girdap — Documentation

version 0

Last generated: August 18, 2016

---

girdap project

Table of Contents

Getting started

Summary: girdap is a c++ based object oriented library for multiphysics simulations on self-managed grids.

Download or clone *girdap*

First download or clone *girdap* from the Github repo (https://github.com/uzgoren/girdap).

Note: It is recommended to download `develop` branch rather than the `master` branch as many described functionalities are available in `develop` branch at this point.

Build *girdap*

Warning: Requires `cmake`

Extract the package into a directory. The girdap's base folder is named as `girdap` as default. It comes with the following directory structure;

```
- girdap
  + src      // *.cpp files are here; also basic main.cpp is pl
aced here;
  + include  // *.hpp files are here;
  + bin      // executables are placed here after build
  + library  // a library w/o main.cpp is placed here after bui
ld
  + example  // various examples of main.cpp can be found her
e;
```

`example` direction contain `main.cpp` files which can utilize *girdap*'s functionality. Some of the tutorials will be placed inside this directory. Develop your own or modify one of them as your `girdap` code and place it in the root of src directory and follow the procedure below to build your code:

```
cd dir_of_your_choice
tar -xzvf girdap.tar.gz
cd girdap
# Make changes to the main_xxx.cpp
# modify CMakeLists.txt to point it to main_xxx.cpp
cmake .
make
```

Now, you can run your code with the executable `girdap` which is placed in the `girdap/bin` directory.

```
bin/girdap
```

Examples/tutorials can be built with the make command:

```
make div          # div is the name of the example
```

On a successful build, the executable will be placed in `girdap/bin` directory with the example's name. For the example above, executable file will be named as `div` .

[]:

Features - Grid

Grid supports the following cell structures:

	line	quad	tri	hexa
Base	✔	✔	✗	✗
Auto domain	✗	`Block2`	✗	✗
h-refinement	✔	✔	✗	✗
Var operators				
+	✗	✔	✗	✗
-	✗	✔	✗	✗
dot product	✗	✔	✗	✗
pde operators				
gradient	✗	✔	✗	✗
laplacian	✗	✔	✗	✗
divergence	✗	✔	✗	✗
time integration	✗	✔	✗	✗

Tutorials

Following examples will help you start using *girdap*. Examples below should be considered as the driving code that use *girdap* library. You can create your own `main.cpp` or modify one of `main_xxx.cpp` files which can be found in the `girdap_rootdir/src` directory.

Any code using *girdap* should include girdap header file placed in `girdap_rootdir` before the `main()` function;

```
#include <girdap_rootdir/grdap>
#include <stdio>

int main(int argc, char *argv[]) {
    // examples should be placed here
    // --> begin here

    // --> end here
    exit(0);
}

```

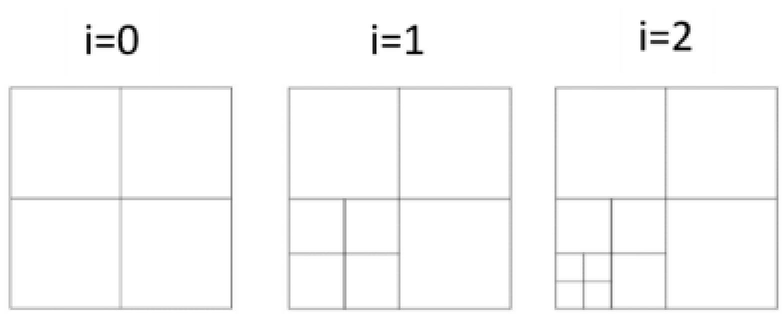
Tutorial - Grids

Your first grid

This example creates a grid handle and manually adds vertices and cells. h-refinement can be applied through adapt flag on the first cell. The output is written in VTK format to be visualized in additional software, i.e. Paraview.

```
Grid* grid = new Grid();
grid->addVertex({ {0, 0, 0}, {1, 0, 0}, {1, 1, 0}, {0, 1,
0} });

grid->addCell( {0, 1, 2, 3} ) ;

for (auto i =0; i<3; ++i) {
   grid->listCell[0]->adapt = {1, 1};
   grid->adapt();
   grid->writeVTK("myFirstGrid_");
}

delete(grid);
```

Above code produces the following result:



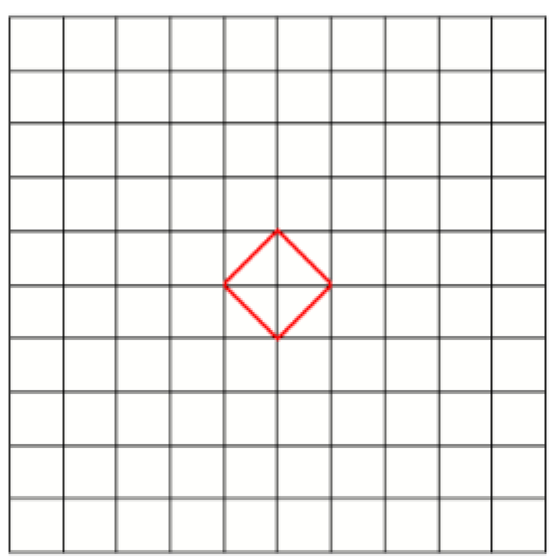
My first grid

Multiple grids (volume & surface)

This example creates two grids; one formed by quads, named as `volgrid`; and the other formed by line strings, named as `surf`. `volgrid` is formed by a an automated block generation utility in 2D, called `Block2`, while `surf` is generated by manually adding vertices and cells. The output is written in VTK format to be visualized in additional software, i.e. Paraview.

```
Block2* volgrid = new Block2({0,0,0}, {1,1,0}, 10, 10);
Grid* surf = new Grid();

surf->addVertex( { {0.5,0.4}, {0.6,0.5}, {0.5,0.6}, {0.4,0.5} } );
surf->addCell( { {0,1}, {1,2}, {2,3}, {3,0} } );

volgrid->writeVTK("vol");
surf->writeVTK("surf");

delete(volgrid);
delete(surf);
```

Above code produces the following result:



My first grid

Field Variables

Variables are 1D arrays; which maintain an order same as the cells/vertices/faces indices in a grid. In addition to values, variables also contain boundary condition information.

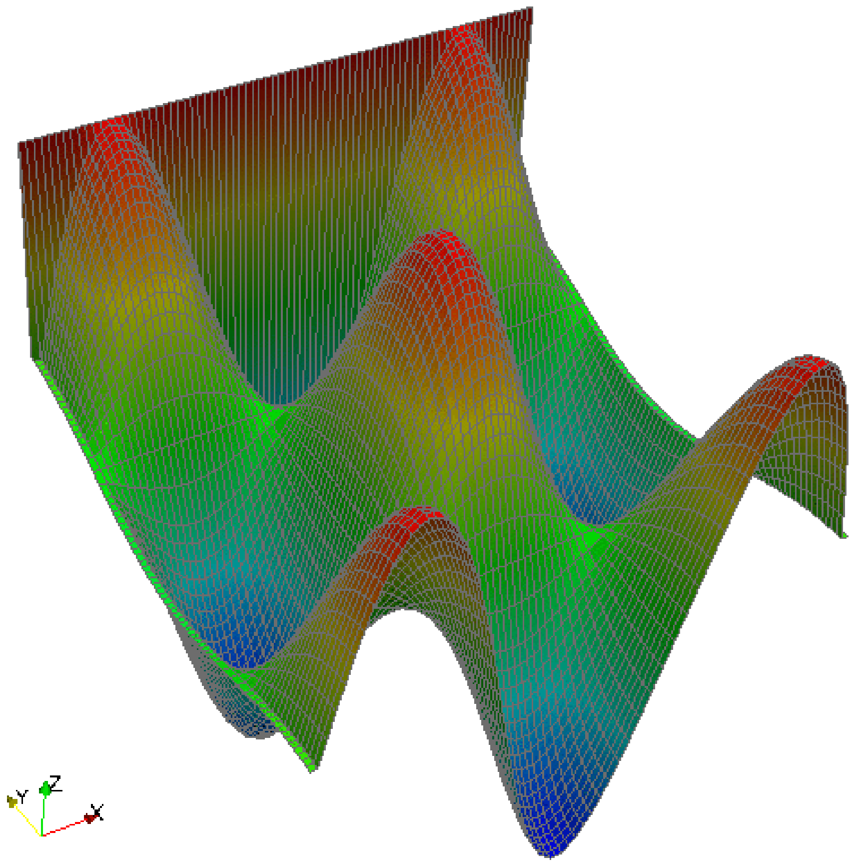
This example creates a 5x5 block using quads in a unit 2D domain. A new variable is defined and named as `f`; where boundary conditions on `east` and `north` are defined using Neumann and Dirichlet conditions. For `south` and `west`, default boundary condition, i.e. zero gradient, is used.

Typical loop over celllist is given in lines xx-xx, where `c->id` gives the array's index.

Grid adaptation based on gradient of the function is used to refine base grid.

```
double pi = 4*atan(1.0);

Block2* volgrid = new Block2({0,0,0}, {1,1,0}, 5, 5);

// add a new variable
volgrid->addVar("f");
auto f = volgrid->getVar("f"); // variable handle

f->setBC("east", "grad", 0);   // This is the default
f->setBC("north", "val", 1);   //

for (auto i=0; i < 4; ++i) {
  for (auto c : volgrid->listCell) {
    auto x = c->getCoord(); // cell-centers
    f->set(c->id, sin(3*pi*x[0])*cos(2*pi*x[1]));
  }
  volgrid->solBasedAdapt2(volgrid->getError(f));
  volgrid->adapt();
  volgrid->writeVTK("field_");
}

delete(volgrid);
```

Above code produces the following result:

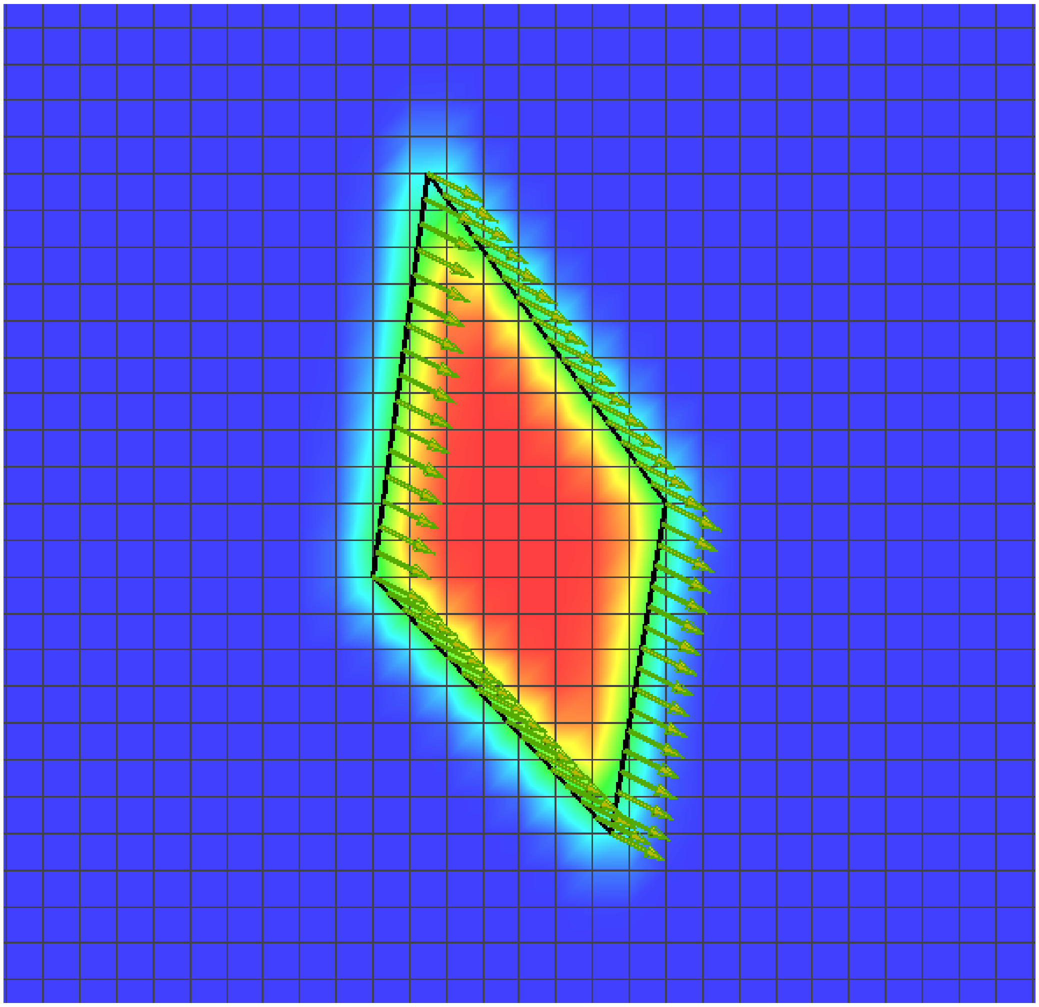


Field variable assigned by function and refined based on error.
Boundary conditions are reflected to the solution field.

Communication between grids

This example generates a volume and a surface grid. Those two are connected through each other by `updateOtherVertex(..)` method. The velocity assigned to volume grid is passed to the surface grid nodes, which are formed after h-refinement. Also an indicator function is generated using the location of the surface.

```
 Block2* volgrid = new Block2({0,0,0}, {1,1,0}, 50, 50);

   // Velocity field
   auto uv = volgrid->getVar("u"); auto vv = volgrid->getVar("v");
   uv->set(1.0); // set velocity
   vv->set(-0.5); // set velocity
   // New variable at cell center
   volgrid->addVar("f"); auto f = volgrid->getVar("f");

   Grid* surf = new Grid();

   surf->addVertex( { {0.55,0.32}, {0.58,0.5}, {0.45,0.68}, {0.42,0.46} } );
   surf->addCell( { {0,1}, {1,2}, {2,3}, {3,0} } );
   // Refine cell;
   for (auto i=0; i<4; ++i) {
     for (auto c: surf->listCell) if (c->vol().abs() > 0.02) c->adapt[0] = 1;
     surf->adapt();
   }
   volgrid->updateOtherVertex(surf);
   // mark location of this surface
   volgrid->indicator(surf, f);

   // Assign velocity variables to surface at vertex
   surf->addVec("u",1);

   // Get velocity on the surface
   auto us = surf->getVar("u"); auto vs = surf->getVar("v");
   volgrid->passVar(surf, uv, us);
   volgrid->passVar(surf, vv, vs);

   volgrid->writeVTK("vol");
   surf->writeVTK("surf");

   delete(volgrid);
   delete(surf);
```



Velocity vectors are transferred from volume grid to surface grid; and indicator function is created purely using surface grid locations.

Heat equation

Heat equation in the following form is to be solved on a unit domain:

$$\vec{\nabla} \cdot (k \vec{\nabla} T) + \dot q_{o} = 0$$

exposed to following boundary conditions:

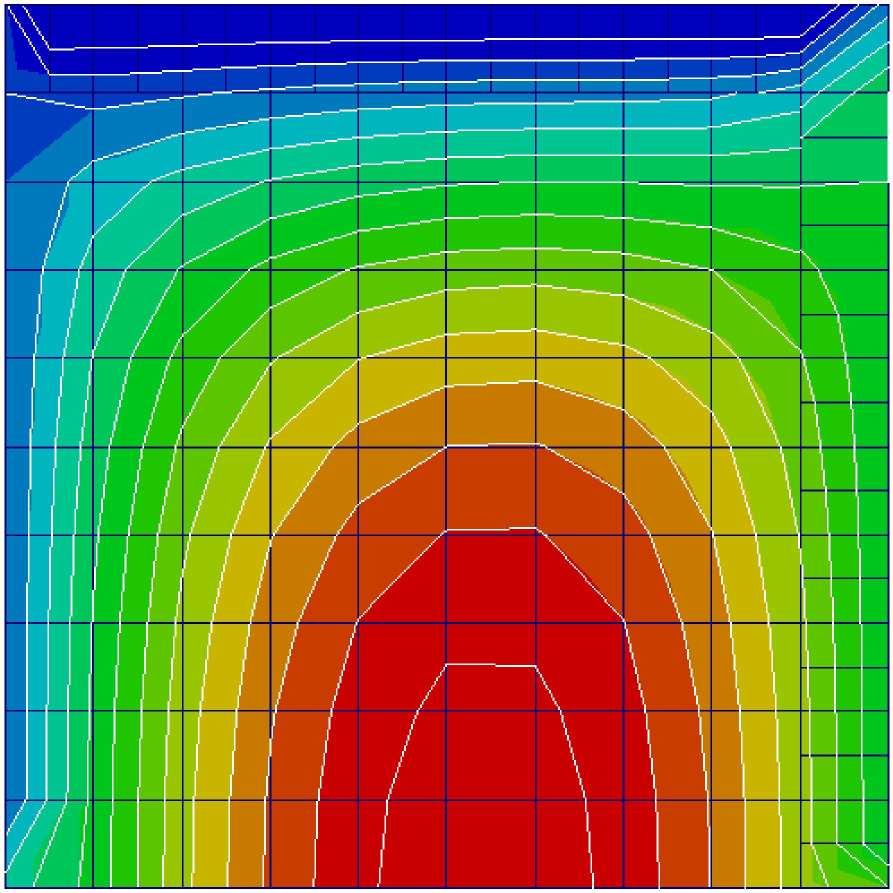
- Top wall:
- Bottom wall:
- Right wall:
- Left wall:

with the following set of parameters:

, , ,

```
// Problem parameters
auto k = 2.0; auto qdot = 5e3; auto h = 50; auto Tinf = 20;
// Grid
Block2* grid = new Block2({0, 0, 0}, {1, 1, 0}, 10, 10);
grid->levelHighBound[0] = 2;
grid->levelHighBound[1] = 2;
grid->addVar("T");
// Variables
auto T = grid->getVar("T");
// Linear solver
T->solver = "BiCGSTAB";
T->itmax = 1000;
T->set(100);
// Boundary conditions
T->setBC("south", "grad", 0);
T->setBC("north", "grad", -h/k*Tinf, h/k);
T->setBC("east", "val", 200);
T->setBC("west", "val", 100);

for (auto i = 0; i< 4; ++i) {
  grid->solBasedAdapt2(grid->getError2(T), 2e-3, 2e-1);
  grid->adapt();

  // Equation
  grid->lockBC(T);
  T->solve( grid->laplace(k)
              + grid->source(0, qdot) );
  grid->unlockBC();

  grid->writeVTK("heat");
}

delete(grid);
```

Above code produces the following result:



Heat equation solved starting at 5x5 Cartesian grid and refined based on gradient.

Grid - Introduction

Summary: Introducing surface and volume grids

Object: `Grid`

`Grid` object mainly consists of two arrays of `shared_ptrs` . One is for `Vertex` for position vectors and the other for `Cell` for connectivity information. Grid does not impose a particular cell structure; it can consist of different cells.

Constructor

`grid()` : forms blank grid. It can be used for variable decleration. The constructor however sets default minimum and maximum levels for h-refinement; and cfl number.

`grid( { coord } )` : (coord -> x, y, z) forms a grid with a single vertex.

`grid( { {coord0}, {coord1}, ...} )` : forms a grid with many vertices. Connectivity information is not yet defined.

`grid( { {coord0}, {coord1}, ...}, { {cell0}, {cell2}, ... } } )` : forms a grid with many vertices and cells. *cell0*, *cell1* are integers pointing out the order of vertex in the first list, starting from 0.

Grid - Block1

Summary: 1D grid blocks

Object: `Block1`

`Block1` inherits all methods of `Grid` while it mainly focuses on generating/manipulating 1D grids formed by `Line` cells.

Constructor

`Block1()` : generates Grid of Block1 type

`Block1({ coord1 }, { coord2 }, N)` generates a line between *coord1* and *coord2* which is made up of N cells; all at a level marked as 0. This means that the grid can not be further coarsened through h-refinement.

`Block1( geo, { {param0}, {param1}, ... }, nx )` generates a predefined geometry which is charaterized by parameters listed next; and using h-refinement approximately *nx* cells are formed.

geo can have two options:

- “poly”: *param#* are simply coordinates of connected lines. This can be used to define any polygon with arbitrary number of sides when the first and last parameters point to the same coordinate. If they are different, then the polygon is left unclosed.
- “arc”: requires *param0* to be center; first number in *param1* to be radius; first number in *param2* is start angle in degrees and second number in *param2* is end angle in degrees. If *param2* is omitted, a full circle will be formed. Degrees can be placed in increasing or decreasing form, to yield counter-clockwise or clockwise trace; which alters the normal direction to point outside or inside, respectively.

Methods

`merge( Block1 )` merges one Block1 with another.