girdap — Documentation

version 0

Last generated: August 22, 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);
}

```

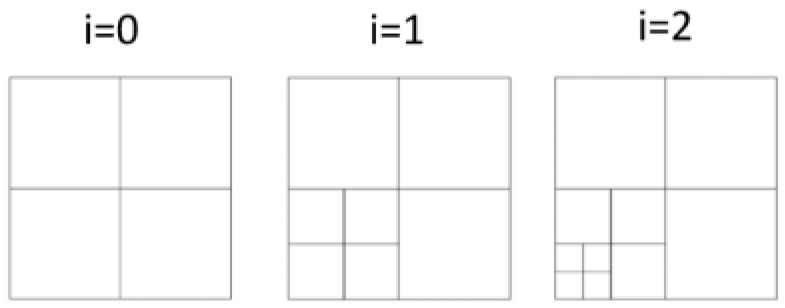
Tutorial - Grids

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:



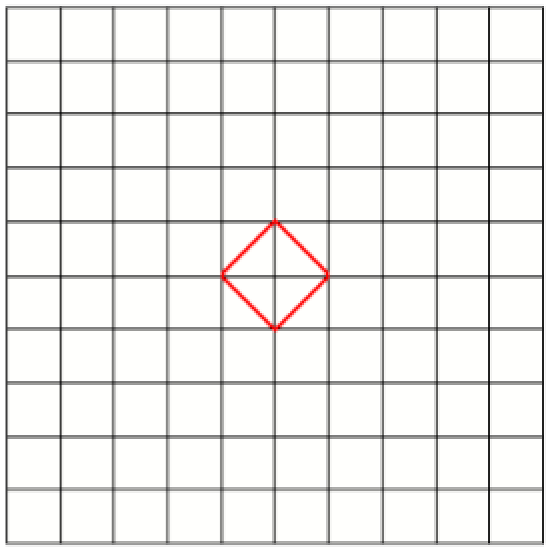
My first grid

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.

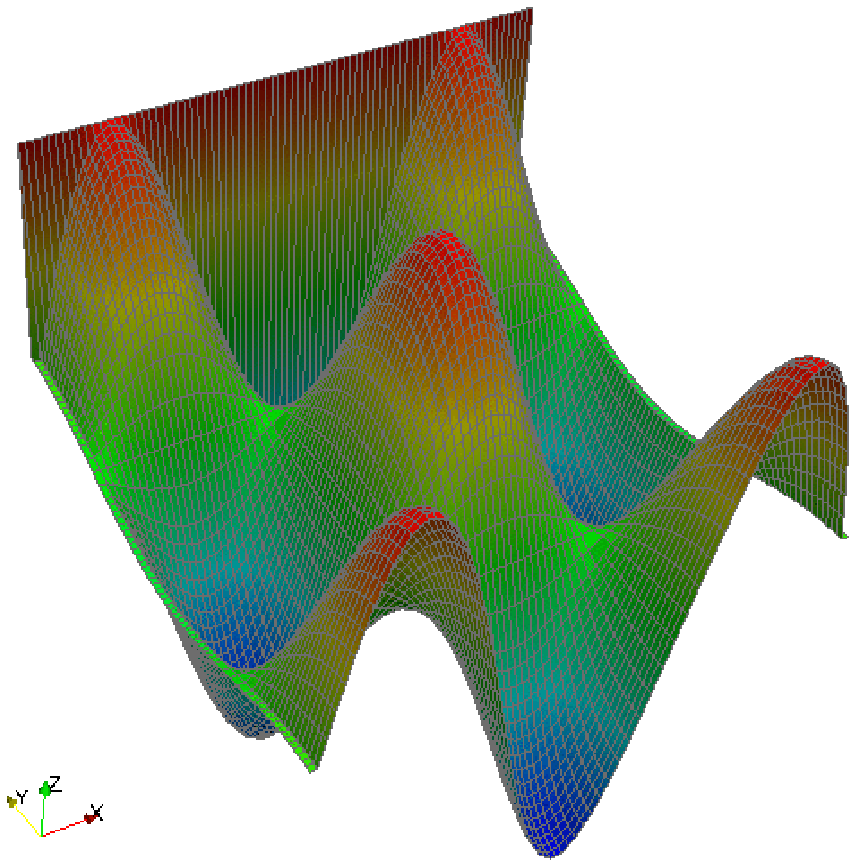
This example creates a 5x5 block using quads in a unit 2D domain. A new variable is defined and named as `f`; where boundary conditions on `east` and `north` are defined using Neumann and Dirichlet conditions. For `south` and `west`, default boundary condition, i.e. zero gradient, is used.

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.

```
 1  double pi = 4*atan(1.0);
 2
 3  Block2* volgrid = new Block2({0,0,0}, {1,1,0}, 5, 5);
 4
 5  // add a new variable
 6  volgrid->addVar("f");
 7  auto f = volgrid->getVar("f"); // variable handle
 8
 9  f->setBC("east", "grad", 0);   // This is the default
10  f->setBC("north", "val", 1);   //
11
12  for (auto i=0; i < 4; ++i) {
13    for (auto c : volgrid->listCell) {
14      auto x = c->getCoord(); // cell-centers
15      f->set(c->id, sin(3*pi*x[0])*cos(2*pi*x[1]));
16    }
17    volgrid->solBasedAdapt2(volgrid->getError(f));
18    volgrid->adapt();
19    volgrid->writeVTK("field_");
20  }
21
22  delete(volgrid);
```

Above code produces the following result:

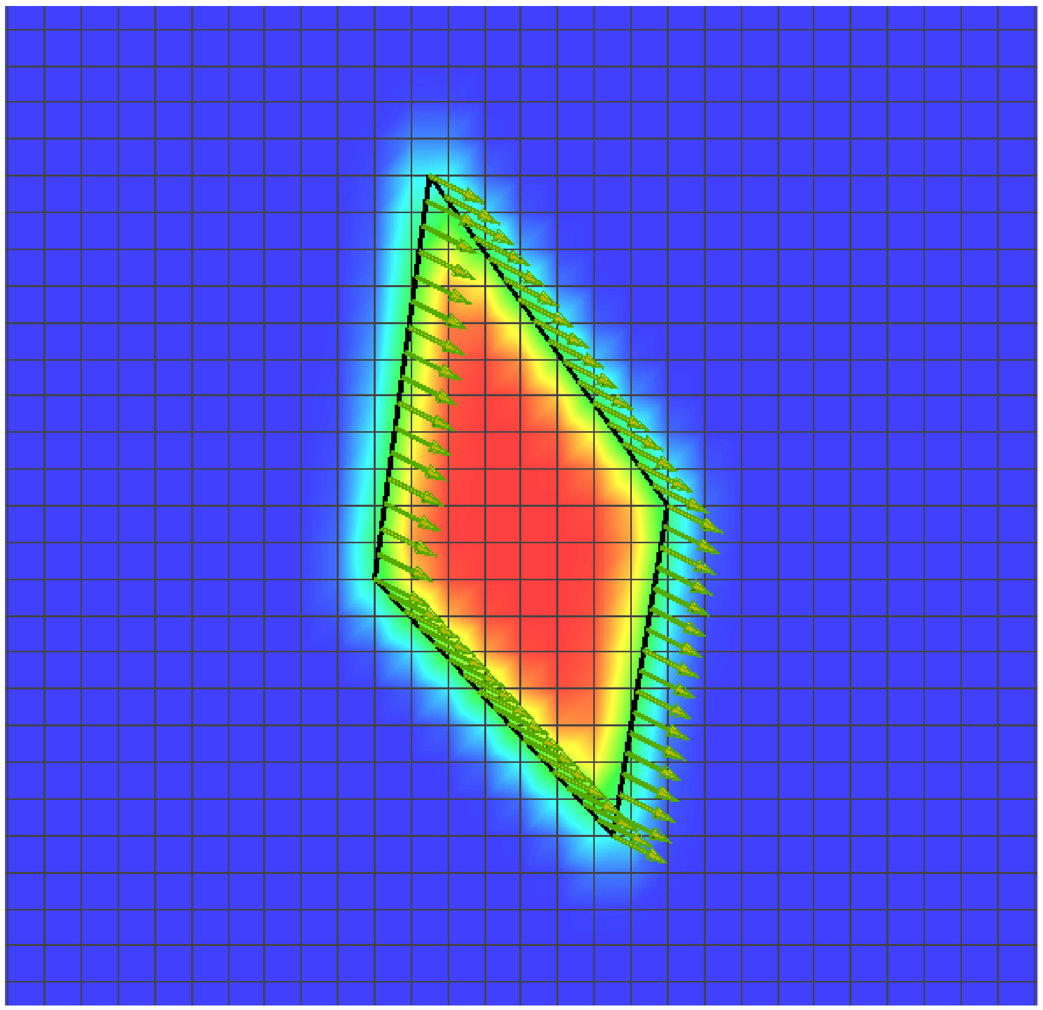


Field variable assigned by function and refined based on error.
Boundary conditions are reflected to the solution field.

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.

```cpp

 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:

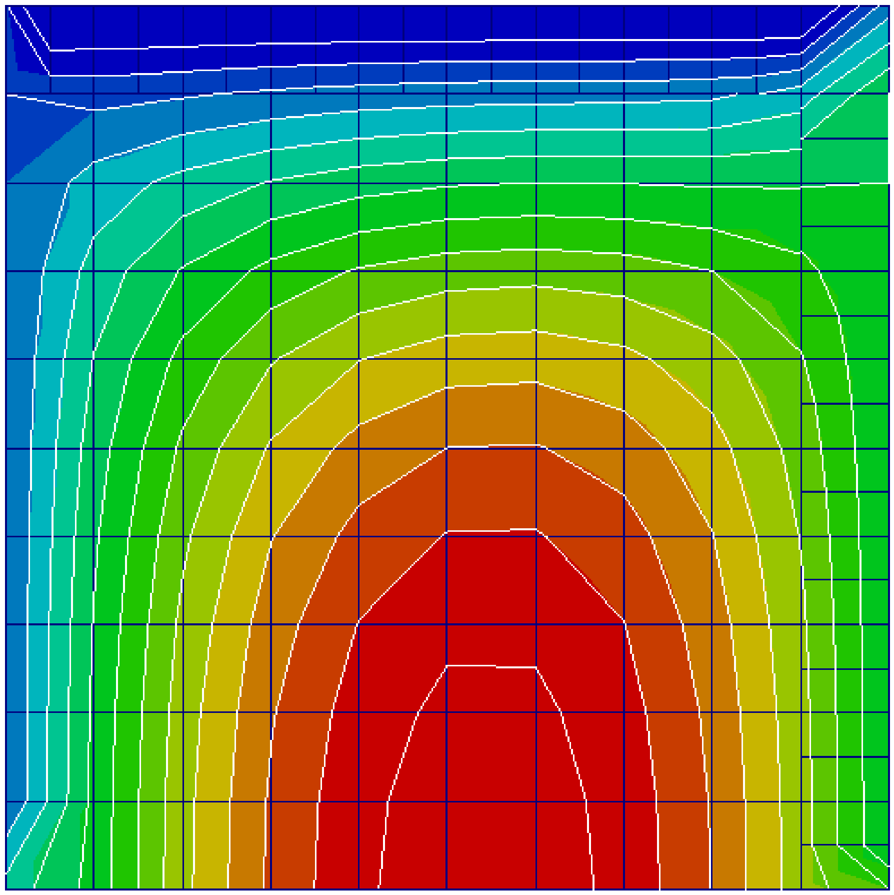
- Top wall:
- Bottom wall:
- Right wall:
- Left wall:

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.

Geometry - Geo1

Summary: 1D basic geometric definitions

Object: Geo1

`Geo1` is a special class equipped with a parametric function to describe simple geometries.

The function accepts parametric variable, `s` (a `double`) to yield a vector in space (a `Vec3`). The start and end values of `s` completes geometric definition.

- `s0` : Lower bound of the parametric variable (0 by default)
- `s1` : Upper bound of the parametric variable (1 by default)
- `f` : function in terms of lambda (See c++11 tutorials like Dr. Dobb's (http://www.drdobbs.com/cpp/lambdas-in-c11/240168241))

Proper way to define lambda is as follows:

```
function<Vec3 (double)> f = [=] (double s)->Vec3 {return Vec3(x(s), y(s), z(s));}
```

Constructors

`Geo1()` : Creates a geometric definition using default values; (a line between and);

`Geo1(f)` : Creates a geometric definition using a new function defined by the default limits

For example, following defines a heart shape;

```
Geo1 heart( [](double s)->Vec3 {
                                auto t = 4*atan(1.0)/180*s;
                                return Vec3( (1-sin(t))*cos(t), (1-sin(t))*sin(t));
                             } );
```

`Geo1(s0, s1, f)` : Creates a geometric definition using a new function and new limits:

```
Geo1 heart(0, 8*atan(1.0), [=](double s)->Vec3 {
                                    return Vec3( (1-si
n(t))*cos(t), (1-sin(t))*sin(t));
                    } );
```

`heart.f(1.0)` returns a vector with components, and .

Subclasses

Following subclasses consists of the predefined functions (and limits) for simple geometries;

- `Geo1Line(x0, x1)` defines a line between vectors (`Vec3`) and
- `Geo1Circle(x0, r)` defines a full circle (in CCW direction) with a center at and a radius of
- `Geo1Circle(x0, r, d0, d1)` defines an arc from to (in degrees) with a center at and a radius of
- `Geo1Sine(x0, x1, a, f)` defines a sinusodial function between x0 and x1, with an amplitude of a and a frequency of f;

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.

Methods

Grid - Block1

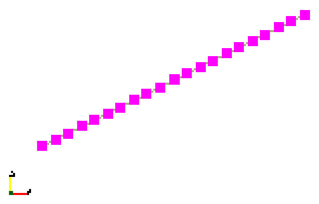
Summary: 1D grid blocks

Object: `Block1`

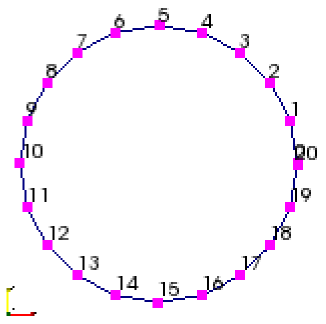
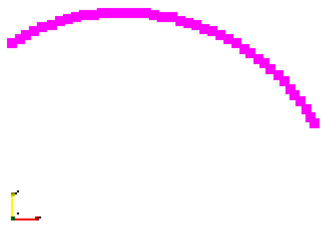
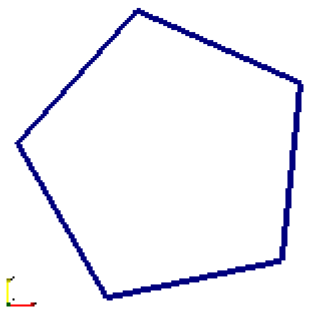
`Block1` inherits all methods of `Grid` while it mainly focuses on generating/manipulating 1D grids formed by `Line` cells.

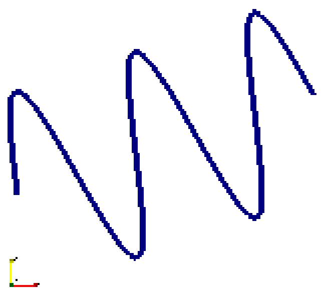
Constructor

- `Block1()` : generates Grid of Block1 type
- `Block1({ coord1 }, { coord2 }, N)` generates a line between and which is made up of cells; all at a level marked as 0. By default, the cell-levels cannot go below 0. However, one can change that if masters in each direction are defined (-not defined).

Code	`Block1 lin({0.2, 0.3, 0}, {0.8, 0.6, 0}, 20)`
Line	

- `Block1(geo, N, <d0>, <d1>)` generates a 1D array in the shape discribed by the geometry (see Geo1 (page 17) using cells. Optional parameters and provide a new definition for start and end points for the arc in degrees. Note that . Following are some examples on the usage:

Code	`Block1 circ(new Geo1Circle(Vec3(0.5, 0.75), 0.15), 40);`
Full circle (in CCW)	
Code	`Block1 arc(new Geo1Circle(Vec3(0.5, 0.75), 0.15, 30, 120), 40);`
Arc ()	
Code	`Block1 rotpenta(new Geo1Circle(Vec3(0.5,0.75), 0.15, 30, 400), 5);`
Pentagon (in CCW and rotated by)	

Code	`Block1 line2(new Geo1Sine(Vec3(0.2, 0.4), Vec3(0.5, 0.5), 0.1, 5), 50);`
Sinusodial	

Note that one needs to make sure that the geometry created by `new` command needs to be properly destroyed. Even though above examples provide one line statements; it is better to follow a proper decleration of a geometry which deleted from the memory by a proper `delete` command as follows:

```
  auto tmp_geo = new Geo1Sine(Vec3(0.2, 0.4), Vec3(0.5, 0.5), 0.1, 5);
  Block1 line2(tmp_geo, 50);
  delete(tmp_geo);
```

- `Block1(, delta)` produces connected lines between all supplied coordinates . If the first and the last coordinates are the same, it closes the loop properly. If is provided (non-zero) then cells are refined until all have a length smaller than . This is especially useful if a geometry needs to be defined purely based on coordinates; i.e. NACA airfoils.

Specific methods

- `resolve(delta)` brings all cell lengths to a level closer to (a double).
- `add(Geo, N)` adds a new geometry which is made up of cells to an existing geometry. If first vertex point of the added geometry is the same as the last vertex of the existing geometry, new geometries become linked through those vertices. Otherwise, a separate geometry is combined into the existing one.

- `add(Block1)` adds vertices and cells into the one this method is called from.
- `add(s0, s1, N, f)` adds a new geometry defined by the parametric function f (similar to Geo) to the current geometry

Following methods are yet to be defined:

- `addVol(Block1)` will involve merging areas enclosed by two grid (Block1 and current grid). Requires changing topologies.
- `subVol(Block1)` will involve removing areas enclosed by two grids; i.e. removing a triangular pie from a circle.
- `trans(dir, distance)` will involve translating vertex points in a particular direction () by given distance ().
- `rot(x0, norm, theta)` will involve rotating vertex point locations along an axis passing through in the direction of () by an angle prescribed by .
- `rotate(x0, norm, theta, N)` will create a 3D surface by rotating cells around a axis passing through in the direction of () by an angle prescribed by . The rotation angle will be discretized by cells.
- `extrude(norm, double dist, N)` will create a 3D surface by extruding cells in the direction defined by by a length defined by . There will cells in the direction of extrusion.