Guidebook for the finite volume program

1 Introduction

This document is the guidebook refering to the finite volume code to solve the Saint-Venant system in an open channel. It is dedicated to people who want to use the code in order to obatin numerical solutions of the Saint-Venant system or maintain and change the program. In this paper, we are going to describe each file of the program with the procedures and functions. When it will be necessary, we will detail some choice of implementation, espacially for the implementation of the kinetic solver which need to be clarify. Then, we will give an example of using in order to obtain numerical results using screenshots of the code. We point out that, in some screenshots, the code is uncomment to be more clear and to save space. But in the files, all the code is commented. The programm is under a GPL licence and each file contains the following lines :

```
/* ==========================================================

  A finite volume code to solve the Saint−Venant system in an
       open channel.

   ==========================================================

  Copyright (C) 2013 Mathieu Besson

  This program is free software: you can redistribute
  it and/or modify it under the terms of the
  GNU General Public License as published by
  the Free Software Foundation, either
  version 3 of the License, or (at your option) any later
      version.

  This program is distributed in the hope that it will be
      useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty
  of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
  See the GNU General Public License for more details.

  You should have received a copy of the GNU General Public
      License
  along with this program. If not, see <http://www.gnu.org/
      licenses/>.*/
```

Listing 1: GPL licence

2 Listing of files and comments

- **constant.h**

```
#ifndef constant_h
  #define constant_h

  #define PI 3.1415
  #define g 9.81 /* gravitational constant */
  #define bInf -10.0 /* the lower bound of the domain */
  #define bSup 10.0 /* the upper bound of the domain */
  #define nbCell 1000 /* Number of cell */
  #define freq 1 /* The frequence where you want to save
      data */
  #define T 1.5 /* time */
  #define epsilon 0.00001 /* the precision we want */
  #define N 1000 /* N is the number of iteration you want
       to do to approximate the integrals for each flux
      */
#endif
```

Listing 2: Definition of constants for the program in the header ”constant.h”

As we see, nine constants are defined.

The first one is the famous constant π usfull for the kinetic scheme. Then we also define the gravitational constant g.

The constants **bInf** and **bSup** are dedicated to the domain where we want to solve the problem. The lower bound of the domain is defined by **bInf** and the upper born by **bSup**.

The framework of the finite volume method requires to define the number of cell we want for the discretization of the domain. This number is define by the constant **nbCell**.

The constant **freq** refers to the recording of data. Indeed, if there are many time iterations, it is not wise to save all the data at each iteration. The constant **freq** allows us to deal with this fact. For example, if $freq = 3$, we are going to save data each 3 time iterations. The programm saves automatically the initial data and data at last time iteration.

The constant **T** represents the duration of the experiment in seconds.

Epsilon is a criterion to test if a quantity is equal to zero or not as, in computing, we don't test the exact equality with zero.

The last constant **N** is the number of iterations we want to approximate integrals with the quadrature for the kinetic scheme.

- **initialConditions.h & initialConditions.c**
  The files **initialConditions.h** and **initialConditions.c** are dedicated to initial conditions. The prototypes of the various functions and procedures are in the file **initialConditions.h**. The implementation of the functions and procedures are located in the file **initialConditions.c**.

  For the moment, only three functions are available.

  - **The procedure damBreakWetBed**
    This procedure allows to start with the initial condition of a dam break on wet bed. It takes for input arguments three arrows of size nbCell + 2 (nbCell for the within domain and 2 for the boundary conditions) h for the height of water, hu for the discharge and Z for the topography and a float ”dx” which represents the space step. The arrows h and hu are modified by the procedure with the definition of the initial condition and are also output variables.

  - **The procedure damBreakDryBed**
    This procedure allows to start with the initial condition of a dam break on dry bed. It takes for input arguments three arrows of size nbCell + 2 (nbCell for the within domain and 2 for the boundary conditions) h for the height of water, hu for the discharge and Z for the topography and a float ”dx” which represents the space step. The arrows h and hu are modified by the procedure with the definition of the initial condition and are also output variables.

  - **The procedure lakeAtRest**
    This procedure allows to start with the initial condition of a dam break on wet bed. It takes for input arguments three arrows of size nbCell + 2 (nbCell for the within domain and 2 for the boundary conditions) h for the height of water, hu for the discharge and Z for the topography. The arrows h and hu are modified by the procedure with the definition of the initial condition and are also output variables.

  We point out that we have used float numbers for this functions and procedures and in what follow. Of course the programmer can change it if he wants to work with double numbers.

```c
#ifndef initialConditions_h
  #define initialConditions_h

  #include <stdio.h>
  #include "constant.h"

  void damBreakWetBed(float h[nbCell+2],float hu[nbCell
      +2],float Z[nbCell+2],float dx);
  void damBreakDryBed(float h[nbCell+2],float hu[nbCell
      +2],float Z[nbCell+2],float dx);
  void lakeAtRest(float h[nbCell+2],float hu[nbCell+2],
      float Z[nbCell+2]);
#endif
```

Listing 3: The header ”initialConditions.h”

- **boundaryCondition.h & boundaryCondition.c**
  The files **boundaryCondition.h** and **boundaryCondition.c** deal with boundary conditions. In the file **boundaryCondition.h**, you can find prototypes of functions and procedures. The impelmentation of the previous function and procedure is on the file **boundaryCondition.c**.

  Three procedures are available now for three different boundary conditions.

  - **The procedure updateBC**
    With this procedure, we model free boundary conditions. To do this, we need two arrows in input of size nbCell + 2. The arrows hNew and huNew represent respectively the height of water and the discharge at the current time iteration. The procedure modify the arows hNew and huNew thanks to the definition of free boundary conditions that is why this two variables are also output variables.

  - **The procedure updateBCPeriodic**
    The procedure allows us to model periodic boundary conditions. To do this, we need four arrows in input of size nbCell + 2. The arrows hNew and huNew represent respectively the height of water and the discharge at the current time iteration. The last two arrows hOld and huOld refer respectively the height of water and the discharge at the previous time iteration. The procedure modify the arows hNew and huNew thanks to the definition of periodic boundary conditions that is why this two variables are also output variables.

  - **The procedure updateBCLakeAtRest**
    With the procedure updateBCLakeAtRest, we model boundary conditions for a lake at rest, it means that the discharge is equal to zero and the height plus the topography is equal to a constant. To do this, we only need two arrows in input of size nbCell + 2. The arrows hNew and huNew represent respectively the height of water and the discharge at the current time iteration. The procedure modify the arows hNew and huNew thanks to the definition of free boundary conditions that is why this two variables are also output variables.

```
#ifndef boundaryCondition_h
  #define boundaryCondition_h
  #include <stdio.h>
  #include <stdlib.h>
  #include <string.h>
  #include "constant.h"
  #include <math.h>

  void updateBC(float hNew[nbCell+2],float huNew[nbCell
      +2]);
  void updateBCPeriodic(float hNew[nbCell+2],float huNew[
      nbCell+2],float hOld[nbCell+2],float huOld[nbCell
      +2]);
  void updateBCLakeAtRest(float hNew[nbCell+2],float
      huNew[nbCell+2]);

#endif
```

Listing 4: The header ”boundaryCondition.h”

- **kineticSolver.h & kineticSolver.c.**
  With the two files **kineticSolver.h** and **kineticSolver.c**, we can implement the kinetic scheme of B. Perthame and C. Simeoni. As usual, all the prototypes of functions and procedures are in the file **kineticSolver.h** and they are implemented in the file **kineticSolver.c**. To implement the kinetic scheme, we have separate steps with into ten procedures and two functions in order make the code more understandable.

  - **The procedure updateKinetic**
    This is the central procedure to update the height of water an the discharge at the current iteration, using the height of water and the discharge at the previous iteration and the variation of fluxes. This procedures takes in input/output variables two vectors, hNew for the height of water and huNew for the discharges. This variables are modified thanks to the height of water hOld and the discharge huOld at the previous iteration, the topography Z and a float number, rat which is defined in the file main.c as the ratio between de time step and the space step. This procedure calculates the fluxes for the mass equation and for the momentum equation using repectively the procedures **The procedure flux_hKinetic** and **The procedure flux_huKinetic**.

  - **The procedure flux_hKinetic**
    This procedure allows us to compute the flux for the mass equation of the Saint-Venant system. The value of the flux is computed using the input variables h for the height of water at the previous iteration and hu for the discharge at the previous iteration and the topography Z. The result is saved in the input/output variable fh.

The first step in the procedure flux_hKinetic is to initialize variables we need. Here, we need seven arrows of size nbCell + 2. The arrow FPlus will contain the value of the flux for the mass equation at the interface $i+\frac{1}{2}$ whereas FMinus will contain the flux at interface $i-\frac{1}{2}$. The variables KPlus and KMinus will contain the coefficients K^{+} and K^{-} defined in chapter **??** and in the article of B. Perthame and C. Simeoni [**?**]. We will specify when each coefficient is used later. Then, for each cell, we calculate the Froude number defined as :

$$Fr_i = \frac{u_i^n}{\sqrt{2gh_i^n}}$$

and save it in the variable FrNumber. Then, we compute the value α define as :

$$\alpha = min(1; max(-1; -Fr_i))$$

Be carefull, this value is the value to compute the flux at interface $i+\frac{1}{2}$ and we will need to calculate it again for the flux at interface $i-\frac{1}{2}$. Finally, we calculate $h^{\frac{3}{2}}$ which will be usefull in what follow.

```
for(i=0;i<=nbCell+1;i++)
{
  FPlus[i] = 0.0;
  FMinus[i] = 0.0;
  KPlus[i] = 0.0;
  KMinus[i] = 0.0;
  FrNumber[i] = getFroudeNumber(hu[i],h[i]);
  alpha[i] = min(1.0,max(-1.0,-FrNumber[i]));
  hPower[i] = power3Over2(h[i]);
}
```

Listing 5: Step 1

Then, we initialize the variable IPlus and IMinus. As we said in the chapter **??**, the flux can be separate into three integrals. That is why the size of this variables is 3*(nbCell+2). The variable IPlus will contain the three integrals for the flux at interface $i+\frac{1}{2}$ and IMinus will contain the value of the flux at interface $i-\frac{1}{2}$ for each cell.

```
for(i=0;i<=2;i++)
{
  for(j=0;j<=nbCell+1;j++)
  {
    IPlus[i][j] = 0.0;
    IMinus[i][j] = 0.0;
  }
}
```

Listing 6: Step 2

The next step is to calculate the flux FPlus. Before doing this, we compute $(K^+)_-$ by calling the procedure getKPlus and $(K^-)_+$ by calling the procedure getKMinus. In this case, $(K^+)_-$ and $(K^-)_+$ are defined as follow :

$$(K^-)_+ = \frac{\Delta(Z_{i+\frac{1}{2}})_+}{h_i^n}; \quad (K^+)_- = \frac{\Delta(Z_{i+\frac{1}{2}})_-}{h_{i+1}^n}$$

where

$$(\Delta Z_{i-\frac{1}{2}})_+ = max(0; \Delta Z_{i-\frac{1}{2}}) \quad (\Delta Z_{i-\frac{1}{2}})_- = max(-(\Delta Z_{i-\frac{1}{2}}); 0)$$

Finally, we call the procedure computeIntegralsFPlus_h to calculate the three integrals for the flux FPlus.

```
getKPlus(KPlus,h,Z,0,-1.0);
getKMinus(KMinus,h,Z,0,1.0);
computeIntegralsFPlus_h(alpha,FrNumber,IPlus,KPlus,
    KMinus);
```

Listing 7: Step 3

Then, we need to calculate the flux FMinus. Before doing this, we calculate again the value α defined this time as follow :

$$\alpha = max(-1; min(1; -Fr_i))$$

```
for(i=0;i<=nbCell+1;i++)
{
  alpha[i] = max(-1.0,min(1.0,-FrNumber[i]));
}
```

Listing 8: Step 4

The step number five is similar to the step number three. This time we calculate the integrals for the flux FMinus. Before this we compute the coefficients $(K^+)_+$ and $(K^-)_-$ are defined as follow :

$$(K^-)_- = \frac{\Delta(Z_{i+\frac{1}{2}})_-}{h_{i-1}^n}; \quad (K^+)_+ = \frac{\Delta(Z_{i+\frac{1}{2}})_+}{h_i^n}$$

And then, we calculate the three integrals for the flux FMinus by calling the procedure computeIntegralsFMinus_h.

```
getKPlus(KPlus,h,Z,1,1.0);
getKMinus(KMinus,h,Z,1,-1.0);
computeIntegralsFMinus_h(alpha,FrNumber,IMinus,KPlus,
    KMinus);
```

Listing 9: Step 5

Then, in the step number six, we calculate the value of FPlus and FMinus for each cell using the formula and the variables we calculate previously.

```
for(i=1;i<=nbCell;i++)
{
  FPlus[i] = hPower[i]*IPlus[0][i] + hPower[i]*IPlus
      [1][i] + hPower[i+1]*IPlus[2][i];
  FMinus[i] = hPower[i]*IMinus[0][i] + hPower[i]*
      IMinus[1][i] + hPower[i-1]*IMinus[2][i];
}
```

Listing 10: Step 6

Finally, we calculate the flux for the mass equation by doind the difference of fluxes at interface $i + \frac{1}{2}$ and at the interface $i - \frac{1}{2}$.
We point out that we do the same thing in the procedure **flux_huKinetic**. The only difference is that we compute the flux for the momentum equation of the Saint-Venant system.

```
for(i=1;i<=nbCell;i++)
{
  fh[i] = (2.0*sqrt(2.0*g)*(FPlus[i] - FMinus[i]))/PI
      ;
}
```

Listing 11: Step 7

– **The procedure getKPlus**
  The procedure getKplus allows us to calculate the values $(K^+)_-$ and $(K^+)_+$. The result is saved in the input/output vaiable KPlus. To compute this values, we need the height hOld and the topography Z. The integer face allows us to choose the right interface. If $face = 0$, we are at interface $i + \frac{1}{2}$ and if $face \neq 0$, we are at interface $i - \frac{1}{2}$. Then, if the float $coef = 1$, we get the coefficient $(K^+)_+$ and if $coef = -1$ we get the value of $(K^+)_-$.
  We point out that the procedure **getKMinus** works as the procedure getKPlus. The only difference is that we can calculate $(K^-)_-$ and $(K^-)_+$.

– **The procedure getK**
The procedure getK is similar to procedures getKPlus and getKMinus. The difference is that we calculate the values of K^{-} and K^{+} which are usfull for the quadratures for the computation of the flux for the momentum equation.

```
#ifndef kineticSolver_h
  #define kineticSolver_h

  #include <stdio.h>
  #include <stdlib.h>
  #include <string.h>
  #include "usefull.h"
  #include "constant.h"
  #include <math.h>

  void updateKinetic(float hNew[nbCell+2],float huNew[
      nbCell+2],float hOld[nbCell+2],float huOld[nbCell
      +2],float Z[nbCell+2],float rat);
  void flux_hKinetic(float h[nbCell+2],float hu[nbCell
      +2],float fh[nbCell+2],float Z[nbCell+2]);
  void flux_huKinetic(float h[nbCell+2],float hu[nbCell
      +2],float fhu[nbCell+2],float Z[nbCell+2]);
  void computeIntegralsFPlus_h(float a[nbCell+2],float Fr
      [nbCell+2], float I[3][nbCell+2],float KPlus[nbCell
      +2],float KMinus[nbCell+2]);
  void computeIntegralsFMinus_h(float a[nbCell+2],float
      Fr[nbCell+2], float I[3][nbCell+2],float KPlus[
      nbCell+2],float KMinus[nbCell+2]);
  void computeIntegralsFPlus_hu(float a[nbCell+2],float
      Fr[nbCell+2], float I[3][nbCell+2],float KPlus[
      nbCell+2],float KMinus[nbCell+2],float K[nbCell+2])
      ;
  void computeIntegralsFMinus_hu(float a[nbCell+2],float
      Fr[nbCell+2], float I[3][nbCell+2],float KPlus[
      nbCell+2],float KMinus[nbCell+2],float K[nbCell+2])
      ;
  float getFroudeNumber(float hu,float h);
  void getKPlus(float KPlus[nbCell+2],float h[nbCell+2],
      float Z[nbCell+2],int face,float coef);
  void getKMinus(float KMinus[nbCell+2],float h[nbCell
      +2],float Z[nbCell+2],int face,float coef);
  float f(float omega,float Fr, float K);
  void getK(float K[nbCell+2],float h[nbCell+2],float Z[
      nbCell+2],int face);
#endif
```

Listing 12: The header ”kineticSolver.h”

- **solverRusanov.h & solverRusanov.c.**
The files **solverRusanov.h** and **solverRusanov.c** allow us to implement the Rusanov scheme to solve the Saint-Venant system.

In the header **solverRusanov.h** we can find three prototypes of procedures we need to implement the Rusanov scheme.

- **The procedure updateRusanov.**
  The procedure updateRusanov has six variables in input and two in output. As usual, input/output variables are the height of water and the discharge at the current time iteration. Then, in input, we can also find the height of water and the discharge at the previous iteration, the topography Z and a float rat which represents the ratio between the time step and the space step.

- **The procedure flux_hRusanov**
  This procedure, like the procedure flux_huRusanov, has been created to simplify the understanding of the impelmentation of the Rusanov scheme. In this first procedure, we calculate the value of the flux for the mass conservation equation of the Saint-Venant system using the definition :

$$F(u_l, u_r) = \frac{f(u_l) + f(u_r)}{2} - c\frac{u_r - u_l}{2}$$

  We calculate the fluxes at the two interfaces and we make the difference to obtain the flux for each cell.
  This procedure takes in input variable the height of water and the discharge at the previous iteration, a vector fh which is also an output variable which contains the difference of fluxes at interface $i + \frac{1}{2}$ and $i - \frac{1}{2}$ and two other arrows, c1 and c2 which are calculated in the procedure updateRusanov and contain repectively the value of the first characteristic speed and the second characteristic speed difined section **??** in order to calculate :

$$c = max(|u_l| + \sqrt{gh_l}, |u_r| + \sqrt{gh_r})$$

- **The procedure flux_hRusanov**
  The procedure flux_huRusanov does exactly the same job as the procedure flux_hRusanov but this time for the computation of the flux for the momentum equation of the Saint-Venant system.

```
#ifndef solverRusanov_h
  #define solverRusanov_h
  #include <stdio.h>
  #include <stdlib.h>
  #include <string.h>
  #include "usefull.h"
  #include "constant.h"
  #include <math.h>

  void updateRusanov(float hNew[nbCell+2],float huNew[
      nbCell+2],float hOld[nbCell+2],float huOld[nbCell
      +2],float Z[nbCell+2],float rat);
  void flux_hRusanov(float h[nbCell+2],float hu[nbCell
      +2],float fh[nbCell+2],float c1[nbCell+2],float c2[
      nbCell+2]);
  void flux_huRusanov(float h[nbCell+2],float hu[nbCell
      +2],float fhu[nbCell+2],float c1[nbCell+2],float c2
      [nbCell+2]);

#endif
```

Listing 13: The header ”solverRusanov.h”

- **timeSetting.h & timeSetting.c**
  In this two files, we implement the prototypes and the bodies of functions we need to compute the time step at each time iteration. Of course, each scheme is different and requires his own CFL condition so his own function to calculate the time step.

  The procedure **timeStepRusanov** calulate the time step for the Rusanov solver using the CFL conditon

$$2\max_{i\in Z}(|\lambda_1(U_i^n)|, |\lambda_2(U_i^n)|)\Delta t \leq \Delta x$$

  Then, the procedure **timeStepKinetic** calculate the time step in order to respect the CFL condition for the kinetic scheme wich is the following one

$$\Delta t^n max\left(|u_i^n| + \sqrt{2gh_i^n}\right) \leq \Delta x$$

  For each procedure, we need as input variable de space step dx , the height of water and the discharge at the previous iteration. The output variable is the right time step dt.

```
#ifndef timeSetting_h
  #define timeSetting_h

  #include <stdio.h>
  #include "constant.h"
  #include "usefull.h"
  #include <math.h>

  float timeStepRusanov(float dx,float huOld[nbCell+2],
      float hOld[nbCell+2]);
  float timeStepKinetic(float dx,float huOld[nbCell+2],
      float hOld[nbCell+2]);
#endif
```

Listing 14: The header ”timeSetting.h”

- **topography.h & topography.c**
  The files **topography.h** and **topography.c** refer to the implementation of the topography. For the moment, we have been implemented four different kind of bed.

  - **The procedure noBed**
    The procedure **noBed** is really simple because it implements the null topography. The input/output variable is an arraw Z which contains the topography on each cell.

  - **The procedure bumpBed**
    This procedure implements a parabolic bump bed defined by the following formula :

    $$Z(x) = (0.2 - 0.05x^2)\mathbb{I}_{-2\leq x\leq 2}$$

    Like in the previous procedure, Z is an arrow which is an input/output variable and contains the topography. We also need in input the space step to calculate the topography on each cell.

  - **The procedure stepBed**
    The procedure **stepBed** implements a topography define as a step function. The input/output variable is an arraw Z which contains the topography on each cell.

  - **The procedure constantBed**
    This last procedure implements a constant topography defined by the user in the file **topography.c**. The input/output variable is an arraw Z which contains the topography on each cell.

```
#ifndef topography_h
  #define topography_h

  #include <stdio.h>
  #include <stdlib.h>
  #include <string.h>
  #include "constant.h"
  #include <math.h>

  void noBed(float Z[nbCell+2]);
  void bumpBed(float Z[nbCell+2],float dx);
  void stepBed(float Z[nbCell+2],float dx);
  void constantBed(float Z[nbCell+2]);

#endif
```

Listing 15: The header ”topography.h”

- **testCase.h & testCase.c**
  Each code needs to be validate. This is the role of the files **testCase.h** and **testCase.c**. For the moment two tests case have been implemented. First, the Ritter's solution for the test of a dam break on dry bed. This solution is implemented by the procedure **Ritter**. As usual, the input/output are the height of water h and the discharge hu. Then, to calculate the Ritter's solution we also need in input the space step dx, the time t and the value of the height of water hg for the initial dam break. Then, the Stoker's solution for a dam break on wet bed is implemented by the procedure **Stoker**. The variables used for this procedure are the same that in the procedure **Ritter**. The only difference is the float hd which represents the height of water at the right of the dam at the initial moment.

```
#ifndef testCase_h
  #define testCase_h

  #include <stdio.h>
  #include "constant.h"
  #include "usefull.h"
  #include <math.h>

  void Ritter(float h[nbCell+2],float hu[nbCell+2],float
      dx, float t,float hg);
  void Stoker(float h[nbCell+2],float hu[nbCell+2],float
      dx, float t,float hg,float hd);
#endif
```

Listing 16: The header ”testCase.h”

- **usefull.h & usefull.c**
  Sometimes in the program, we need to use functions or procedures which

don't have a direct link with the code. That's is why we put all this procedures and functions in the following files **usefull.h** and **usefull.c**.

- The function **maxVal** (resp. max, resp min) returns de maximum value of an arrow of size nbCell+2 (resp. the maximum value between two values in input, resp. the minimu value between two values in input).
- The function **maxVal** returns the absolute value of an input variable.
- The procedure **save** is used to save two arrows in input (typically the height of water and the discharge) at time t in input. The arrow tab1 is saved in the file *"t_height.txt"* and the arrow tab2 in saved in the file *"t_discharge.txt"*.
- The procedure **saveExact** has the same role as the previous procedure but is used to save exact data in files *"t_hExact.txt"* for the exact value of the height of water and *"t_huExact.txt"* for the exact value of the discharge.
- The procedure **getEigenValues** returns the eigenvalues

$$\lambda = u - \sqrt{gh} \quad and \quad \lambda = u + \sqrt{gh}$$

  respectively in the input/output variables EGMoins and EGPlus using the input arrows hu for the discharge and h for the height of water. The size of variables is nbCell + 2 since we calculate eigenvalues on each cell.
- The function **power3Over2** takes a float a in input and returns $a^{\frac{3}{2}}$.
- The last function **getNorme** takes two arrows in input and return the infinite norm of the difference of the arrows.

```c
#ifndef init_h
  #define init_h

  #include <stdio.h>
  #include <stdlib.h>
  #include <string.h>
  #include <math.h>
  #include "constant.h"

  float maxVal(float u[nbCell+2]);
  float max(float a,float b);
  float min(float a,float b);
  float vAbs(float x);
  void save(float tab1[nbCell+2],float tab2[nbCell
      +2],int t);
  void saveExact(float tab1[nbCell+2],float tab2[
      nbCell+2],int t);
  void getEigenValues(float hu[nbCell+2],float h[
      nbCell+2],float EGPlus[nbCell+2],float EGMoins[
      nbCell+2]);
  float power3Over2(float a);
  float getNorme(float h[nbCell+2],float hex[nbCell
      +2],float dx);

#endif
```

Listing 17: The header ”usefull.h”

- **main.c**

  If you want to use the program in order to obtain numerical results, the file **main.c**, with the file **constant.h**, is the most important. Indeed, in this file you select all the options you need to your experiment (initial condition, numerical solver, boundary conditions ...).

  First, you need to define the domain (lower bound and upper bound), the number of cell, the frequence to save data, and the duration of the experiment, everything in the file **constant.h**.

  Then, you need to choose the topography you want for the experiment. In this example, we choose the procedure **noBed** to model a flat bottom. Finally, the program saves the topography in the file ”topo.txt” for display.

```
/* Select the topography you want */
  /*-------------------------------------------------*/
    /*bumpBed(Z,dx);*/
    /*stepBed(Z,dx);*/
    noBed(Z);
    /*constantBed(Z);*/

    /* We save the topograhy in the file topo.txt*/
    fichier3 = fopen("topo.txt","w");
    for(i=0;i<=nbCell+1;i++)
    {
      fprintf(fichier3,"%f ",Z[i]);
    }
    fclose(fichier3);
  /*-------------------------------------------------*/
```

Listing 18: Selection of the topography

If you plan to compare your numerical results with the exact solution, for example in the case of a dam break, the first thing you need to do is to initialize variables which contain the exact solution. In this case, hExact for the exact height of water and huExact for the exact discharge.

```
/* If you want to compute exact solution, first
   initialise
 * hExact and huExact */
  /*-------------------------------------------------*/
    for(i=0;i<=nbCell+1;i++)
    {
      hExact[i] = 2.0 - Z[i];
      huExact[i] = 0.0;
    }
  /*-------------------------------------------------*/
```

Listing 19: Initialization of variables for the exact solution

The next step is to choose the initial condition you want, in this case, a dam break on dry bed.

```
/* Choose the initial condition you want */
  /*-------------------------------------------------*/
    /*damBreakWetBed(hOld,huOld,Z,dx);*/
    damBreakDryBed(hOld,huOld,Z,dx);
    /*lakeAtRest(hOld,huOld,Z);*/
  /*-------------------------------------------------*/
```

Listing 20: Selection of the initial condition

Then, in what follow, we are in the loop on time. The first thing you need to do is to select the right function to calculate the time step. Here, we plan to use the kinetic solver so we choose the function **timeStepKinetic** in order to compute the time step with repsect to the CFL condition for the kinetic scheme. Finally, we do a test because, at the last time iteration, the time step could be such as the time ”tNew” is taller than the duration T. So we calculate again the time step in order to fit with the last time wich is the time T.

```
/* We compute the time step thanks to te CFL condition.
   */
  /*-------------------------------------------------*/
    dt = timeStepKinetic(dx,huOld,hOld);
    /*dt = timeStepRusanov(dx,huOld,hOld);*/
  /*-------------------------------------------------*/

  tNew = tOld + dt;
  if(tNew > T)
  {
    dt = T - tOld;
    tNew = T;
  }
```

Listing 21: Selection of the function to compute the time step

The next step is to choose the solver you want, in this case, the kinetic solver with the procedure **updateKinetic**.

```
/* Choose the solver you want */
  /*-------------------------------------------------*/
      updateKinetic(hNew,huNew,hOld,huOld,Z,rat);
      /*updateRusanov(hNew,huNew,hOld,huOld,Z,rat);*/
  /*-------------------------------------------------*/
```

Listing 22: Selection of the numerical solver

Before choosing the numerical solver, you need to select the boundary condition you want. In this example we choose free boundary conditions with the procedure **updateBC**.

```
/* Choose the boundary conditions you want */
  /*-------------------------------------------------*/
    updateBC(hNew,huNew);
    /*updateBCPeriodic(hNew,huNew,hOld,huOld);*/
    /*updateBCLakeAtRest(hNew,huNew);*/
  /*-------------------------------------------------*/
```

Listing 23: Selection of the boundary condition

The last thing to do, if you want to compare your results with the exact solution, is to select the right exact solution. For our example, we wanted to simulate a dam break on dry bed. In this case, the exact solution is given by the Ritter's solution.

```
/* If you want to compare your results with the exact
    solution
 * choose the right one. */
  /*-------------------------------------------------*/
    Ritter(hExact,huExact,dx,tNew,1.0);
    /*Stoker(hExact,huExact,dx,tNew,2.0,1.0);*/
  /*-------------------------------------------------*/
```

Listing 24: Selection of the exact solution

Then, the program save data at each time iteration in order to repect the frequence you have defined. At each time iteration the program also display the time ”tNew” and the time step. At the end of the running, the program print the number of time iterations and the time of the running in seconds.

- **Makefile**
  To compile the program without to much efforts, a makefile has been created.

The first thing you need to do to compile the program is execute the command **make** in your terminal to creat all the object files and the executable **mainProgram**. Then, to run the program, you need to execute the command **./mainProgram** in your terminal.