

GO++ : A modular Lagrangian/Eulerian software for Hamilton Jacobi equations of Geometric Optics type

Jean-David Benamou, Philippe Hoch

▶ **To cite this version:**

Jean-David Benamou, Philippe Hoch. GO++ : A modular Lagrangian/Eulerian software for Hamilton Jacobi equations of Geometric Optics type. [Research Report] RR-4409, INRIA. 2002. inria-00072179

HAL Id: inria-00072179
https://inria.hal.science/inria-00072179

Submitted on 23 May 2006

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

GO++ : A modular Lagrangian/Eulerian software for Hamilton Jacobi equations of Geometric Optics type

Jean-David Benamou — P. Hoch

N° 4409

février 2002

THÈME 4



ISSN 0249-6399 ISRN INRIA/RR--4409--FR+ENG

INRIA
ROCQUENCOURT

GO++ : A modular Lagrangian/Eulerian software for Hamilton Jacobi equations of Geometric Optics type

Jean-David Benamou , P. Hoch *

Thème 4 — Simulation et optimisation
de systèmes complexes
Projet OTTO

Rapport de recherche n° 4409 — février 2002 — pages

Abstract: We describe both the classical Lagrangian and the Eulerian methods for first order Hamilton-Jacobi equations of geometric optic type. We then explain the basic structure of the software and how new solvers/models can be added to it. A selection of numerical examples presented.
Keywords : Hamilton-Jacobi,Hamiltonian System, Ray Tracing, Viscosity Solution, Upwind Scheme, Geometric Optics, C++

Key-words: No keywords

* INRIA, Domaine de Voluceau, B.P.105 78153 Le Chesnay cedex, France, e-mail :Jean-David.Benamou@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

GO++ : Une librairie modulaire pour la resolution de problème Lagrangien/Eulerien de type optique géométrique

Résumé : La méthode Lagrangienne classique et la méthode Eulérienne de résolution des équations du premier ordre de Hamilton-Jacobi de type optique géométrique sont présentées. Nosu expliquons ensuite la srtucture de base du logiciel et comment il peut évoluer pour incorporer de nouveaux modéles et de nouvelles applications. Une selection d'examples numériques est présentée.

Mots-Clés : Hamilton-Jacobi, Système Hamiltonien, Lancer de rayons, Solution de Viscosté, Schéma Décentrés, Optique Géométrique, C++

Mots-clés : Pas de motclef

Contents

1 Introduction **4**

2 The models **5**
2.1 The Lagrangian solution and the ray method 5
2.2 The Eulerian solution and the Hamilton-Jacobi partial differential equation 5
2.3 Domain and Boundary conditions 5
2.4 Initialization 6

3 The main structure of GO++ **6**

4 Domain, discretization and data structure **7**
4.1 Domain : the *GEOM* class 7
4.2 Discretization : the *DSCR* class 7
4.3 Data structure : the *ISH* class 7

5 Describing functions of the configuration (x) space : the *XFCT* class **8**

6 The model **10**
6.1 Boundary conditions : the *BCON* class 10
6.2 Hamiltonian function : the *HAMI* class 11

7 Numerical methods **12**
7.1 Space solvers, the *HAMI_NUM* and *GRAP* classes 12
7.2 Time solver, the *TSLV* class 15

8 Defining the type of application : the *APLI* class **15**

9 The main program : a first example **15**
9.1 Results 17

10 A second example : a waveguide problem **18**
10.1 Creating a new Hamiltonian class 18
10.2 Implementing a new numerical Hamiltonian 20
10.3 Changing the initialization 21
10.4 Changing the boundary conditions 21
10.5 Results 22

11 A generalization to semi-analytically defined Hamiltonians **23**
11.1 Interpolation methods : the *XFCT_INTR* and the *INTR* classes 23
11.2 Result 24

12 A new application : the Stationary problem **24**
12.1 The Algorithm 24
12.2 The GO++ implementation 24
12.3 results 26

13 Outputs **27**
13.1 Basic « overloading 27
13.2 Plotmtv files 27

1 Introduction

Ray tracing is a widespread numerical technique. Its is, for instance, routinely used in the oil industry to compute travel-times of seismic waves in the underground under the geometric optics approximation of wave propagation. Geometric optics is also a prototype problem for first order Hamilton-Jacobi equations when the Hamiltonian function enjoys convexity properties. This class of problems appears in rather diverse applications such as wave propagation, mesh generation, combustion, crystal growth and more generally in a large number of problems of the calculus of variations.

Ray tracing consists in solving a set of ordinary differential equations called the Hamiltonian systems. One of the variable describes a Lagrangian trajectory (the ray) along which the others, the Lagrangian variables, are transported. This approach is conceptually simple and numerically easy to use. However, as a Lagrangian method, it does not give any control on the spatial resolution of the Lagrangian solution. Indeed, the rays are necessarily in finite numbers and produce when diverging poor resolution. Various refinement procedures exist to fix this problem [20] [16] which are based on interpolation techniques.

It the Eulerian approach the main variable, defined on the configuration space, satisfies a Hamilton-Jacobi partial differential equation. It can be discretized on an a-priori fixed grid. The space resolution of the numerical method is therefore automatically maintained and the accuracy of the approximation depends on this resolution. These two approaches, Lagrangian and Eulerian, are mathematically equivalent only when the Lagrangian solution is well defined and single-valued (rays do not cross). The Eulerian method otherwise produces a weak "viscosity" solution which can be identified as the minimum value of the multi-valued Lagrangian solution (the problem of computing the Lagrangian solution using a Eulerian representation when it is multi-valued has been the subject of on-going work in the last ten years ([19] [10] [5] [1] [18] [7] [11] [6] [14] ...).

We review here the basic framework of the problem, introducing both Lagrangian and Eulerian solutions and their relations. Next we detail the structure of our software which is intended to provide a C++ framework for an easy implementation, use and comparison of both Lagrangian and Eulerian numerical methods applied to Hamilton-Jacobi equations. We explain then how GO++ can be used to solve a standard Geometric Optics problem and modified to accommodate for a general Hamiltonian. Finally we give indications on how a new solver can be added to this modular software and again give an example. Even though large parts of GO++ are documented here, this paper is not a manual or a tutorial but rather a description of the choices we made to build the software.

We would like to point out that one of our purposes is to explore the use of C++ for providing an efficient open source library for education and research. Scientific computing nowadays often combines several numerical techniques and makes software development rather difficult and time consuming. Instead of programming from scratch, meta-languages (such as Matlab) allowing a fast and easy (bug free) use of standard numerical algorithms are popular. However, theses libraries are too large to be fast enough for 2-D and 3-D "realistic" applications. A more efficient general scientific solver like FreeFem +
(`http://www-rocq.inria.fr/Frederic.Hecht/FreeFemPlus.htm`) uses a C++ idiom as a wrapper. We want to simplify further the interface and we believe that it is possible to directly use the C++ language. We think that a C++ modular library which targets a particular class of application can be made "user-friendly" for researchers (used to programming but not necessarily C++ experts) without any sort of "wrapping". The expected gain is the efficiency of the produced compiled code in term of running time.

2 The models

2.1 The Lagrangian solution and the ray method

A Hamiltonian function $H(s,y,p)$ is given, defined on $\mathbb{R}_s^+ \times \mathbb{R}_y^2 \times \mathbb{R}_p^2$ (we will generically set $y=(y_1,y_2)$, $p=(p_1,p_2)$ ...); $\mathbb{R}_s^+ \times \mathbb{R}_y^2$ is the time-space configuration in which rays can evolve and $\mathbb{R}_s^+ \times \mathbb{R}_y^2 \times \mathbb{R}_p^2$ is called the phase space.

The Lagrangian method consists in solving (using an ODE solver) the Hamiltonian system formed by the following set of ordinary differential equations (ODEs) [21] [3] [13] [9] :

$$\begin{cases} \dot{y}(s,y^0) = H_p(s,y(s,y^0),p(s,y^0)),\ y(0,y^0) = y^0, \\ \dot{p}(s,y^0) = -H_y(s,y(s,y^0),p(s,y^0)),\ p(0,y^0) = \phi^0_{y^0}(y^0), \\ \dot{\varphi}(s,y^0) = p(s,y^0)\cdot H_p(s,y(s,y^0),p(s,y^0)) - H(s,y(s,y^0),p(s,y^0)), \\ \varphi(0,y^0) = \phi^0(y^0). \end{cases} \tag{1}$$

The dot stands for derivation with respect to "time" $\dot{(.)} = \frac{d(.)}{ds}$, $g_x(x_1,x_2)$ denotes the gradient with respect to $x=(x_1,x_2)$, ϕ^0 is a given initial phase and also appears in the initial condition for p. For each $y^0 \in \mathbb{R}_y^2$ (or a subset of $\mathbb{R}_y^2$), the system generates a "bicharacteristic strip" $(y(s,y^0),p(s,y^0))$ depending on s and y^0. The projections of the strips onto $\mathbb{R}_y^2$: $y(s,y^0)$, are called the rays. Each ray is therefore "labeled" by its initial position y^0. The phase $\varphi(s,y^0)$ is transported by the corresponding ray $y(s,y^0)$ and, when rays are crossing, is a multi-valued function of the configuration space $\mathbb{R}_s^+ \times \mathbb{R}_y^2$.

2.2 The Eulerian solution and the Hamilton-Jacobi partial differential equation

As we proceed along the ray and as long as rays do not cross, we can apply a local inversion theorem to the mapping $y^0 \to y(s,y^0)$. Assuming that every point $(s,x) \in \mathbb{R}_s^+ \times \mathbb{R}_y^2$ is reached by only one ray, we can introduce the Eulerian variable $\phi(s,x)$ which, evaluated at the Lagrangian coordinates specified by the rays, matches the Lagrangian phase :

$$\phi(s,y(s,y^0)) = \varphi(s,y^0). \tag{2}$$

It is possible to derive a Hamilton-Jacobi equation satisfied by $\phi(s,x)$ either in the classical sense when rays do not cross or in the viscosity sense [8] [4] else

$$\begin{cases} \frac{\partial \phi}{\partial s}(s,x) + H(s,x,\nabla\phi(s,x)) = 0,\ for\ (s,x) \in \mathbb{R}_s^+ \times \mathbb{R}_y^d \\ \phi(0,y^0) = \phi^0(y^0),\ for\ y^0 \in \mathbb{R}_y^d \end{cases} \tag{3}$$

A link can be made between Lagrangian and Eulerian viscosity solution using the theory of optimal control [6]. The viscosity solution can be characterized as the minimum of the associated Lagrangian phases :

$$\phi(s,x) = \min_{y^0,\ s.t.\ y(s,y^0)=x} \varphi(s,y^0). \tag{4}$$

If there is a zone where no ray penetrates, the viscosity solution implicitly generates "non-classical" rays to fill this empty zone. It means that the optimal curves will satisfy the Hamiltonian system (1) with initial conditions different from those specified for classical rays. This situation seems to be linked to diffraction phenomena and should hopefully be investigated elsewhere.

2.3 Domain and Boundary conditions

In practice, computations are limited to a given bounded domain and we must enforce boundary conditions. The simplest condition for rays is to ignore them as soon as they exit the domain.

It corresponds to an "out-going" boundary condition for the Hamilton-Jacobi equation. It is implemented by the enforcement of the continuity of the flux across the boundary (for instance for Lax-Friedrich solvers) or more simply, in the case of Godunov solver by the Dirichlet boundary condition $\phi = +\infty$ (in practice $\phi = M$, M large enough) (Soner condition). More generally, we want to be able to treat general Dirichlet, Neumann or mixed boundary conditions which can be for instance interpreted in the Lagrangian case as reflecting laws for the rays (see [4] for more on boundary conditions).

2.4 Initialization

The simplest Eulerian initialization (the default initialization) is $\phi^0(y^0) = 0$ in (3). In term of rays, it means that they start with a direction $p = 0$ from all points of the domain.

An other common initialization for ray tracing is the isotropic point source. All rays start at a source point S with direction p on the unit circle. We can use a Eulerian initialization using the distance function from S : $\phi^0(y^0) = \|S - y^0\|$ to obtain a Eulerian emulation of the Lagrangian isotropic source initialization.

3 The main structure of GO++

The computational kernel of GO++ is more general that is needed for our purpose. It is designed to solve a set of ordinary differential equations

$$\dot{X}(s) = RHS(X(s)) \tag{5}$$

where $X(s)$ is a collection of variables and RHS a function depending on the Hamiltonian, the boundary conditions and the discretization. We recall that GO++ must allow Lagrangian and/or Eulerian computations. We therefore decided to define a unique data structure in a class called ISH for solving (5). A member of this class X represents either a set of bicharacteristics and associated phases : $(X.y, X.p, X.phi) = (y, p, \varphi)$ at a fixed time s and (5) represents the Hamiltonian system (1) or, in the Eulerian framework, $X.y = x$ stands for the discretization points while $(X.p, X.phi) = (\nabla\phi(s,x), \phi(s,x))$ are respectively the gradient of the phase and the phase itself at those points. In this case the equation satisfied by $X.phi$ in (5) corresponds to the Hamilton-Jacobi equation (3).

This approach is obviously not optimal in the Eulerian case alone because the remaining equations in (5) are not a priori needed and the data could resume to a simple scalar function ϕ. In our opinion these drawbacks are compensated by the following advantages :

- This setting allows to define a unique class describing the Hamiltonian in which a function called $Hxph$ maps the ISH data X described above to another ISH data Y which stands for $(H_y(s,y,p), H_p(s,y,p), H(s,y,p))$. At a fixed time s, the same data structure can be used for X and Y.
- The Hamiltonian function may be straightforwardly generalized to depend also on ϕ : $H(s,y,p,\phi)$.
- The time solver modules used to solve (5) can be applied in Lagrangian and Eulerian modes.
- In the Eulerian mode, $y = x$ and $p = \nabla\phi(s,x)$; the additional quantities $(H_y(s,x,\nabla\phi(s,x))$, $H_p(s,x,\nabla\phi(s,x))$ are needed to implement some Hamilton-Jacobi solvers.
- The data $X.y$ representing the grid points and the associated equation in (5) could be used to implement a moving or adaptative grid.

The software consists of independent modules in the the form of C++ classes. Those of these classes which prescribe the problem or the numerical methods are virtual classes containing virtual functions. Adding new problems or new numerical methods to GO++ consists in writing or modifying existing children classes of the virtual classes. We now describe the basic classes needed to get started.

4 Domain, discretization and data structure

The software is for now restricted to a rectangular 2-D spatial domain and regular Cartesian discretizations. This choice simplifies the implementation of finite difference solvers.
In order to generalize the software to treat 3-D problems or unstructured grid, the class presented in this section must be modified. Even though it may be possible to treat such generalization by creating virtual geometry, discretization and data classes, we believe that a better (and simpler) solution is to create several versions of the software, each one dedicated to a particular discretization and to 2D or 3D problems.

4.1 Domain : the $GEOM$ class

The $GEOM$ class describes the geometry of the problem. It is used to define rectangular domains.

The declaration $GEOM\ geometry(x_{1,min}, x_{1,max}, x_{2,min}, x_{2,max})$ creates a regular 2-D rectangular geometry by means of the x_1 and x_2 coordinates of its corners :
$x_{1,min}, x_{1,max}, x_{2,min}, x_{2,max}$.

4.2 Discretization : the $DSCR$ class

The $DSCR$ class describes the discretization of the problem. We will consider that in the Eulerian framework the discretization stands for the set of grid points at which the solution will be computed. In the Lagrangian mode, the discretization will be the set of initial points for the rays. The $DSCR$ needs a $GEOM$ class to build this set of points. The accuracy of the Eulerian solution depends on the number of discretization points and also on the expected "order" of accuracy of the numerical methods to be used. An integer ord specify the order and sets the number of points of the external layer used for computing finite differences at the boundary.

The declaration $DSCR\ discretization(geometry, nx_1, nx_2, ord)$ can be used both in Lagrangian and Eulerian mode. It specifies the number of discretization points (nx_1, nx_2) in each coordinate direction. It builds the regular 2-D cartesian grid of points $(x_{1,i}, x_{2,j}), i = 1, .., nx_1 + 2 * ord, j = 1, .., nx_2 + 2 * ord$ needed to implement a finite difference method of order ord with $nx_1 \times nx_2$ points covering the domain specified by the GEOM class or one of its children (note that the actual domain is spanned by the set of points $(x_{1,i}, x_{2,j}), i = ord, .., nx_1 + ord - 1, j = ord, .., nx_2 + ord - 1$).

In the Lagrangian mode, it determines the size of the arrays which will describes number of rays we use $(nx_1 \times nx_2)$. It can also be used for a default initialization of the initial points $y^0 = (x_{1,i}, x_{2,j}), i = 1, .., nx_1, j = 1, .., nx_2$. The general initialization process is described in section 5.

4.3 Data structure : the ISH class

The discretization class provides the necessary information for defining the data structure of a member X of ISH. It will typically be three arrays which numbers of dimensions and sizes are determined by the discretization of the problem.

The basic data structure is made of two $2 \times nx_1 \times nx_2$ arrays of reals called $X.y$ and $X.p$ and one $nx_1 \times nx_2$ array of reals called $X.phi$.

ISH is built on top of a collection of freely available array classes : *RNM*
(see `http://www-rocq.inria.fr/Frederic.Hecht`). These classes are much simpler than other packages (such as Blitz), easy to compile and the functionalities (vectorial Matlab-like operations, scalar products ...) sufficient for our purpose.
ISH also contains utility member functions for reading and writing at any time step s either the collection of $(y^0, \varphi_x^0(s, y^0), \varphi^0(y^0))$ for the rays or of the Eulerian variables $(x, \nabla\phi(s,x), \phi(s,x))$. It uses the overload of "<<" (used for the output) in the *RNM* class.

When an *ISH* variable is declared, the default initialization is $X = 0$. There is however the possibility of specifying a function ϕ^0 in the initialization list. The function maps an array X describing either y^0 or x into an ISH Y for which you have to specify (Y.y=X of course) $Y.phi = f(X)$ and $Y.p = f_x(X)$ (see next section).

5 Describing functions of the configuration (x) space : the *XFCT* class

We first need a virtual class *XFCT* which describes a function of the configuration space. This class has a virtual function called F which takes an ISH variable FX for which FX.y is an $2 \times n_{x_1} \times n_{x_2}$ array describing a set of points in the domain. The remaining variables $FX.p = \nabla F(X)$ is the gradient of the function at these points and $FX.phi = F(X)$ the function itself.

The class has two families of children in which the *XFCT* can be specialized. In the *XFCT_ANAL* class the function is specified analytically. In the second class *XFCT_INTR*, the function is specified by its evaluation at given configuration points. To build such an object, we must read data in the form of two arrays (the specification of sample points and the value of the function at these points) and we need an interpolation method to be able to evaluate the *XFCT* on any array of points (more details on this class in section 11.1.)

As an example, suppose we want to initialize a Lagrangian problem where all the rays start at a source point S, with initial direction on the sphere. We need to define an *XFCT* child class called *XFCT_INIT_LAG_SRCP* for which S is given as a data member.

```
class XFCT_INIT_LAG_SRCP : public XFCT
{
 public:
  XFCT_INIT_LAG_SRCP(point& S, R anglmin=0.,R anglmax=2.*pi, R inds=1, int nfdism=1) : XFCT(nfdism),
  ~XFCT_INIT_LAG_SRCP(){ }

  //* point = 2 reals
  point Source;

  //* shooting angle oriented from anglemin to anglemax
  R angmax;
  R angmin;
  //* value of the index at source point
  R indS;

  void F(ISH* FX) const;
};
```

Then the F function, is specialized in the following C++ source :

```
//* specializes the F function of the new class
```

```
void XFCT_INIT_LAG_SRCP ::F(ISH* FX) const
{
  for (int i=FX->discr.ord;i<FX->discr.nx+FX->discr.ord;i++)
    {
      for (int j=FX->discr.ord;j<FX->discr.ny+FX->discr.ord;j++)
        {
          // All rays at the source point
          FX->y(0,i,j)=S.x;
          FX->y(1,i,j)=S.y;

          // With an equidistributed shooting angle
          R ang(angmin+(angmax-angmin)*(j+FX->discr.ny*i)/(FX->discr.nx*FX->discr.ny-1.));

          FX->p(0,i,j)=cos(ang)/indS;
          FX->p(1,i,j)=sin(ang)/indS;
        }
    }

  // All with a phase equal to zero
  FX->phi=0.;
}
```

As discussed in section 2.4, the corresponding behavior in the Eulerian mode can be emulated by the initial function $\phi^0(X) = \|S - X\|$. The implementation of this function is similar. We create the child $XFCT_INIT_EUL_SRCP$ for which S is given as a data member.

```
class XFCT_INIT_EUL_SRCP : public XFCT
{
 public:
  XFCT_INIT_EUL_SRCP(point& S, int nfdism=1) : XFCT(nfdism),Source(S) {}
  ~XFCT_INIT_EUL_SRCP(){ }

  // point = 2 reals
  point Source;

  void F(ISH* FX) const;
};
```

Then the F function, is specialized as in the following C++ source :

```
//       specializes the F function of the new class
void XFCT_INIT_EUL_SRCP ::F(ISH* FX) const
{
 // Initialization of the position of the unknown at the grid points
 for (int i=0;i<FX->discr.nx+2*FX->discr.ord;i++)
    {
      FX->y(0,i,'.')=FX->discr.xi(i);
    }

  for (int j=0;j<FX->discr.ny+2*FX->discr.ord;j++)
    {
      FX->y(1,'.',j)=FX->discr.yj(j);
    }

  // Computation of euclidean distance between
```

```
  // grid points and the source point (result in FX->phi)

  distE_PtPts(FX->discr,S,FX->phi);

  FX->p=0.;
}
```

(where we omitted the function `distE_PtPts`).

6 The model

6.1 Boundary conditions : the $BCON$ class

The boundary condition class can be used either to implement classical boundary conditions in the Eulerian case or to specify the behavior of rays when the limit of the computational domain is reached. The virtual class $BCON$ has a virtual member function $Sbco$ which uses information from the $DSCR$ class (discretization) to modify an ISH variable. In the Lagrangian mode, $Sbco$ modifies the ISH standing for the RHS in (5) for those of the rays which exit the domain. When solving a Eulerian problem, boundary conditions are implemented by filling the boundary layer (one or more points) with correct values of the ISH representing the current solution. The width of the boundary layer is set by the order *ord*.

The simplest boundary conditions are "out-going" boundary conditions as described in section 2.3. It is implemented in the child classes $BCON_LAG_OUTG$, $BCON_EUL_OUTG_LAXFR$ for Lax-Friedrich solvers and $BCON_EUL_OUTG_GODUNOV$ for Godunov solvers.

```
// Here, X represents the solution, and Y the flux.
void BCON_LAG_OUTG::Sbco(const R& t, ISH* X,ISH& Y) const
{

  for (int i=0;i<X->discr.nx;i++)
    {
      for (int j=0;j<X->discr.ny;j++)
        {
         //* If the ray exit the domain , the whole computation is stopped
          if (X->y(0,i,j)<=X->discr.geom.xmin+1e-6 ||
              X->y(0,i,j)>=X->discr.geom.xmax-1e-6 ||
              X->y(1,i,j)<=X->discr.geom.ymin+1e-6 ||
              X->y(1,i,j)>=X->discr.geom.ymax-1e-6)
            {
         //*     Set  RHS=0
              Y.y('.',i,j)=0.;
              Y.p('.',i,j)=0.;
              Y.phi(i,j)=0.;
            }
        }
    }
}
```

and in the Eulerian mode (for Godunov type solvers)

```
/**
outgoing BC for hami. num  GODUNOV
Ghost points set to + infty
*/
```

```
void BCON_EUL_OUTG_GODUNOV::Sbco(const R& t, ISH* X,ISH& Y) const
{
const R BigVal(1e20);

for (int i=1;i<=X->discr.ord;i++)
  {
    X->phi(X->discr.ord-i,'.')=BigVal;
    X->phi(X->discr.nx+X->discr.ord+i-1,'.')=BigVal;
    X->phi('.',X->discr.ord-i)=BigVal;
    X->phi('.',X->discr.ny+X->discr.ord+i-1)=BigVal;
  }
}
```

6.2 Hamiltonian function : the $HAMI$ class

The Hamiltonian function $H(t, y, p)$ is the central object of the problem and is represented through a virtual class $HAMI$. As discussed in section 3, a virtual function $Hxph$ maps an ISH data X to another ISH data Y. The $HAMI$ hierarchy of class has two main branches.

In the first branch called $HAMI_CONT$, the $(X.y, X.p, X.phi)$ variables are the set of bicharacteristics and phase at a fixed "time"s and the Y data stands for the corresponding set of "continuous" evaluations of $(H_y(s, X.y, X.p), H_p(s, X.y, X.p), H(x, X.y, X.p))$ needed to solve the Hamiltonian system (1) . It can be straightforwardly used for Lagrangian computations.

In the Eulerian case, the space discretization of the Hamilton-Jacobi equation needs a second branch called $HAMI_NUM$ described in section 7.1.

One of the simplest H-J function is analytically specified by $H(s, y, p) = 0.5 * \|p\|^2$. It is contained in the child class $HAMI_CONT_ANAL_P2$. It is implemented in the following piece of C++ header

```
class HAMI_CONT_ANAL_P2 : public HAMI_CONT
{
 public:
  HAMI_CONT_ANAL_P2() : HAMI_CONT() {}
  ~HAMI_CONT_ANAL_P2(){}

  void SupGrH(const Rnmk& X, Rnmk& SupgradH) {SupgradH=1.;} ;
  void Hxph(const R& t, const ISH* X, ISH& Y) ;
};
```

and the code

```
/**
   Analytic Hamiltonian H(x,p)=|p|^2/2
*/
void HAMI_CONT_ANAL_P2 ::Hxph(const R& t, const ISH* X, ISH& Y) const
{
  //* dH/dp=p
  Y.p=X->p;

  //* dH/dy=0
  Y.y=0.;

  for (int i=X->discr.ord;i<X->discr.nx+X->discr.ord;i++)
    {
      for (int j=X->discr.ord;j<X->discr.ny+X->discr.ord;j++)
```

```
        {
          //* H=.5*(p,p)   (RNM scalar product)
          Y.phi(i,j)=((X->p('.',i,j),X->p('.',i,j));
          Y.phi(i,j)/=2.;
        }
    }
```

7 Numerical methods

7.1 Space solvers, the $HAMI_NUM$ and $GRAP$ classes

This section only concerns the Eulerian approach even though one could think of further development in the Lagrangian case which would require an enrichment of the $HAMI$ structure (symplectic solvers for instance).

In the Eulerian framework, the ISH data $(X.y, X.p)$ stands for $(x, \nabla\phi(s,x))$ and we need to specify a method for approximating the gradient of the phase, i.e to numerically implement the affectation $X.p = \nabla\phi(s,x)$. This is the object of the $GRAP$ class. As "upwinding"is a classical tool in this context a member function $Grphi$ takes the $nx_1 \times nx_2$ array representing the phase $X.phi$ as an input and returns two $2 \times nx_1 \times nx_2$ arrays $Grphim = (a,c)$ and $Grphip = (b,d)$ which are the discrete upwind derivatives of order ord : if $ord = 1$ and $\phi(s, x_{1,i}, x_{2,j}) \simeq \phi_{i,j}$ then $a = \frac{(\phi_{i,j} - \phi_{i-1,j})}{\Delta x_1}$, $b = \frac{(\phi_{i+1,j} - \phi_{i,j})}{\Delta x_1}$, $c = \frac{(\phi_{i,j} - \phi_{i,j-1})}{\Delta x_2}$, $d = \frac{(\phi_{i,j+1} - \phi_{i,j})}{\Delta x_2}$..

The simplest child class of $GRAP$ is $GRAP_UPW1$ which computes upwind finite difference of order 1 (the (a,b,c,d) quantities above). Other types of approximation such a TVD or ENO are available.

The discretization of (3) may also require some modifications of the Hamiltonian function itself. We therefore created a virtual child class of $HAMI$ called $HAMI_NUM$ which uses the specifications of a given $HAMI_CONT$ Hamiltonian function and a method of approximation of $\nabla\phi$ (a $GRAP$ class) to evaluate its own "numerical" $Hxph$ function.

A simple method is the Lax-Friedrichs solver which replaces $H(s,x,\nabla\phi(s,x))$ by

$$H^{num}_{LF}(s,x,a,b,c,d) := H(s,x,\frac{a+b}{2},\frac{c+d}{2}) - \frac{\alpha(x)}{2}(b-a) - \frac{\beta(x)}{2}(d-c), \tag{6}$$

where (a,c) resp. (b,d) are defined above, is designed by the $HAMI_NUM_LAXF$ class. The coefficients $\alpha(x) := \sup_{(u,v)} |\frac{\partial H}{\partial u}|$, and $\beta(x) := \sup_{(u,v)} |\frac{\partial H}{\partial v}|$ are the maximum of the speeds ((u,v) stands for the (ϕ_{x_1}, ϕ_{x_2}) variables). They are computed in a function called $SupGrH$ which is a virtual member of the $HAMI_NUM$ class.

The $HAMI_NUM$ and $HAMI_NUM_LAXF$ class of $HAMI$ look like:

```
class HAMI_NUM : public HAMI
{
  public:
    HAMI_NUM(HAMI_CONT &H, const GRAP& ApGrad) : HAMI(H.nf),Hamc(H), ApxGrad(ApGrad)
  {
    Grphip=new Rnmk(2,ApGrad.discr.nx+2*ApGrad.discr.ord,ApGrad.discr.ny+2*ApGrad.dis
cr.ord);
    Grphim=new Rnmk(2,ApGrad.discr.nx+2*ApGrad.discr.ord,ApGrad.discr.ny+2*ApGrad.dis
cr.ord);
    Xmod=new ISH(ApGrad.discr);
```

```
  }

  virtual ~HAMI_NUM(){delete Grphip;delete Grphim;delete Xmod;}

  //* The continuous hamiltonian that we have to discretized
  HAMI_CONT &Hamc ;

  //* The type of approximation of the gradients (TVD, ENO ...)
  const GRAP &ApxGrad ;

  //* current upwind derivatives
  Rnmk* Grphip;
  Rnmk* Grphim;

  //* pointer on modified solution (only the gradient is different)
  ISH* Xmod;

  //*            ACCESS FUNCTION TO FX
  //*           -----------------------
  //* The access function to FX is overloaded,
  //* it permits to use the SAME FX variable in the
  //* Numerical Hamiltonian and the continuous one,
  //* but only the latter has got a real meaning.

  ISH*& fX() {return Hamc.fX();}


  //*           MODIFICATION FUNCTION TO FX
  //*           ---------------------------
  //* In the case of a Numerical Hamiltonian
  //* we don't re compute the X fields (the grid is fixed).
  //* here FX is set to GridVal (fixed grid and values on it) at
  //* the level of rhs in the application, then here, it doesn't
  //* do anything.

  inline void xFHami(ISH* FX) {}

  //* The SupGrH is overloaded like the access function fX() to
  void SupGrH(const Rnmk& X, Rnmk& SupgradH) {Hamc.SupGrH(X,SupgradH);};

};
```

then we can specified the Lax-Friedrichs as a child class of $HAMI_NUM$

```
class HAMI_NUM_LFX  : public HAMI_NUM
{
 public:
  HAMI_NUM_LFX(HAMI_CONT &H,const GRAP& ApGrad) : HAMI_NUM(H,ApGrad)
{
 // Compute and stores maximum speeds for the hamiltonian

    Hamc.fX()=new ISH[Hamc.nf](ApGrad.discr,0.);
    Hamc.SupGRADH()=new Rnmk(2,ApGrad.discr.nx+2*ApGrad.discr.ord,ApGrad.discr.ny+2*ApGrad.discr.ord
    Hamc.SupGrH(Hamc.fX()[0].y,*Hamc.SupGRADH());
    delete[] Hamc.fX();
```

```
}

  ~HAMI_NUM_LFX(){delete Hamc.SupGRADH();}

  void Hxph(const R& t, const ISH* X, ISH& Y) ;
};
```

and the code for the numerical hamiltonian

```
/**
  Lax-Friedrichs numerical Hamiltonian
  Specialization of Hxph
*/
void  HAMI_NUM_LFX::Hxph(const R& t, const ISH* X, ISH& Y) const
{
  //* The modified solution is set to the solution
  //* Below, we change his gradient (Xmod->p) ...

  *Xmod=*X;

  // unpwind derivatives
  ApxGrad.Grphi(X->phi,*Grphim,*Grphip);

  /**
   Centered gradient
  */

  for (int i=X->discr.ord;i<X->discr.nx+X->discr.ord;i++)
    {
      for (int j=X->discr.ord;j<X->discr.ny+X->discr.ord;j++)
        {
          for (int d=0;d<2;d++)
            {
              Xmod->p(d,i,j)=(*Grphip)(d,i,j)+(*Grphim)(d,i,j);
              Xmod->p(d,i,j)/=2.;
            }
        }
    }
  //*   evaluate the Hamiltonian function with approximate gradient
  Hamc.Hxph(t,Xmod,Y) ;

  //* Artificial viscosity
  //* Compute de maximum of the speed.

  for (int i=X->discr.ord;i<X->discr.nx+X->discr.ord;i++)
    {
      for (int j=X->discr.ord;j<X->discr.ny+X->discr.ord;j++)
        {
          Y.phi(i,j)-=((*Hamc.SupGRADH())(0,i,j)*((*Grphip)(0,i,j)-(*Grphim)(0,i,j))
                       +(*Hamc.SupGRADH())(1,i,j)*((*Grphip)(1,i,j)-(*Grphim)(1,i,j)))/2.;
        }
    }
}
```

7.2 Time solver, the $TSLV$ class

Time solvers are designed to solve the set of ODE's (5) satisfied by an ISH unknown that has to be declared and initialized (see the ISH section). It also assumes that a particular Rhs function is identified through the declaration of the type of application (see the $APLI$ section below). The $TSLV$ virtual class has a virtual member function $Tstep$ which solves (5) on a time step dt (e.g. it takes the ISH unknown $X(t)$ and the scalar dt and return $X(t+dt)$. We already mentioned the structure of GO++ makes it possible to use time solver for Lagrangian or Eulerian problems. The discretization is passed to time solver first for optimization purposes. It also provides the possibility to modify the discretization in time.

The simplest time solver is the Euler method $X(t+dt) = X(t) + dt * RHS(X(t))$ and is implemented in the child class $TLSV_EUL$. Several other methods of Runge-Kutta type are also available.

8 Defining the type of application : the $APLI$ class

Once a discretization and a model is fixed, an ISH variable X can be declared which represents the solution of the problem at a fixed time. It is then necessary to specify the type of application (e.g. Lagrangian or Eulerian) we wish to solve. It is done through the specialization of the Rhs function on the right hand side of (5).

Thus there exist a virtual class $APLI$ of which Rhs is a virtual member function. It can be specialized into (at least) two child classes $APLI_LAG$ and $APLI_EUL_EV$ for Lagrangian and Eulerian evolution applications. The function Rhs of course needs information on the model characterized by an Hamiltonian function $Hxph$ and boundary conditions $BCON$. In the Lagrangian mode the boundary conditions are enforced (calling the $Sbco$ function) after the evaluation of Rhs while in the Eulerian mode it is before (see section 5.1).

9 The main program : a first example

This section shows how all these objects can be combined to produce a simple Lagrangian/Eulerian computation for the Hamiltonian $H(s,y,p) = 0.5 * \|p\|^2$ on a rectangular domain with outgoing boundary conditions, cartesian discretization and point source initialization. This following source code is a prototype main program for GO++ applications. Changes in domain, discretization, initialization, hamiltonian functions, numerical methods ..., can be implemented separately in child classes of the basic classes. The following sections detail such changes.

```
#include <iostream>
#include <fstream>
#include <assert.h>
#include <math.h>
#include <utility>
#include <algorithm>
#include <GO++.hpp>

/*
  A typical main program for
  eulerian and lagrangian computations
*/

int main(int argc, char **argv)
{
```

```
/* *******  Domain and discretization  *******
   define a 2D rectangular domain with (x,y)
   coordinates of the corners
*/
R xmin(0.),xmax(4.),ymin(0.),ymax(2.);
GEOM geom(xmin,xmax,ymin,ymax);

//  Define a discretization of the domain
//  Eulerian case
int nxE(51),nyE(51),ord(1);
DSCR discrE(geom,nxE,nyE,ord);

//  Lagrangian case
int nxL(5),nyL(5);
DSCR discrL(geom,nxL,nyL);

/* ***********      model      ************
 */

// Boundary Conditions: Eulerian Case
BCON_EUL_OUTG_LAXFR bcond;

// Boundary Conditions: Lagrangian Case
BCON_LAG_OUTG bclag;

// Hamiltonian
HAMI_CONT_ANAL_P2 ham;

/* ****    Numerical methods in space for HJ eq. **** */

// Type of approximation of grad phi
GRAP_UPW1 gradap(discrE);

// Numerical Hamiltonian (space) Lax-Friedrich
HAMI_NUM_LFX hamnum(ham,gradap);

// ***** type of application *****
//       define type of RHS

// Eulerian Evolution application
APLI_EUL_EV eul(hamnum,bcond);

// Lagrangian application
APLI_LAG lag(ham,bclag);

// ****** define and initialize by XFCT *****
// Source Point
point S;
S.x=(xmax+xmin)/2.;
S.y=(ymax+ymin)/2.;

// Eulerian Initialization (source point)
XFCT_INIT_EUL_SRCP phi0(S);
ISH* X_eul= new ISH(discrE,phi0);
```

```
  // Lagrangian Initialization (source point)
  XFCT_INIT_LAG_SRCP xsi0(S);
  ISH* X_lag= new ISH(discrL,xsi0);

  // **** time solver define TSTEP ****
  // euler for Eulerian
  TSLV_EUL tseul(eul,discrE);
  // RK4 for Lagrangian
  TSLV_RK4 tslag(lag,discrL);

  // Initial / Final time / constant time step
  R t(0),t1(3.4),dt(1e-3);

  // counter to write solutions at every nexec times step
  int cp(0),nexec(10);

  ofstream solLag("SOL_lag.dat");
  ofstream solEul("SOL_eul.dat");

  while (t<t1)
    {
      cout << " t=" << t << "\n";
      cp++;

      // Evolution of Lagrangian Sol over one time step
      tslag.Tstep(t,dt,X_lag);

      // Evolution of Eulerian Sol over one time step
      tseul.Tstep(t,dt,X_eul);

      t+=dt;

      // Every nexec time step, write data into files
      if (fmod(cp,nexec)==0)
        {
          cp=0;
          //* write the array of ray positions at time t
          solLag << t << endl << X_lag->y << endl;
          //* write the array of phases on the Eulerian grid at time t
          solEul << t << endl << X_eul->phi << tendl;
        }
    }

}
```

9.1 Results

The parameters are $x1_{min} = 0, x1_{max} = 4, x2_{min} = 0, x2_{max} = 2$. The source point S is located at the center of the domain. In the Eulerian case, we took $nx_1^e = nx_2^e = 51$, $ord = 1$, and for the Lagrangian problem $nx_1^l = nx_2^l = 5$ which gives 25 rays. We evolve the two problems simultaneously until time $tmax = 3.4$ with a constant time step 0.01. The compile time takes about 20 sec, and the run time 1.5 sec, on a pentium 750 mhz.

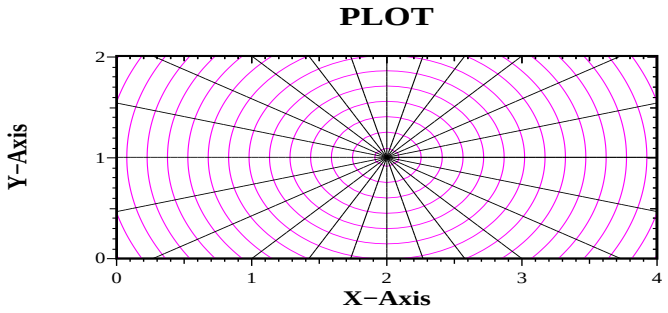


Figure 1:

Figure 1 shows on the same plot the projection of the "space-time rays" on the plane and the zero level curves of $t - \phi(t, x)$ of the Eulerian phase. As expected (the Lagrangian problem can be solved analytically, the phase is single-valued and the Eulerian solution follows automatically) the rays are straight lines and the level curves behave as regular expanding circular wavefronts. Here we have used plotmtv as a graphical software (see [2] for more on graphic facilities in GO++).

10 A second example : a waveguide problem

We review the different steps needed to solve a similar problem set with a different and more general Hamiltonian function and a different numerical method.

10.1 Creating a new Hamiltonian class

Our new Hamiltonian function is the classical geometric optic Hamiltonian with variable index of refraction n :

$$H(s, y, p) = \frac{1}{2}(|p|^2 - n^2(y)) \tag{7}$$

We take

$$n(x) = \begin{cases} 1 & |x| > 0.5, \\ 1 + cos^2(\pi\,|x|) & |x| \leq 0.5. \end{cases}$$

a function of configuration space which can be implemented in a child class of the $XFCT$ class called $XFCT_INDEX_WG$:

```
void XFCT_INDX_WG::F(ISH* FX) const
{
  //*  Cylindrical waveguide
  //*  X.y X.p phase-space coordinates

  R ray,s;

  for (int i=FX[0].discr.ord;i<FX[0].discr.nx+FX[0].discr.ord;i++)
    {
      for (int j=FX[0].discr.ord;j<FX[0].discr.ny+FX[0].discr.ord;j++)
        {
          ray = sqrt((FX[0].y('.',i,j),FX[0].y('.',i,j)));

          if (ray > .5)
            {       //* constant index out of  the ball B(0,0.5)
              FX[0].phi(i,j)=1.;
              FX[0].p('.',i,j)=0.;
            }
          else
            {       //* cosine of the radius inside  the ball B(0,0.5)
              s=cos(pi*ray);
              FX[0].phi(i,j)=1.+s*s;
              FX[0].p('.',i,j)=FX[0].y('.',i,j);
              FX[0].p('.',i,j)*=-2.*s*pi*sin(pi*ray)/(ray+1e-9);
            }
        }
    }
}
```

Then we create a child class of $HAMI_CONT$ which includes the specification of an index of refraction in the form of a $XFCT$ class. This new Hamiltonian class is called $HAMI_CONT_XFCT$. We specialize further the Hamiltonian class to take into account the particular geometrical optics form (7) in the class $HAMI_CONT_XFCT_GO$.

```
class HAMI_CONT_XFCT: public HAMI_CONT
{
 public:
  //*  Specify a function XFF
  HAMI_CONT_XFCT(const XFCT& XFF ) : HAMI_CONT(XFF.nfdis), XF(XFF) {}
  virtual ~HAMI_CONT_XFCT(){}

  //* The XFCT class (analytic or discrete)
  const XFCT &XF;

  //* The xFHami function is then F member
  //* data function of an instantiation of a XFCT
  inline void xFHami(ISH* FX) {XF.F(FX);}

};
```

and the code for the child class $HAMI_CONT_XFCT_GO$

```
/**
  H  = Y.phi = (|p|^2-n^2(y))/2
  Hp = Y.p   =   p
```

```
  Hy = Y.y   = -n Grad n
*/
void HAMI_CONT_XFCT_GO::Hxph(const R& t, const ISH* X,ISH& Y) const
{
 //* ci dessous dH/dp = p (idem)
 Y.p=X->p;

 //* ci-dessous dH/dy (initialized to Grad n)
 Y.y=fX()[0].p;

 for (int i=X->discr.ord;i<X->discr.nx+X->discr.ord;i++)
   {
    for (int j=X->discr.ord;j<X->discr.ny+X->discr.ord;j++)
      {
      // H=(|p|^2 - n(y)^2)/2
      Y.phi(i,j)=(X->p('.',i,j),X->p('.',i,j))-(fX()[0].phi(i,j)*fX()[0].phi(i,j));
      Y.phi(i,j)/=2.;

      // dH/dy=-n * Grad n
      Y.y('.',i,j)*=-fX()[0].phi(i,j);
      }
   }

}
```

The last step is to modify the declaration lines for the Hamiltonian to make these changes operational, (i.e. modify the declaration in the model section of the main program)

```
\* define the Hamiltonian function
HAMI_CONT_ANAL_P2 ham;
```

by

```
\* define the index of refraction
XFCT_INDX_WG indrefct;
\* define the GO Hamiltonian function , specify an index
HAMI_CONT_XFCT_GO ham(indrefct);
```

10.2 Implementing a new numerical Hamiltonian

The same mechanism is used to modify the numerical Hamiltonian. A Godunov method specialized for convex Hamiltonian depending on p^2 like for $HAMI_CONT_XFCT_GO$ can simply be written as:

$$H^{num}_{GOD}(s,x,a,b,c,d) := H(s,x,max(max(a,0),-min(b,0)),max(max(c,0),-min(d,0)))$$

where (a,b,c,d) are defined in section 7.1. We then create a child class $HAMI_NUM_GOD$ (source below) and modify accordingly the class declaration:

```
GRAP_UPW1 gradphi(discr)
\* numerical hamiltonian (space)
HAM_NUM_GOD hamnum(ham,gradphi)
```

Note that *ham* describes our new Hamiltonian function and *gradphi* the upwind formula, the C++ code in this case is

```
/*
  Numerical Hamiltonian for Square Hamiltonian
*/
void  HAMI_NUM_GOD::Hxph(const R& t, const ISH* X, ISH& Y) const
{
  //* The modified solution Xmod is set to the solution X
  //* Below, we will change his gradient (Xmod->p) ...
  *Xmod=*X;

  //* approximate upwind gradient
  ApxGrad.GradPM(X->phi,*Grphim,*Grphip);

  /**
    when H=H(x,p^2,q^2)=G(x,p,q),
    Godunov gradient is grad phi = max(max(Grphi-,0),-min(Grphi+,0))
  */

  /**
    Godunov gradient for the solution
  */
  for (int i=X->discr.ord;i<X->discr.ord+X->discr.nx;i++)
    {
      for (int j=X->discr.ord;j<X->discr.ord+X->discr.ny;j++)
        {
          for(int d=0;d<2;d++)
            {
              Xmod->p(d,i,j)=max(max((*Grphim)(d,i,j),0.),-min((*Grphip)(d,i,j),0.));
            }
        }
    }

  //* Evaluate the continuous hamiltonian with this gradient
  Hamc.Hxph(t,Xmod,Y);
}
```

10.3 Changing the initialization

In section 4.4, we already created initialization functions (for a source point simulation). One can modify the initialization function by creating his own $XFCT$ child class. For the wave guide simulation the default initialization $\phi^0 = 0$ (straight rays) is used.

```
//******** define X and initialize *****
// Eulerian
ISH* X_eul= new ISH(discrE,0.);

// Lagrangian
ISH* X_lag= new ISH(discrL,0.);
```

10.4 Changing the boundary conditions

As we use a Godunov solver for our numerical Hamiltonian we must modify the boundary conditions accordingly.

```
// Conditions Limites cas Eulerien (Godunov solver)
BCON_EUL_OUTG_GODUNOV bcond;
```

10.5 Results

The index of refraction has a maximum at $x = 0$ and bends the rays towards the $x = 0$ line. Shooting straight rays ($\phi^0 = 0$) in the interior region $|x| < 0.5$ will produce a wave guide effect. In this simulation, $x_{1,min} = -1, x_{1,max} = 1, x_{2,min} = -1, x_{2,max} = 1$. In the Eulerian case, we took $nx_1^e = nx_2^e = 101$, $ord = 1$, and for the Lagrangian case 11×11 rays. We evolve the two problems simultaneously until the time $tmax = 4.5$ with a constant time step 0.0075.

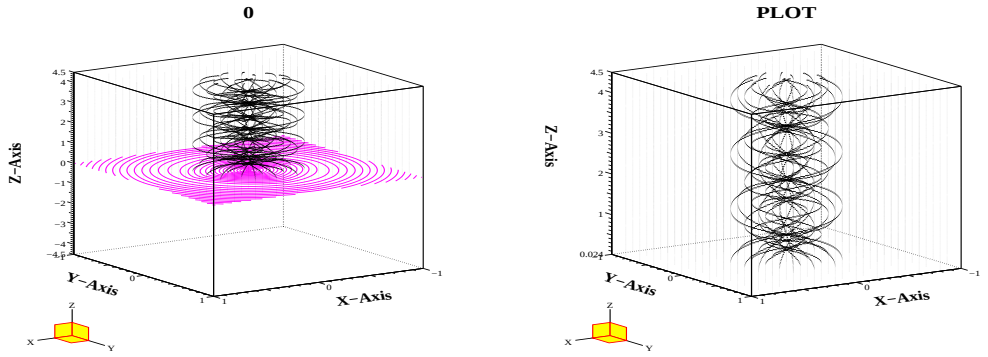


Figure 2: Space time perspective of the rays

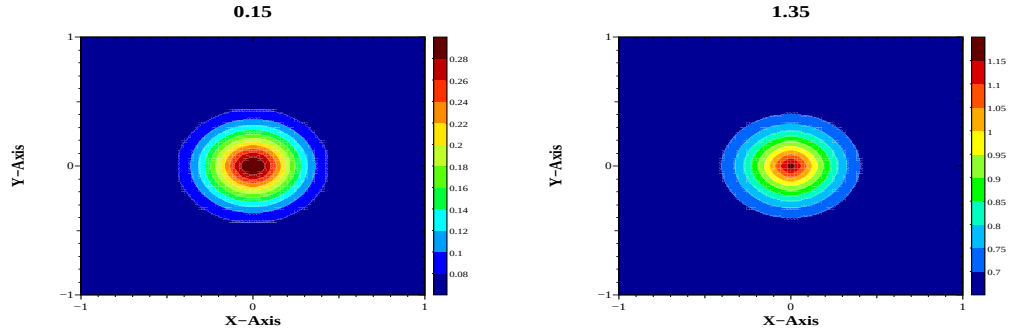


Figure 3: Greyscale map of times slices of the phase

In figure 2 we see the waveguide effect on the rays while outside the zone of inhomogeneity of the index rays propagate in straight lines.

Figure 3 shows two time slices of the phase computed with the Hamilton Jacobi equation. On can observe the radial spatial gradient of the phase corresponding to the index of refraction. The Eulerian solution picks up the phase of only the fastest rays according to formula (4).

11 A generalization to semi-analytically defined Hamiltonians

In many applications the Hamiltonian function reflects the properties of a medium which are not necessarily known analytically. For example the sound speed c in the ground may be specified on a grid like in the Marmousi test case
(see `http://www-rocq.inria.fr/ benamou/traveltimes.html`). In this case, the geometric optics Hamiltonian is $H(s,y,p) = c(y)|p|$ with the sound speed c given as collection of samples at grid points $(x^{int}_{1,i}, x^{int}_{2,j}), i = 1,..,n^{int}_{x1}, j = 1,..,n^{int}_{x2}$. An interpolation method is therefore needed at least for the Lagrangian resolution and also in the Eulerian case when the Eulerian grid does not match with the location of the samples. This functionality can be integrated into GO++ by the addition of a family of a new class.

11.1 Interpolation methods : the $XFCT_INTR$ and the $INTR$ classes

We recall that the $XFCT$ has a child $XFCT_INTR$ which stands for functions specified only at a discrete set of points and needs and interpolation method. The F member function of $XFCT$ is then replaced by a member function of $INTR$ called $Interp$ which does the interpolation job. The $INTR$ virtual class has therefore a virtual member function $Interp$ which takes an ISH variable say IX that contains $IX.y = X$ the set of points, $IX.p = \nabla F(X)$ the gradient of the function at these points and $FX.phi = F(X)$ the function itself.

To build such an object, we must read sample data. We treated the case of data specified on a Cartesian discretization of a square domain in a specialized class called $INTR_CART$. A text data file (generically called $Intrfile.dat$) containing ($x^{int}_{1,min}$, $x^{int}_{1,max}$, $x^{int}_{2,min}$, $x^{int}_{2,max}$), n^{int}_{x1} n^{int}_{x2}) the coordinates and discretization of the sample points followed by the value of the function at these points must be specified. The several methods can be used such as B-splines ($INTR_CART_BSPLINE$) or Catmull-Romm splines ($INTR_CART_CRSPLINE$).
In the Marmousi test case the top lines of the $Intrfile.dat$ (called "marmousi.dat") file looks like

```
0. 9192.     //  x1intmin x1intmax
2904. 0.     //  x2intmin x2intmax
384 122      //   nx1 nx2
1            // only 1 field to interpolate
  1500.      // index in m/s (column wise.....)
  1549.
  1598.
  1637.
  1675.
   :
   :
```

where the data are respectively : $x^{int}_{1,min}$, $x^{int}_{1,max}$, and $x^{int}_{2,min}$, $x^{int}_{2,max}$, and n^{int}_{x1} n^{int}_{x2}, and the number of data fields, and finally the values (regularly spaced) of the sound speed.
Then if we want to use B-spline, the syntax are defined by the following specifications

```
\* define the interpolation method and data file
INTR_CART_BSPLINE interp("marmsmooth.dat");

\* Define the index
XFCT_INTR marmind(interp);

\*define the hamiltonian
HAMI_XFCT_CONT_MI ham(marmind) ;
```

11.2 Result

We solve the problem on the domain defined by $x1_{min} = 0, x1_{max} = 9192, x2_{min} = 0, x2_{max} = 2904$ with a source point S located at $(6000, 104)$. In the Eulerian case, we use the same initial condition as in section 9 and apply outgoing boundary conditions. The discretization uses $nx_1^e = 384$, $nx_2^e = 122$, and a 2nd order TVD ([17]) ($ord = 2$) method for the approximation of the gradient.
In the Lagrangian case, we shoot 7×7 rays, the initial condition is the same as in the isotropic case except that $X.p$ is divided by the sound speed at the source point S. We evolve the two problems simultaneously until the time $tmax = 2.6$ with a constant time step 0.0043.

Figure 4 shows the heterogeneous speed function characterizing the underground, the location of the source point and the associated rays which are bent according to the variation of the index $\frac{1}{c}$. A grayscale plot of two time slices of the Eulerian phase are given in figure 5. It provides in particular the phase function in shadow regions.

12 A new application : the Stationary problem

We propose here a new application for the resolution of the Eulerian stationary problem

$$H(x, \nabla\Phi) = 0, \tag{8}$$
$$\Phi(x) = g(x). \tag{9}$$

Hypothesis that yield well-posededness of such an application can be found in [4] [8].

12.1 The Algorithm

The approximation is based (see [12]) on a first order (ord=1 ...) algorithm using an hyperbolic approach and an iterative process introduced by Rouy-Tourin [15] :

(1) Initially: n=0; we choose $U^0 = (U^0_{ij})$ such that $U^0_{ij} = g_{ij}$ for (i,j) on ∂Q, and $H(M_{ij}, \nabla U^0) \leq 0$ for $(i,j) \in Q$.

(2) At time step $n+1$, for each $(i,j) in Q$, with fixed values U_{kl} $(k,l) \neq (i,j)$, solve $L(U)_{ij} = 0$ with

$$L(U)_{ij} = \left(\frac{\alpha(M_{ij})}{\Delta x} + \frac{\beta(M_{ij})}{\Delta y}\right) U^{n+1}_{ij} + K(M_{ij}, U^n_{i-1j}, U^n_{i+1j}, U^n_{ij-1}, U^n_{ij+1}), \tag{10}$$

where

$$K(M,a,b,c,d) = H(M, \tfrac{b-a}{2\Delta x}, \tfrac{d-c}{2\Delta y}) - \tfrac{1}{2}\left(\tfrac{\alpha(M)}{\Delta x}(a+b) + \tfrac{\beta(M)}{\Delta y}(c+d)\right). \tag{11}$$

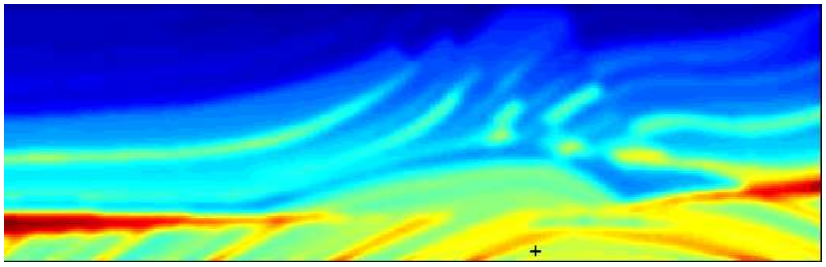
(1) means that we start with a sub-solution and (2) that we solve at each step the equation $L(V)_{ij} = 0$ with the unknown $V_{ij} = U^{n+1}_{ij}$ with the "frozen" neighboring values U^n_{i-1j}, U^n_{i+1j}, U^n_{ij-1}, U^n_{ij+1}, and we have $\alpha(x) := \sup_{(u,v)} |\frac{\partial H}{\partial u}|$, $\beta(x) := \sup_{(u,v)} |\frac{\partial H}{\partial v}|$.

It is important to notice that $L(U)_{ij} = 0$ can be reset in the familiar form

$$U^{n+1}_{ij} = RHS(M_{ij}, U^n_{i-1j}, U^n_{i+1j}, U^n_{ij-1}, U^n_{ij+1}) \tag{12}$$

12.2 The GO++ implementation

Even though the problem is Eulerian, we will only need a continuous Hamiltonian and no numerical Hamiltonian as we chose to directly incorporated the discretization method (10-11) in the RHS function of the application. The usual way to specify the discretization (section 4.2) to the application is through the numerical hamiltonian. Therefore in this case the discretization has



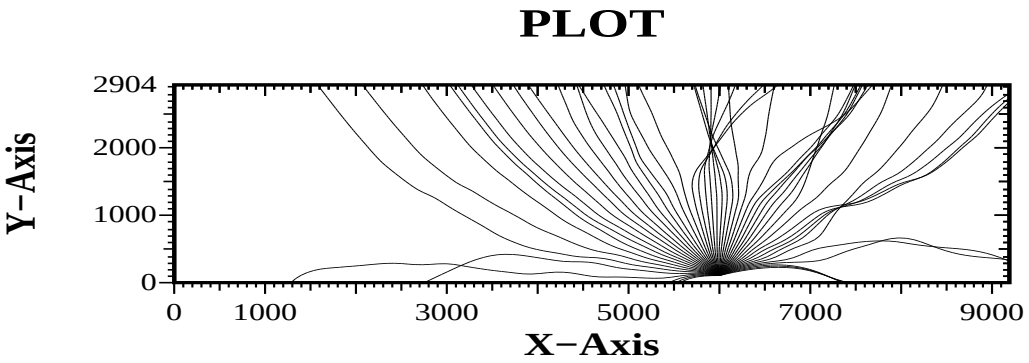


Figure 4: The speed c, the point source and the rays

to be explicitly given in this initialization list. It is not necessary to call any timestep routines because (12) is a simple iterative process that can be implemented in the main source.

The new application is added in the the application hpp and cpp file under the $APLI_EUL_ST$ name. the heading of the associated RHS function is

```
/**
 RHS  Eulerian Stationnary problems   H(x,grad(u)) = 0  + Boundary Conditions
```

```
 Here, we use an iterative scheme comming from Rouy-Tourin
 coupled with a numerical hamiltonian of Lax-Friedrichs type

 This scheme can be used for ANY Hamiltonian H(x,grad(u))
 (beause of the regular cartesian mesh)
*/

void APLI_EUL_ST::Rhs(const R& t,ISH* X, ISH& Y) //const
{
  //* Set Boundary conditions
  CL.Sbco(t,X,Y);

  /**
    Compute center derivatives

etc ........
```

As before only the class declaration in the main program have to be modified. Replace

```
APLI_EUL_ eul(hamnum,bceul);
```

by the lines:

```
//* specify continuous Ham. hamS, Bound. Condition bcsta,
//* Discretization discrS
APLI_EUL_ST statio(hamS,bcstat,discrS);
```

12.3 results

We recall that the zero level curves for the solution of the Eulerian evolution problem with H, are intimately connected to the t level curves of the Eulerian stationary problem with $\tilde{H} = H - 1$ see e.g. [4] and [12] for numerical examples.
Note that even if we solve $\tilde{H} = 0$, we take the same Hamiltonian like the evolution problem, instead in the RHS of stationary problem we do that:

```
Ham.Hxph(t,X,Y);
Y->phi-=1.;
```

bcstat is a boundary condition, and discrS is a discretization (with ord=1).
Here we show an example on the marmousi model:

Eulerian Evol (18 zero level curves) Eulerian Stat (15 level curves)

13 Outputs

13.1 Basic « overloading

The ISH classes propose a basic overloading of the Rnmk array classes « functionality. The y,p and phi variable either in the Lagrangian or the Eulerian case are always declared as such arrays and represent the solutions at a fixed time step (see section 4.3 for a precise description). An example follows that writes on one output file the arrays of positions of the rays at all timesteps and the arrays of phases on the Eulerian grid again at all timesteps.

```
//*  declare output files

   ofstream solLag("SOL_lag.dat");
   ofstream solEul("SOL_eul.dat");

//* iterate
  while (t<t1)
    {
      cout << " t=" << t << "\n";
      cp++;

      // Evolution of Lagrangian Sol over one time step
      tslag.Tstep(t,dt,X_lag);

      // Evolution of Eulerian Sol over one time step
      tseul.Tstep(t,dt,X_eul);

      t+=dt;

      //* write the array of ray positions at time t
      solLag << X_lag->y << endl;
      //* write the array of phases on the Eulerian grid at time t
      solEul << X_eul->phi << endl;
    }
```

The output files are text files where each line correspond to a line of the arrays and the arrays are stacked one above the other. When the Rnmk array is 3-dimensional, i.e two space dimension $nx \times ny$ and another dimension for vector valued unknown (here only 2-D vectors y and p) the vector components arrays are written one after the other.

13.2 Plotmtv files

We also provide basic c++ functions to produce output files which can be used with the "plotmtv" graphics freeware. (`ftp://keel.mit.edu/pub/Plotmtv/`. All these functions are in the *sortie_resMTV* cpp and hpp file. The available output functions are

- $sortieMTV_cbniv0(ostream\& \, os, const \, ISH * \, X, const \, R\& \, t, const \, R \, val = 0)$: saves at fixed time t the val level curves of $X.y$ in file os. If val is ommited it defaults to 0.
- $sortieMTV_cbnivx(ostream\& \, os, const \, ISH * X, const R\& \, t)$: saves at fixed time t a default 15 level curves of $X.y$ in file os.
- $sortieMTV_lagra(ostream\& \, os, const \, ISH * \, Xap, const \, R\& \, t)$

All these functions must be called after an output file declaration and inside the time loop. As an example the

```
solEul << t << endl << Xeul->phi << endl;
```

by
Example : level curve $\psi(t,.) = t$

```
sortieMTV_cbniv0(solEul,Xeul,t,t);
```

or every level curves:

```
sortieMTV_cbnivx(solEul,Xeul,t);
```

At the end of the computation, you have to execute on the command line

```
plotmtv -plotall SOL_eul.mtv
```

to visualize the level curves. and

```
plotmtv -plotall SOL_lag.mtv
```

to visualize the rays.

Acknowledgments

Many thanks to S. Delpino and W.W. Symes for their advice on the C++ language.

References

[1] R. Abgrall and J.-D. Benamou. Big ray tracing and eikonal solver on unstructured grids : Application to the computation of a multi-valued travel-time field in the marmousi model. *Geophysics*, 64:230–239.

[2] J.-D. Benamou and P. Hoch. GO++ : A modular Lagrangian/Eulerian software for Hamilton Jacobi equations of Geometric Optics type *INRIA Tech. Report RR*.

[3] V. I. Arnol'd. *Mathematical methods of Classical Mechanics*. Springer-Verlag, 1978.

[4] G. Barles. *Solutions de viscosité des équations de Hamilton-Jacobi*. Springer-Verlag, 1994.

[5] J.-D. Benamou. Big ray tracing : Multi-valued travel time field computation using viscosity solutions of the eikonal equation. *J. Comp. Physics*, 128:463–474, 1996.

[6] J.-D. Benamou. Direct solution of multi-valued phase-space solutions for hamilton-jacobi equations. *Comm. Pure Appl. Math.*, 52, 1999.

[7] Y. Brenier and L.Corrias. A kinetic formulation for multi-branch entropy solutions of scalar conservation laws. *Ann. IHP Analyse non-linéaire*, 1996.

[8] M.G. Crandall and P.L. Lions. Viscosity solutions of hamilton-jacobi equations. *Trans. Amer. Math. Soc.*, 277:1–42, 1983.

[9] J.J. Duistermaat. Oscillatory integrals, lagrange immersions and unfolding of singularities. *Comm. Pure Appl. Math.*, 27:207–281, 1974.

[10] B. Engquist, E. Fatemi, and S. Osher. Numerical resolution of the high frequency asymptotic expansion of the scalar wave equation. *J. Comp. Physics*, 120:145–155, 1995.

[11] B. Engquist and O. Runborg. Multi-phase computation in geometrical optics. *Tech report, Nada KTH*, 1995.

[12] Ph. Hoch. PhD Thesis, May 2000.

[13] S. Izumiya. The theory of legendrian unfoldings and first order differential equations. *Proc. Royal Soc. Edinburgh*, 123A:517–532, 1993.

[14] M. Fang J. Steinhoff and L. Wang. A new eulerian method for the computation of propagating short acoustic and electromagnetic pulses. *J. Comp. Physics, to appear.*

[15] Rouy E., Tourin A., A viscosity solutions approach to shape-from-shading, *SIAM J. Num. Anal.*, vol 29, No3, p 867-884, 1992.

[16] G. Lambare, P. Lucio, and A. Hanyga. Two dimensional multi-valued traveltime and amplitude maps by uniform sampling of a ray field. *Geophys. J. Int*, 125:584–598, 1996.

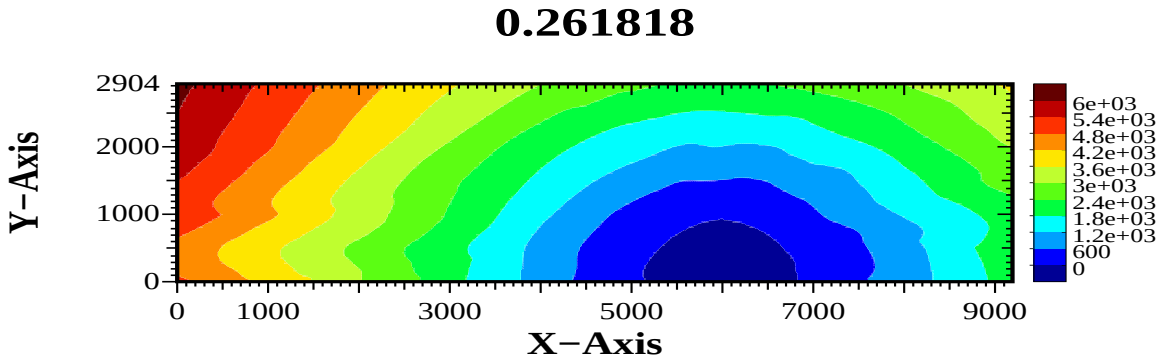
[17] S. J. Osher and C.W. Shu. High-order essentially nonoscillatory schemes for hamilton-jacobi equations. *SIAM J. Numer. Anal.*, 83:32–78, 1989.

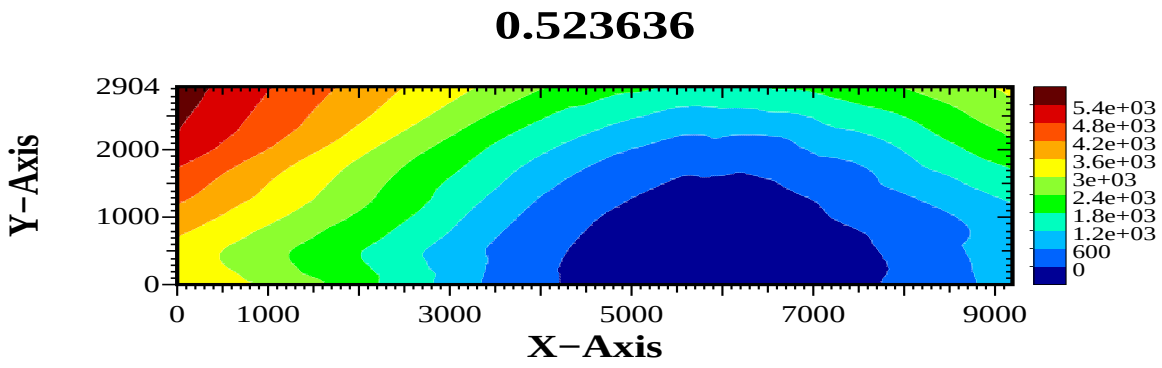
[18] B. Merryman S. Ruuth and S.J. Osher. A fixed grid method for capturing the motion of self-intersecting interfaces and related pdes. *preprint*, 1999.

[19] W. Symes. A slowness matching algorithm for multiple traveltimes. *TRIP report*, 1996.

[20] V. Vinje, E. Iversen, and H. Gjoystdal. Traveltime and amplitude estimation using wavefront construction. *Geophysics*, 58:1157–1166, 1993.

[21] L.C. Young. *Lecture on the Calculus of Variation and Optimal Control Theory.* 1969.


0.261818
2904
2000
1000
0
Y−Axis
0
1000
3000
5000
7000
9000
X−Axis
6e+03
5.4e+03
4.8e+03
4.2e+03
3.6e+03
3e+03
2.4e+03
1.8e+03
1.2e+03
600
0


0.523636
2904
2000
1000
0
Y−Axis
0
1000
3000
5000
7000
9000
X−Axis
5.4e+03
4.8e+03
4.2e+03
3.6e+03
3e+03
2.4e+03
1.8e+03
1.2e+03
600
0



Unité de recherche INRIA Rocquencourt
Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay 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 - 38330 Montbonnot-St-Martin (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 0249-6399