explicitely by (2.9).

- Neumann condition: one gets, for any $x \in \mathcal{V}_{o,n}$:

$$\frac{\Delta x_{e(x)}}{2\Delta t^2}\left(U^{n+1}_{e(x),\mathbf{0}(x)} - 2U^{n}_{e(x),\mathbf{0}(x)} + U^{n-1}_{e(x),\mathbf{0}(x)}\right) + \frac{U^{n}_{e(x),\mathbf{0}(x)} - U^{n}_{e(x),\mathbf{1}(x)}}{\Delta x_e} + n_x(n\Delta t) = 0 \tag{2.15}$$

- Outgoing condition: one gets, for any $x \in \mathcal{V}_{o,o}$:

$$\frac{\Delta x_{e(x)}}{2\Delta t^2}\left(U^{n+1}_{e(x),\mathbf{0}(x)} - 2U^{n}_{e(x),\mathbf{0}(x)} + U^{n-1}_{e(x),\mathbf{0}(x)}\right) + \frac{U^{n}_{e(x),\mathbf{0}(x)} - U^{n}_{e(x),\mathbf{1}(x)}}{\Delta x_e} + \frac{U^{n+1}_{e(x),\mathbf{0}(x)} - U^{n-1}_{e(x),\mathbf{0}(x)}}{2\Delta t} = 0 \tag{2.16}$$

3 Implementation of the discrete problem

Here, we show the implementation of the numeric scheme dscribed in the numerical code `netwaves`. This program uses mainly two types of structures: a structure of geometrical graph, and a structure of problem. Actually, this code considers that Dirichlet and Neumann conditions are homogeneous conditions.

Remark. In this section and associated sub-sections, we'll introduce some classes and some methods (we won't introduce all the classes and methods used, one can check them by retrieving the sources and run `Doxygen`, available at `http://www.stack.nl/~dimitri/doxygen/`). We decided to prefix all members of our classes by an underscore, to differientiate with local variable in our class methods (in addition of using the object ***this**).

3.1 Geometrical graph structure

For the graph structure, one defines a template class struct_graph based on a float point class. The complete syntax is the following:

```
1 template<class real>
2 class struct_graph
3 {
4   protected:
5     std::map<int, std::vector<real> > * _vertices;
6     std::map<int, std::vector<int> > * _edges;
7 }
```

The two associatives maps _vertices and _edges respectively contains $\mathcal{V}$ and $\mathcal{E}$. More precisely:

- index of map _vertices is the label of vertices of the graph, and value correspond to coordinates of those vertices,
- index of map _edges is the label of edges of the graph, and value correspond to the two vertices linked with each edge.

To describe more quickly the following functions, we introduce the two following **typedef**:

```
1 typedef std::map<int,std::vector<real> > tabular_vertices;
2 typedef std::map<int,std::vector<int> > tabular_edges;
```

3.2 Problem structure

For the problem structure, one defines also a template class struct_problem based on a float point class and on a geometrical graph class. The complete syntax is:

```
1  template<class geometry, class real>
2  class struct_problem
3  {
4    protected:
5      geometry * _geom;
6      std::map<int,real> * _hspaces
7      std::map<int,real> * _relative_width
8      std::map<int,std::vector<real> > * _Unminus;
9      std::map<int,std::vector<real> > * _Un;
10     std::map<int,int> * _tabular_boudary_conditions;
11 }
```

Each associative map (except the map _tabular_boudary_conditions which index is linked to index of the map _vertices of member _geom) are indexes linked to index of the map _edges of member _geom. We stock

- in map _hspaces the couples $(e, \Delta x_e)$, for $e \in \mathcal{E}$,
- in map _relative_width the couples $(e, c(e))$ for $e \in \mathcal{E}$ (we called this map as it because the conductance is linked to some geometrical parameters of our network),
- in maps _Unminus and _Un the solution computed on discretization points at iteration $n-1$ and n.

3.3 Set up the problem

3.3.1 Compute initlal data

Assume that we only know the structure of the graph and we know the desired value of space step on each edge, named $\widetilde{\Delta x}_e$. For each edge e:

1. we retreive the coordinates of the two vertices associated with this edge,
2. we compute the length L_e,
3. we compute $N_e = \lfloor L_e/\widetilde{\Delta x}_e \rfloor$ and then $\Delta x_e = L_e/N_e$. This ensures that $\Delta x_e \geqslant \widetilde{\Delta x}_e$, and if the CFL condition is satisfied for the choice $\widetilde{\Delta x}_e$, it will be satisfied for the choice Δx_e,
4. we create a pointer U on a vector whose size is equal to $N_e + 1$ and whose elements are equal to 0,
5. we set up in our problem structure the space step Δx_e and we initialize vectors _Unminus and _Un by the following instructions:

```
1  (*this->_hspaces)[label_edge] = delta_x_e;
2  (*this->_Unminus)[label_edge] = *U;
3  (*this->_Un)[label_edge] = *U;
```

6. we compute the initial data by using formula (2.12) by using this method (thanks to hypothesis 2.1):

```
1  template<typename struct_geom , class real>
2  void struct_problem<struct_geom , real>::init_data(
3                      const real dt,
4                      real (*pointer_f)(real,real,real),
5                      real (*pointer_g)(real,real,real))
6  {
7    std::vector<real> coordinates(3);
8    std::vector<real> coordinates_previous(3);
9    std::vector<real> coordinates_next(3);
10
11   // Loop over edges
12
13   typename tabular_edges::const_iterator
14     edge_iter = this->_geom->get_edges()->begin(),
15     edge_end = this->_geom->get_edges()->end();
16
```

```
17      for(; edge_iter != edge_end; ++edge_iter)
18      {
19        // Get informations about edge
20        int label_edge = edge_iter->first;
21        int label_point1 = (edge_iter->second)[0];
22        int label_point2 = (edge_iter->second)[1];
23        // Retrieve vertices
24        typename tabular_vertices::const_iterator
25          vertex1 = this->_geom->get_vertices()->find(
              label_point1),
26          vertex2 = this->_geom->get_vertices()->find(
              label_point2);
27        // Retrieve number of unknowns and associate Delta x_e
28        int N = this->_Unminus->find(label_edge)->second.size();
29        real dx = this->_hspaces->find(label_edge)->second;
30        // Compute length of edge
31        real length = 0.;
32        for(int j=0; j<3; j++)
33          {
34            length+=((vertex1->second)[j]-(vertex2->second)[j])
35              *((vertex1->second)[j]-(vertex2->second)[j]);
36          }
37        length = std::sqrt(length);
38        // Initialize vectors
39        std::vector<real> *U1 = new std::vector<real>(N,0.),
40          *U2 = new std::vector<real>(N,0.);
41        // Evalutate function (remember that N = N_e + 1)
42        for(int evalpoint=1; evalpoint < N-1; ++evalpoint)
43          {
44            for(int j=0; j<3; j++)
45              {
46                coordinates[j] = real(evalpoint) * dx / length
47                  * ((vertex2->second)[j]-(vertex1->second)[j])
48                  + (vertex1->second)[j];
49                coordinates_previous[j] =
50                  real(evalpoint-1) * dx / length
51                  * ((vertex2->second)[j]-(vertex1->second)[j])
52                  + (vertex1->second)[j];
53                coordinates_next[j] =
54                  real(evalpoint+1) * dx / length
55                  * ((vertex2->second)[j]-(vertex1->second)[j])
56                  + (vertex1->second)[j];
```

```
57              }
58            real Fn = (*pointer_f)(coordinates[0],
59                                   coordinates[1],
60                                   coordinates[2]);
61            real Fnm = (*pointer_f)(coordinates_previous[0],
62                                    coordinates_previous[1],
63                                    coordinates_previous[2]);
64            real Fnp = (*pointer_f)(coordinates_next[0],
65                                    coordinates_next[1],
66                                    coordinates_next[2]);
67            real Gn = (*pointer_g)(coordinates[0],
68                                   coordinates[1],
69                                   coordinates[2]);
70            real Gnm = (*pointer_g)(coordinates_previous[0],
71                                    coordinates_previous[1],
72                                    coordinates_previous[2]);
73            real Gnp = (*pointer_g)(coordinates_next[0],
74                                    coordinates_next[1],
75                                    coordinates_next[2]);
76            (*U1)[evalpoint] = Fn;
77            (*U2)[evalpoint] = (dx-dt)/dx*Fn+dt/(2.*dx)*(Fnm+Fnp
                  )
78              + 0.5*dt * (2. - dt/dx) * Gn
79              + 0.5*dt*dt/(2*dx) * (Gnm+Gnp);
80
81          }
82        (*this->_Unminus)[label_edge] = *U1;
83        (*this->_Un)[label_edge] = *U2;
84        U1->clear(); delete U1;
85        U2->clear(); delete U2;
86      }
87  }
```

3.3.2 Computation over time

Assume that _Unminus and _Un contains the vectors U^{n-1} and U^n. Then we compute U^{n+1} as described in section 2.3.2. The computation is the following (for the sequential form - one can see it would be easy to parallelize the code by using shared memory):

```
1  template<typename struct_geom , class real>
2  void struct_problem<struct_geom , real>::resolve(const real dt)
     const
```

```
3  {
4    // Set up a new map (that will be U^(n+1))
5    std::map<int,std::vector<real> > *Up
6      = new std::map<int, std::vector<real> >(*(this->_Un));
7
8    // Set up map of spacesteps
9    std::map<int,int> *Ne = new std::map<int,int>;
10
11   // First part - computation over edges
12   typename tabular_edges::const_iterator
13     edge_iter = this->_geom->get_edges()->begin(),
14     edge_end = this->_geom->get_edges()->end();
15
16   for(; edge_iter != edge_end; ++edge_iter)
17     {
18       edge_label = edge_iter->first;
19       real dx = this->_hspaces->find(edge_label)->second;
20       real cfl = dt/dx;
21       assume(cfl <= 1.);
22
23       (*Ne)[edge_label]=int((*(this->_Un))[edge_label].size())-1;
24
25       for(int m=1; m < (*Ne)[edge_label]; m++)
26         {
27           (*Up)[edge_label][m]=2*(1-cfl*cfl)
28                                *(*this->_Un)[edge_label][m]
29                                +cfl*cfl*(
30                                  (*this->_Un)[edge_label][m+1]
31                                  +(*this->_Un)[edge_label][m-1] )
32                                -(*this->_Unminus)[edge_label][m];
33         }
34     }
35     // Second part - computation over vertices
36
37     // Set up tabular of edges connected to a given vertex
38     // Convention of boolean : true if vertex is the right point,
39     // false if vertex is the left point
40   std::map<int,bool> tabular_edges_connected;
41
42   typename tabular_vertices::const_iterator
43     vertex_iter = this->_geom->get_vertices()->begin(),
44     vertex_end = this->_geom->get_vertices()->end();
```

```
45
46    for(; vertex_iter!=vertex_end; ++vertex_iter)
47      {
48        int vertex_label=vit->first;
49        int nb_edges2vertex
50          =this->_geom->get_number_of_edges_connected_to_vertex(
                vertex_label);
51        this->_geom->print_edges_connected_to_vertex(
              tabular_edges_connected, vertex_label);
52
53        if(nb_edges2vertex > 1)
54          {
55            // Inner vertex - Kirchhoff conditions
56            real sum_div = 0.; // Contains sum \Delta x_e / c(e)
57            real sum_prod = 0.; // Contains sum \Delta x_e * c(e)
58            real sum_Un  =0. // Contains sum \Delta x_e U_{e,1}^n /
                  c(e)
59            // Retrieve values of U^{n-1} and U^n on this vertex
60            edge_label = sit->first;
61            real val_Un, val_Unminus;
62            if (sit->second)
63              {
64                val_Un = (*this->_Un)[edge_label][(*nb_iter)[
                      edge_label]];
65                val_Unminus = (*this->_Unminus)[edge_label][(*
                      nb_iter)[edge_label]];
66              }
67            else
68              {
69                val_Un = (*this->_Un)[edge_label][0];
70                val_Unminus = (*this->_Unminus)[edge_label][0];
71              }
72            // Loop over edges connected to this vertex
73            std::map<int,bool>::const_iterator
74              sit = tabular_edges_connected.begin(),
75              site = tabular_edges_connected.end();
76            // Create values of differents sums
77            for(; sit != site; ++sit)
78              {
79                // Retrieve label
80                edge_label= sit->first;
```

```
81                real mu = this->_relative_width->find(edge_label)->
                      second;
82                real dx = this->_hspaces->find(edge_label)->second;
83                sum_div += mu / dx;
84                sum_prod += mu * dx;
85                if (sit->second)
86                  {
87                    //vertex on the right side of the edge
88                    sum_Un += mu/dx*(*this->_Un)[edge_label][(*
                          nb_iter)[edge_label]-1];
89                  }
90                else
91                  {
92                    //vertex on the left side of the edge
93                    sum_Un += mu/dx*(*this->_Un)[edge_label][1];
94                  }
95
96              }
97            // Sum up
98            real val_Up = 2. * val_Un - val_Unminus + dt*dt*2.* (
                  sum_Un - sum_div
99            * val_Un) / sum_prod ;
100           // Push values
101           sit = tabular_edges_connected.begin();
102           for(;sit!=site;sit++)
103             {
104               //on edge labelled sit->first
105               edge_label= sit->first;
106               if (sit->second)
107                 {
108                   (*Up)[edge_label][(*nb_iter)[edge_label]]=val_Up
                          ;
109                   //vertex on the right side of the edge
110                 }
111               else
112                 {
113                   //vertex on the left side of the edge
114                   (*Up)[edge_label][0]=val_Up;
115                 }
116             }
117         }
118       else
```

```
119           {
120             // Outer vertex
121             edge_label = tabular_edges_connected.begin()->first;
122             int BC = this->_tabular_boundary_conditions->find(
                    vertex_label)->second,
123             int zero,one;
124             if (tabular_edges_connected.begin()->second)
125               {
126                 zero = (*nb_iter)[edge_label];
127                 one = (*nb_iter)[edge_label]-1;
128               }
129             else
130               {
131                 zero = 0;
132                 one = 1;
133               }
134             switch (BC)
135               {
136                 case bc_dirichlet:
137                   (*Up)[edge_label][zero]=0;
138                 break;
139                 case bc_neumann:
140                   (*Up)[edge_label][zero]
141                     = 2 * (*this->_Un)[edge_label][zero]
142                     + (2*dt*dt/dx/dx) * ( (*this->_Un)[edge_label][
                            one]
143                                         - (*this->_Un)[edge_label][
                                              zero])
144                                         - (*this->_Unminus)[edge_label
                                              ][zero];
145                 break;
146                 case bc_outgoing:
147                   real c1,c2,c3;
148                   c1 = dx/(2.*dt*dt) + 1./dt;
149                   c2 = dx/(dt*dt);
150                   c3 = - dx/(2.*dt*dt) + 1./dt;
151                   (*Up)[edge_label][zero] = (
152                       c2 * (*this->_Un)[edge_label][zero]
153                     + c3 * (*this->_Unminus)[edge_label][zero]
154                     + ((*this->_Un)[edge_label][one]
155                       -(*this->_Un)[edge_label][zero])) / c1;
156                 break;
```

```
157                 default:
158                   std::cerr << "ERROR : : resolve() : : Condition "
159                             << BC << "not yet implemented" << std::
                                  endl;
160                   std::cerr << "Aborting..." << std::endl;
161                   exit(EXIT_FAILURE);
162                 break;
163               }
164           }
165       }
166   }
```

Some remarks about this listing:

- on line 51, we get in the map tabular_edges_connected the set $E(x)$, for each $x \in \mathcal{V}$ (we do not differientiate if $x \in \mathcal{V}_i$ or not),
- on lines 62-71, since the values $\mathfrak{U}^{n-1}$ and $\mathfrak{U}^n$ does not depend on the edge, we look for these values on the first edge (in sense of the std :: less order associated to **int**) e, and we write that
$$\mathfrak{U}^{n-1} = U^{n-1}_{e,\mathbf{0}(x)} \quad \text{and} \quad \mathfrak{U}^n = U^n_{e,\mathbf{0}(x)}$$
lines 68-71 are for the case $\mathbf{0}(x) = 0$, lines 63-66 are for the other case.
- on lines 124-133, we compute, for $x \in \mathcal{V}_0$, the values of $\mathbf{0}(x)$ and $\mathbf{1}(x)$, then we use this values in various boundary conditions implemented in lines 134-163.

4 Error estimates

In this section we give some error estimate associated to both continuous framework (section 1) and discrete framework (section 2). One has to define some energy, and then to check error estimates both theoretically and numerically.

4.1 Continuous framework

One can check in (1.3) that taking $v = \frac{\partial u}{\partial t}$ leads to the relation

$$\frac{1}{2}\frac{\partial}{\partial t}\left[\sum_{e\in\mathcal{E}} c(e)\int_e \left|\frac{\partial u}{\partial t}(t,\cdot)\right|^2 + |\nabla u(t,\cdot)|^2\right] + \sum_{x\in\mathcal{V}_{o,n}} c(x)n_x(t)\frac{\partial u}{\partial t}(t,x) + \sum_{x\in\mathcal{V}_{o,o}} c(x)\left|\frac{\partial u}{\partial t}(t,x)\right|^2 = 0 \tag{4.1}$$

From (4.1), one can define some continuous energy by the following:

Definition 4.1. The energy of solution u of (1.3) at time t is defined by

$$\mathfrak{E}(t) = \frac{1}{2}\left[\sum_{e\in\mathcal{E}} c(e)\int_e \left|\frac{\partial u}{\partial t}(t,\cdot)\right|^2 + |\nabla u(t,\cdot)|^2\right] \tag{4.2}$$

The idea is to get some *a priori* error estimates. For this case, we suppose that there's only Neumann or Outgoing conditions (for the Dirichlet conditions, one has to introduce some function η locally supported the graph satisfying the Dirichlet conditions, and to check up which problem is satisfied by $u-\eta$). We will suppose also that the Neumann functions n_x belong to the set $\mathrm{H}^1(\mathbb{R}_+^*)$. One starts from (4.1), integrate over $t\in[0,T]$, and by using the fact that some terms are positive, one gets:

$$\mathfrak{E}(T) - \mathfrak{E}(0) \leqslant -\sum_{x\in\mathcal{V}_{o,n}} c(x)\int_0^T n_x(t)\frac{\partial u}{\partial t}(t,x)dt \tag{4.3}$$

Using some intergration by parts in (4.3) and using that $u(0,x)=0$ for each $x\in\mathcal{V}$, one gets

$$\mathfrak{E}(T) - \mathfrak{E}(0) \leqslant -\sum_{x\in\mathcal{V}_{o,n}} c(x)n_x(T)u(T,x) + \sum_{x\in\mathcal{V}_{o,n}} c(x)\int_0^T n_x'(t)u(t,x)dx \tag{4.4}$$

To be able to conclude, one has to use the two following lemmas

Lemma 4.1 (Poincare-Wirtinger inequality)**.** *For any $x\in\mathcal{V}_{o,n}$, by taking $e=e(x)$, there exists a constant C_e which depends only of e such that, for any $t\in\mathbb{R}_+$*

$$|u(t,x)| \leqslant C_e\,\|u\|_{\mathrm{H}^1(e)} \leqslant C_e\left(\|u(t,\cdot)\|_{\mathrm{L}^2(e)} + \|\nabla u(t,\cdot)\|_{\mathrm{L}^2(e)}\right)$$

Lemma 4.2. *For any $t\in\mathbb{R}_+$, for any $e\in\mathcal{E}$, one has*

$$\|u(t,\cdot)\|_{\mathrm{L}^2(e)} \leqslant 2\left(\|u(0,\cdot)\|_{\mathrm{L}^2(e)} + \sqrt{t}\int_0^t \left\|\frac{\partial u}{\partial t}(s,\cdot)\right\|_{\mathrm{L}^2(e)} ds\right)$$

Proof of lemma 4.2. One starts from

$$\begin{aligned}
\|u(t,\cdot)\|^2_{\mathrm{L}^2(e)} &= \int_e u(t,x)^2 dx \\
&= \int_e u(t,x)\left(u(0,x) + \int_0^t \frac{\partial u}{\partial t}(s,x)ds\right)dx \\
&\leqslant \int_e |u(t,x)|\left(|u(0,x)| + \sqrt{t}\left(\int_0^t \left|\frac{\partial u}{\partial t}(s,x)\right|^2 ds\right)^{1/2}\right)dx
\end{aligned}$$

Then using Cauchy-Schwartz inequality gives the relation of the lemma. □

By using both lemmas 4.1 and 4.2, there exists two constant C, C' and $C"$ depending on time such that

$$\mathfrak{E}(T) \leqslant \mathfrak{E}(0) + C(T) + \int_0^T C'(t)\sqrt{\mathfrak{E}(t)}dt + C"(T)\sqrt{\mathfrak{E}(T)} \tag{4.5}$$

Remark. One can easily see that all these constants are equal to zero when all Neumann conditions are homogeneous Neumann conditions.

Now using the fact that

$$C"(T)\sqrt{\mathfrak{E}(T)} \leqslant \frac{(C"(T))^2}{2} + \frac{1}{2}\mathfrak{E}(t)$$

in (4.5) gives that there exist two constants $\tilde{C}(T)$ and $C'(T)$ (which is the same constant as before) such that

$$\mathcal{E}(T) \leqslant \tilde{C}(T) + \int_0^T C'(t)\sqrt{\mathfrak{E}(t)}dt \tag{4.6}$$

Finally, using Gronwall inequality leads to the following *a priori* following majoration:

Theorem 4.3. *There exist functions $\tilde{C}$ and C' which depends on time and on boundary conditions of problem (1.1) such that, for all $T \in \mathbb{R}_+$,*

$$\mathfrak{E}(T) \leqslant \left(\sqrt{\tilde{C}(T)} + \frac{1}{2}\int_0^T C'(t)dt\right)^2$$

where $\tilde{C}$ and C' are the previously defined constants.

Remark. When all Neumann conditions are homogeneous Neumann conditiosn, theorem 4.3 gives that energy $\mathfrak{E}T$ is bounded over time (see back the remark associated to equation (4.5)).

4.2 Discrete framework

Here we will suppose for convenience that we have only homogeneous Neumann conditions. We recall the numeric scheme (2.7) with our hypothesis:

$$\sum_{e\in\mathcal{E}} c(e)\left(\mathfrak{M}_e\frac{U_e^{n+1} - 2U_e^n + U_e^{n-1}}{\Delta t^2} + \mathfrak{K}_e U_e^n\right) = 0$$

One has to mulltiply this line by $U_e^{n+1} - U_e^{n-1}$, and one gets, by definying the following discrete energy

$$\begin{aligned}\mathfrak{E}^n &= \sum_{e\in\mathcal{E}} c(e)\left(\frac{U_e^{n+1} - U_e^n}{\Delta t}\right)^t \mathfrak{M}_e\left(\frac{U_e^{n+1} - U_e^n}{\Delta t}\right)\\ &+ \sum_{e\in\mathcal{E}} c(e)(U_e^{n+1})^t\mathfrak{K}_e U_e^n\end{aligned} \tag{4.7}$$

the fact that this energy is constant over time. However, one can also check that not all the terms of energy defined by (4.7) are positive ones. To avoid that, one remarks that, thanks to the symmetry of each $\mathfrak{K}_e$,

$$\begin{aligned}(U_e^{n+1})^t \mathfrak{K}_e U_e^n &= \frac{1}{2}\left((U_e^{n+1})^t \mathfrak{K}_e U_e^{n+1} + (U_e^n)^t \mathfrak{K}_e U_e^n\right) \\ &- \frac{\Delta t^2}{2}\left(\frac{U_e^{n+1}-U_e^n}{\Delta t}\right)^t \mathfrak{K}_e \left(\frac{U_e^{n+1}-U_e^n}{\Delta t}\right)\end{aligned} \tag{4.8}$$

Using (4.8) in (4.7) let us allow to rewrite the discrete energy in a different way: one has

$$\begin{aligned}\mathfrak{E}^n &= \sum_{e\in\mathcal{E}} c(e)\left(\frac{U_e^{n+1}-U_e^n}{\Delta t}\right)^t \left(\mathfrak{M}_e - \frac{\Delta t^2}{2}\mathfrak{K}_e\right)\left(\frac{U_e^{n+1}-U_e^n}{\Delta t}\right) \\ &+ \sum_{e\in\mathcal{E}} \frac{c(e)}{2}\left((U_e^{n+1})^t \mathfrak{K}_e U_e^{n+1} + (U_e^n)^t \mathfrak{K}_e U_e^n\right)\end{aligned} \tag{4.9}$$

Let us define, for each edge $e \in \mathcal{E}$, the two follwing matrices

$$\widetilde{\mathfrak{M}_e} = \frac{2}{\Delta x_e}\mathfrak{M}_e = \begin{pmatrix} 1 & 0 & \dots & \dots & 0 \\ 0 & 2 & \ddots & & \vdots \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & \ddots & 2 & 0 \\ 0 & \dots & \dots & 0 & 1 \end{pmatrix} \tag{4.10}$$

and

$$\widetilde{\mathfrak{K}_e} = \Delta x_e \mathfrak{K}_e = \begin{pmatrix} 1 & -1 & 0 & \dots & 0 \\ -1 & 2 & \ddots & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & 2 & -1 \\ 0 & \dots & 0 & -1 & 1 \end{pmatrix} \tag{4.11}$$

One has then, thanks to (4.10) and (4.11),

$$\mathfrak{M}_e - \frac{\Delta t^2}{2}\mathfrak{K}_e = \frac{\Delta x_e}{2}\left(\widetilde{\mathfrak{M}_e} - \frac{\Delta t^2}{\Delta x_e^2}\widetilde{\mathfrak{K}_e}\right) \tag{4.12}$$

The advantage to introduce the matrices $\widetilde{\mathfrak{M}_e}$ and $\widetilde{\mathfrak{K}_e}$ is that their coefficients do not depend on the discretisation. Let us now resolve the generalised eigenvalue problem: find $(\lambda, X) \in \mathbb{R} \times \mathbb{R}^{N_e}$ such that

$$\widetilde{\mathfrak{K}_e} X = \lambda \widetilde{\mathfrak{M}_e} X \tag{4.13}$$

Then one can easily say that, for all $X \in \mathbb{R}^{N_e}$,

$$X^t \left(\widetilde{\mathfrak{M}_e} - \frac{\Delta t^2}{\Delta x_e^2} \widetilde{\mathfrak{K}_e} \right) X \geqslant (1 - \frac{\Delta t^2}{\Delta x_e^2} \lambda_m) X^t \widetilde{\mathfrak{M}_e} X \tag{4.14}$$

where λ_m is the largest eigenvalue of problem (4.13). Rouch calculus show that $\lambda_m \leqslant 1$. Then one can say that

Theorem 4.4. *The energy $\mathfrak{E}^n$ contains positive terms (and then numeric scheme is stable) if and only if*

$$\forall e \in \mathcal{E}, \quad \frac{\Delta t}{\Delta x_e} \leqslant 1$$

Remark. One can also check by computation that the numeric scheme is not stable (with exponential increasing of the computed solution) if the condition of theorem 4.4 is not satisfied on at least one edge.

5 Dealing with Improved Kirchhoff conditions

Instead of writing problem (1.1) with Kirchhoff conditions, we want to solve the following problem: find $u \in \mathrm{C}^0(\mathbb{R}_+, \mathrm{H}^1_{\mathrm{d}}(\mathcal{G})) \cap \mathrm{C}^1(\mathbb{R}_+, \mathrm{H}^0(\mathcal{G}))$ such that:

$$\left\{ \begin{array}{rcl} \dfrac{\partial^2 u}{\partial t^2} - \Delta u &=& 0 \quad \text{in } e, \forall e \in \mathcal{E} \text{ and } \forall t \in \mathbb{R}_+^* \\ u &=& f \quad \text{for } t = 0 \text{ and } \forall e \in \mathcal{E} \\ \dfrac{\partial u}{\partial t} &=& g \quad \text{for } t = 0 \text{ and } \forall e \in \mathcal{E} \end{array} \right. \tag{5.1}$$

with some improved Kirchhoff conditions, introduced by P. Joly and A. Semin in [6] for the particular case of two slots and in [7] for the general case (which interest us). However, we have to define some notations. In particular, the notation $\mathfrak{U}$ here will differs from the notation introduced in section 2.3.2.

5.1 Improved Kirchhoff conditions

We first have to define an *order* function, which says us how we locally number our edges.

Definition 5.1. An order function on the graph is a function $\mathfrak{o}$ defined on $\mathcal{V}_i \times E(\mathcal{V}_i) = \{x, E(x)\}_{x \in \mathcal{V}_i}$ with values in $\mathbb{N}^*$ such that for any $x \in \mathcal{V}_i$, $\mathfrak{o}(x, \cdot)$ is a bijection between $E(x)$ and $\{1, \ldots, \#E(x)\}$. We will then denote $\mathfrak{o}^{-1}(x, \cdot) : \{1, \ldots, \#E(x)\} \in \mathcal{E}(x)$ the reciprocal bijection.

Once we defined our order function $\mathfrak{o}$, for any $x \in \mathcal{V}_i$, one can denote:

- $\mathfrak{U}_x(t)$ the vector in $\mathbb{R}^{\#E(x)}(t)$ defined by

$$\mathfrak{U}_x(t) = \begin{pmatrix} u(t,x) \text{ defined on the edge } \mathfrak{o}^{-1}(x,1) \\ \vdots \\ u(t,x) \text{ defined on the edge } \mathfrak{o}^{-1}(x,\#E(x)) \end{pmatrix} \tag{5.2}$$

  this vector is not proportionnal to the vector $(1,\dots,1)^t$, since the function is not continous on each vertex $x \in \mathcal{V}_i$.

- $\partial_n \mathfrak{U}_x(t)$ the vector in $\mathbb{R}^{\#E(x)}(t)$ defined by

$$\partial_n \mathfrak{U}_x(t) = \begin{pmatrix} \frac{\partial u}{\partial n_{\mathfrak{o}^{-1}(x,1),i}}(t,x) \\ \vdots \\ \frac{\partial u}{\partial n_{\mathfrak{o}^{-1}(x,\#E(x)),i}}(t,x) \end{pmatrix} \tag{5.3}$$

Improved conditions will be given as follow: for any $x \in \mathcal{V}_i$, there exist two matrices $\mathcal{J}$ and $\mathcal{A}$ in $\mathcal{M}_{\#E(x)}(\mathbb{R})$ such that

$$\partial_n \mathfrak{U}_x(t) = \left(\mathcal{J} + \mathcal{A}\frac{\partial^2}{\partial t^2}\right)\mathfrak{U}_x(t) \tag{5.4}$$

The problem with Improved Kirchhoff conditions is (5.1) with (5.4), i.e.: find $u \in \mathrm{C}^0(\mathbb{R}_+, \mathrm{H}^1_{\mathrm{d}}(\mathcal{G})) \cap \mathrm{C}^1(\mathbb{R}_+, \mathrm{H}^0(\mathcal{G}))$ such that:

$$\begin{cases} \dfrac{\partial^2 u}{\partial t^2} - \Delta u &= 0 \quad \text{in } e, \forall e \in \mathcal{E} \text{ and } \forall t \in \mathbb{R}_+^* \\ \partial_n \mathfrak{U}_x(t) &= \left(\mathcal{J} + \mathcal{A}\dfrac{\partial^2}{\partial t^2}\right)\mathfrak{U}_x(t), \quad \forall x \in \mathcal{V}_i, \forall t \in \mathbb{R}_+^* \\ u &= f \quad \text{for } t = 0 \text{ and } \forall e \in \mathcal{E} \\ \dfrac{\partial u}{\partial t} &= g \quad \text{for } t = 0 \text{ and } \forall e \in \mathcal{E} \end{cases} \tag{5.5}$$

with classical conditions (Dirichlet, Neumann, Outgoing). The variationnal formulation is: find $u \in \mathrm{C}^0(\mathbb{R}_+, \mathrm{H}^1_{\mathrm{d}}(\mathcal{G})) \cap \mathrm{C}^1(\mathbb{R}_+, \mathrm{H}^0(\mathcal{G}))$ satisfying inhomogeneous Dirichlet condition such that, for any $v \in \mathrm{H}^1_{\mathrm{d}}(\mathcal{G})$ satisfying associated homogeneous Dirichlet, one has (denoting $\mathfrak{V}_x$ the vector for v defined as (5.2) for u):

$$\begin{aligned} \sum_{e\in\mathcal{E}} \int_e c(e)\frac{\partial^2 u}{\partial t^2}(t,\cdot)v + c(e)\nabla u(t,\cdot)\nabla v + \sum_{x\in\mathcal{V}_{o,n}} c(x)n_x(t)v(x) + \sum_{x\in\mathcal{V}_i} \mathfrak{V}_x^t \mathcal{J}\mathfrak{U}_x(t) + \mathfrak{V}_x^t \mathcal{J}\frac{\partial^2 \mathfrak{U}_x}{\partial t^2}(t) \\ + \sum_{x\in\mathcal{V}_{o,o}} c(x)\frac{\partial u}{\partial t}(t,x)v(x) = 0 \end{aligned} \tag{5.6}$$

5.2 Discretization

The discrete problem associated to the new variationnal formulation (5.6) is the same as the discrete problem described in section 2.3, except for inner vertices which are dealt as below: for any $x \in \mathcal{V}_i$, one introduces

- vector $\mathfrak{U}_x^n$ is defined as the vector containing the values of U on each discretization point associated to vertex x on each edge in $E(x)$, i.e.

$$\mathfrak{U}_x^n = \begin{pmatrix} U^n_{\mathfrak{o}^{-1}(x,1),\mathbf{0}(x)} \\ \vdots \\ U^n_{\mathfrak{o}^{-1}(x,\#E(x)),\mathbf{0}(x)} \end{pmatrix}$$

- vector $\mathfrak{U}_{x,\Delta}^n$ is defined as the vector containing the values of U on each discretization point next to discretization points associated to vertex x on each edge in $E(x)$, i.e.

$$\mathfrak{U}_{x,\Delta}^n = \begin{pmatrix} U^n_{\mathfrak{o}^{-1}(x,1),\mathbf{1}(x)} \\ \vdots \\ U^n_{\mathfrak{o}^{-1}(x,\#E(x)),\mathbf{1}(x)} \end{pmatrix}$$

- Diagonal matrix $\mathcal{C}$ defined by

$$\mathcal{C} = \begin{pmatrix} c(\mathfrak{o}^{-1}(x,1)) & 0 & \dots & 0 \\ 0 & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \dots & 0 & c(\mathfrak{o}^{-1}(x,\#E(x))) \end{pmatrix}$$

- Diagonal matrix Δ_X defined by

$$\Delta_x = \begin{pmatrix} \Delta x_{(\mathfrak{o}^{-1}(x,1))} & 0 & \dots & 0 \\ 0 & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \dots & 0 & \Delta x_{(\mathfrak{o}^{-1}(x,\#E(x)))} \end{pmatrix}$$

Then, for inner vertex, instead of computing (2.14), one has to solve this system:

$$\begin{aligned} \frac{1}{2\Delta t^2}\mathcal{C}\Delta_X\left(\mathfrak{U}_x^{n+1} - 2\mathfrak{U}_x^n + xU_x^{n-1}\right) &+ \mathcal{C}\Delta_x^{-1}\left(\mathfrak{U}_x^n - \mathfrak{U}_{x,\Delta}^n\right) \\ &+ \mathcal{J}\frac{\mathfrak{U}_x^{n+1} + 2\mathfrak{U}_x^n + xU_x^{n-1}}{4} \\ &+ \mathcal{A}\frac{\mathfrak{U}_x^{n+1} - 2\mathfrak{U}_x^n + xU_x^{n-1}}{\Delta t^2} = 0 \end{aligned} \tag{5.7}$$

Remark. One can check in (5.7) that the term associated to $\mathcal{J}$ has been treated implicitely. This is not classical, and this is due to the fact that if we do not implicit this term, there might be some resonance frequencies (linked to the amplitude of matrix $\mathcal{A}$) that will corrupt CFL condition of theorem 4.4 - the case for two slots has been proved in [9].

5.3 Implementation

In our class struct_problem, we add some associative maps, whose indexes will be indexed on indexes of _geom->_vertices.

```
1  template<class geometry, class real>
2  class struct_problem
3  {
4    protected:
5      // ... prevously defined members
6      std::map<int, std::vector<int> > _order;
7      std::map<int, Matrix* > _J;
8      std::map<int, Matrix* > _A;
9  }
```

The class Matrix is the class of matrices of the external library `newmat`, written by R.B. Davies (see [4] for instance).

For initialization of the problem, the matrices _J and _A are read in a data file, as the vector _order, and one checks that the dimensions of those objects match with the number of edges connected to associated vertex. Moreover, one checks (when building vector _order), that there's no element associated with two differents keys, with the following code:

```
1  std::map<int,int> order2;
2  std::vector<int> const_iterator
3    it = this->_order->find(vertex_label)->begin(),
4    ite = this->_order->find(vertex_label)->begin()
5
6  for( ; it != ite ; ++it)
7  {
8    order2[it->second] = it->first;
9  }
10
11 if (_this->order->size() != order2.size())
12 {
13   // In this case, there was two differents iterators with
14   // the same value. We exit with fail
15   exit(EXIT_FAILURE);
```

```
16 }
```

For computation over time, one changes lines 55-116 of C++ code given in implementation of computation over time by the following lines:

```
1  {
2    // Get tabular connected to edges
3    this->_geom->print_edges_connected_to_vertex(
       tabular_edges_connected, vertex_label);
4    // Definition of vectors U (gothic shape) and compute for
5    // iterations n-1 and n
6    ColumnVector Uminus(nb_edges2vertex),
7                 U(nb_edges2vertex),
8                 U_dx(nb_edges2vertex),
9                 Uplus(nb_edges2vertex);
10
11   // Values of functions 0 and 1
12   int zero,one;
13
14   for(int j=0; j < nb_edges2vertex; ++j)
15     {
16       // Computation of o^-1(x,j)
17       int edge_label = (*this->_order)[vertex_label][j];
18       // Test if vertex is the right bound or the left bound
19       if (tabular._edges_connected.find(edge_label)->second)
20         {
21           zero = (*nb_iter)[edge_label];
22           one =  (*nb_iter)[edge_label]-1;
23         }
24       else
25         {
26           zero = 0;
27           one = 1;
28         }
29       Uminus(j)=double((*this->_Unminus)[edge_label][zero]);
30       U(j)=double((*this->_Un)[edge_label][zero]);
31       U_dx(j)=double((*this->_Un)[edge_label][one]);
32     }
33
34   // Definition of diagonal matrices C and Delta
35   DiagonalMatrix C(nb_edges2vertex), Delta(nb_edges2vertex);
36   for(int j=0; j < nb_edges2vertex; ++j)
```

```
37      {
38        // Computation of o^-1(x,j)
39        int edge_label = (*this->_order)[vertex_label][j];
40        C(j,j) = double((*this->_relative_width)[edge_label]);
41        Delta(j,j) = double((*this->_hspaces)[edge_label])
42      }
43
44    // Computation of U (gothic shape) at iteration n+1
45    Matrix M = 0.5 * C * Delta
46             + 0.25 * double(dt*dt) * (*this->_J)
47             + (*this->_A);
48    Matrix N = C * Delta - 0.5 * double(dt*dt) * (*this->_J)
49             + 2. * (*this->_A);
50    // Here we invert matrix M
51    M = M.i();
52    Uplus = M * N * Un + M * C * Delta.i() * (Un_dx - Un) -Uminus;
53
54    // Put back in map Up
55    for(int j=0; j < nb_edges2vertex; ++j)
56      {
57        // Computation of o^-1(x,j)
58        int edge_label = (*this->_order)[vertex_label][j];
59        // Test if vertex is the right bound or the left bound
60        if (tabular._edges_connected.find(edge_label)->second)
61          {
62            zero = (*nb_iter)[edge_label];
63          }
64        else
65          {
66            zero = 0;
67          }
68        (*Up)[edge_label][zero]=real(Uplus(j));
69      }
70
71    // Memory freedom
72    Uminus.Release(); U.Release();
73    U_dx.Release(); Uplus.Release();
74    C.Release(); Delta.Release();
75    M.Release(); N.Release();
76  }
```

Conclusion

We showed the existence of a numeric scheme solving propagation of acoustic wave in networks. Moreover, we can say that this numeric scheme is stable under some classical conditions, and implementation of this scheme is easy - one does not need to use matrices to implement this scheme. However we can see that the time step used is a *global* time step - one expects to use a *local* time step, in the spirit of the works of J. Rodriguez.

Acknolowedgment I would like to thanks P. Joly for his knowledge which helped me in some technical points, and K. Boxberger with helped me to provide the code `netwaves`.

References

[1] Yves Achdou, Christophe Sabot, and Nicoletta Tchou. Transparent boundary conditions for the Helmholtz equation in some ramified domains with a fractal boundary. *J. Comput. Phys.*, 220(2):712–739, 2007.

[2] Katrin Boxberger. Propagation d' ondes dans un réseau de fentes minces: Étude mathématique et simulation. Master's thesis, Université Paris-Sud, 2009.

[3] Gary Cohen, Patrick Joly, and Nathalie Tordjman. Higher-order finite elements with mass-lumping for the 1D wave equation. *Finite Elem. Anal. Des.*, 16(3-4):329–336, 1994. ICOSAHOM '92 (Montpellier, 1992).

[4] Robert Davies. Writing a matrix package in c++. In *Rogue Wave Software*, volume OON-SKI'94: The second annual object-oriented numerics conference, pages 207–213, 1994.

[5] Bertrand Dekoninck and Serge Nicaise. Spectre des réseaux de poutres. *C. R. Acad. Sci. Paris Sér. I Math.*, 326(10):1249–1254, 1998.

[6] Patrick Joly and Adrien Semin. Propagation of an acoustic wave in a junction of two thin slots. Research Report RR-6708, INRIA, October 2008.

[7] Patrick Joly and Adrien Semin. Construction and Analysis of Improved Kirchoff Conditions for Acoustic Wave Propagation in a Junction of Thin Slots. In Hélène Barucq, Anne Sophie Bonnet-Ben Dhia, Gary Cohen, Julien Diaz, Abdelaâziz Ezziani, and Patrick Joly, editors, *Proceedings of Waves 2009*, pages 140–141. INRIA Rocquencourt, June 2009.

[8] Bertrand Maury, Delphine Salort, and Christine Vannier. Trace theorems for trees, application to the human lungs. *Network and Heteregeneous Media*, 4(3):469 – 500, September 2009.

[9] Adrien Semin. Propagation d'ondes dans des jonctions de fentes minces. Master's thesis, Université Paris-Sud, 2007.

Contents

Introduction **3**

1 Setting of the continuous problem **3**
1.1 Geometry of the problem . . . 3
1.2 Continuous problem . . . 5

2 Discretization of the continuous problem **7**
2.1 Discretization in space . . . 7
2.2 Discretization in time . . . 8
2.3 Writing the discete problem and resolution . . . 9
2.3.1 Computation at initial time . . . 9
2.3.2 Computation over time . . . 10

3 Implementation of the discrete problem **11**
3.1 Geometrical graph structure . . . 11
3.2 Problem structure . . . 12
3.3 Set up the problem . . . 13
3.3.1 Compute initlal data . . . 13
3.3.2 Computation over time . . . 15

4 Error estimates **20**
4.1 Continuous framework . . . 20
4.2 Discrete framework . . . 22

5 Dealing with Improved Kirchhoff conditions **24**
5.1 Improved Kirchhoff conditions . . . 24
5.2 Discretization . . . 26
5.3 Implementation . . . 27

Conclusion **30**



Unité de recherche INRIA Rocquencourt
Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-0803