girdap — Documentation

version 0

Last generated: August 16, 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 does include two subdirectories; `src` and `include`. `src` contains different versions of c++'s `main()` function; each with a different purpose. These are files those utilize *girdap*'s functionality; which are available in the `include` directory. Develop your own or modify one of the `main_xxx.cpp` files as the driver file and make sure that `CMakeLists.txt` file to make sure that line at the end that starts with `add_executable` points to your driver file (`main_xxx.cpp`) in `girdap/src` folder. Now, you can go ahead with `cmake` and `make` commands as usual.

```
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 named as `girdap` :

```
./girdap
```

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);
}

```

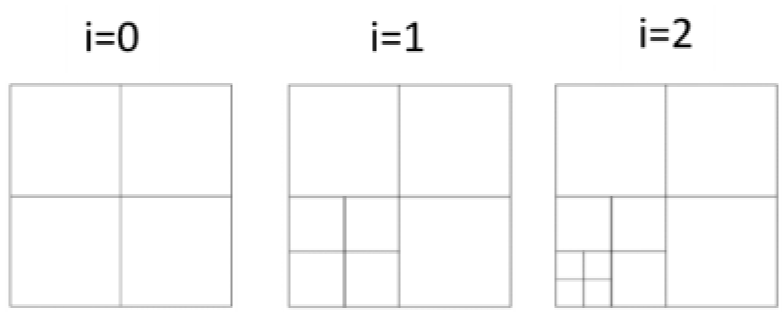
Tutorial - Grids

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:



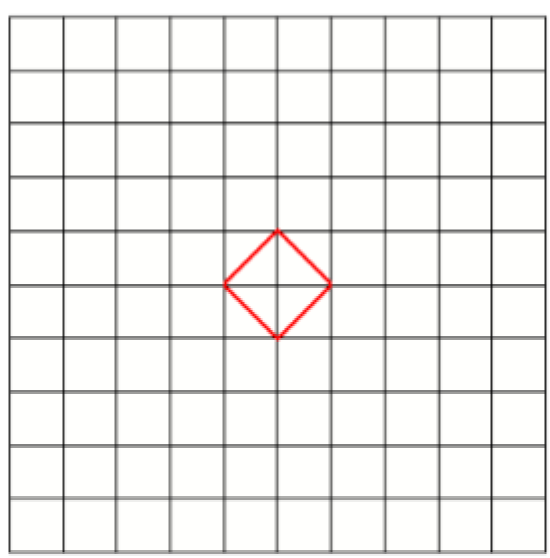
My first grid

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.

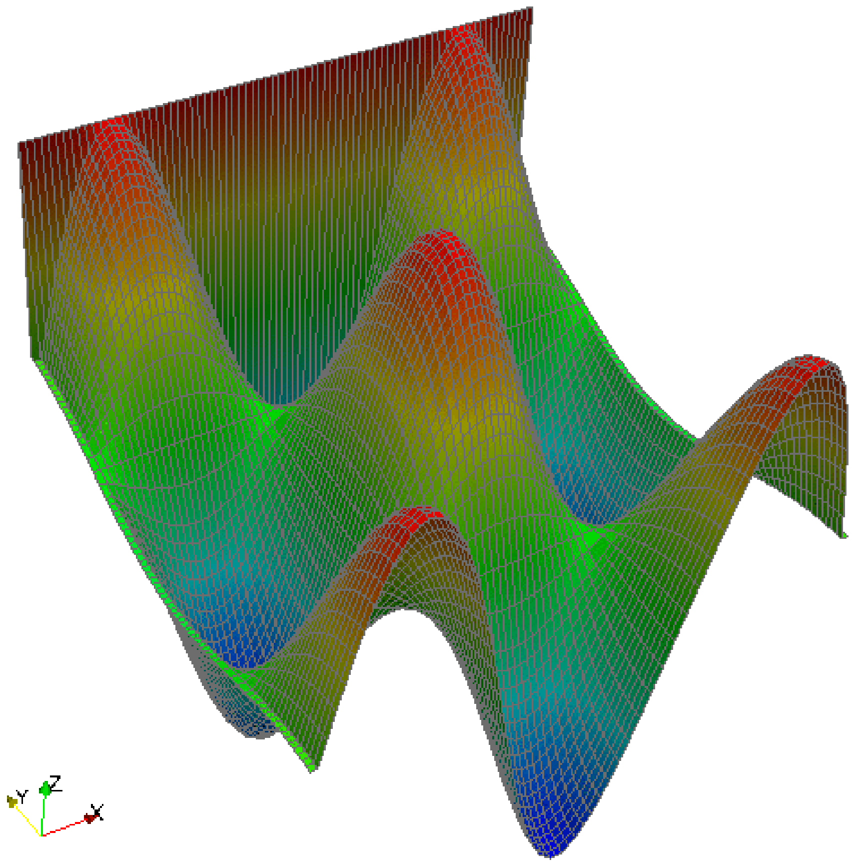
This example creates a 5x5 block using quads in a unit 2D domain. A new variable is defined and named as `f`; where boundary conditions on `east` and `north` are defined using Neumann and Dirichlet conditions. For `south` and `west`, default boundary condition, i.e. zero gradient, is used.

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:

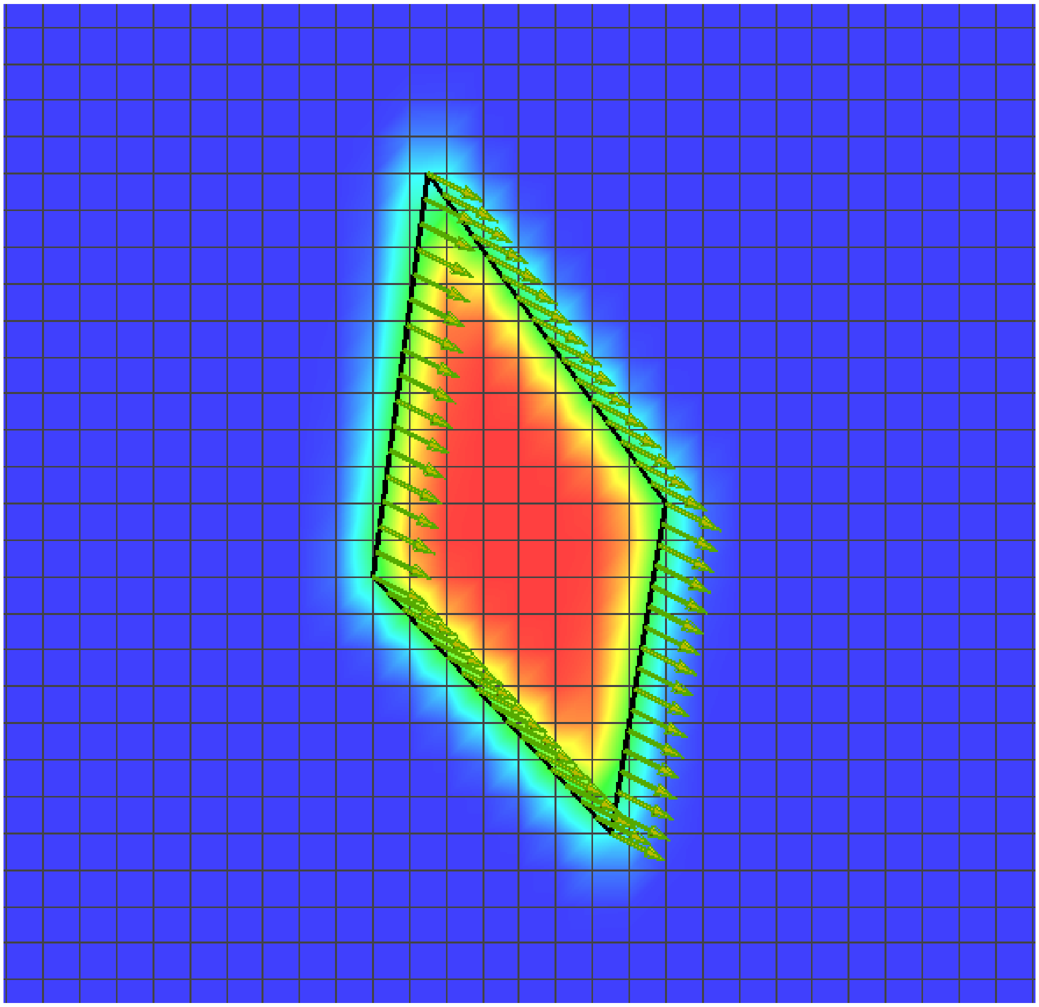


Field variable assigned by function and refined based on error. Boundary conditions are reflected to the solution field.

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:

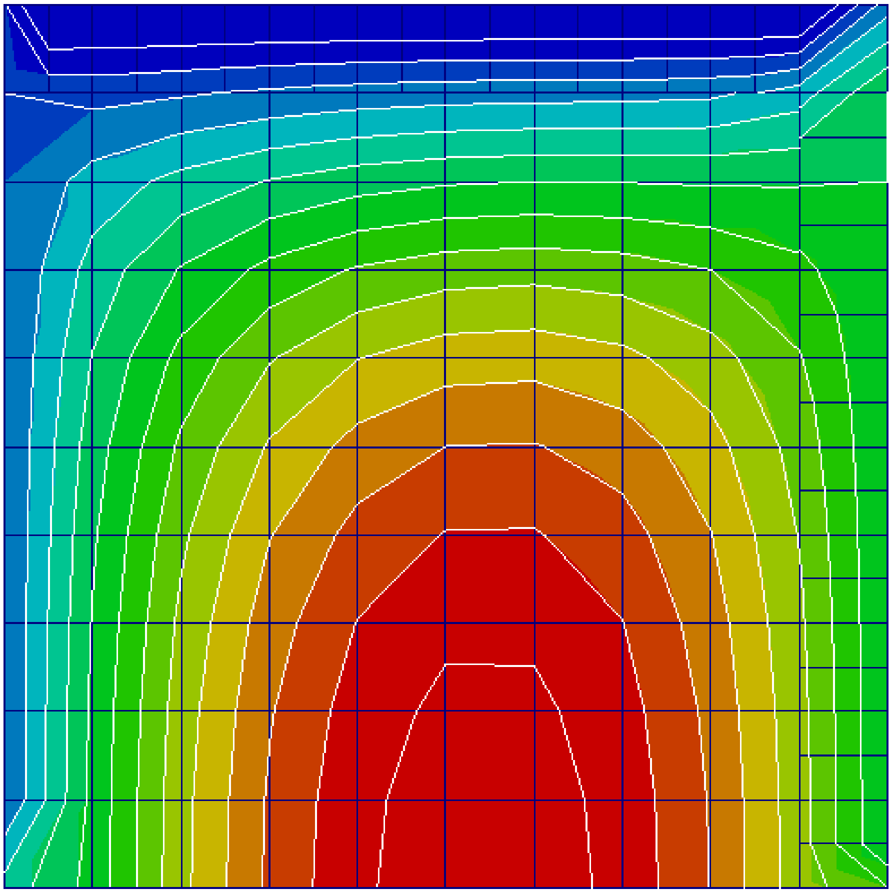
- Top wall:
- Bottom wall:
- Right wall:
- Left wall:

with the following set of parameters:

, , ,

```
// Problem parameters
auto k = 2.0; auto qdot = 5e3; auto h = 50; auto Tinf = 2
0;
// 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.