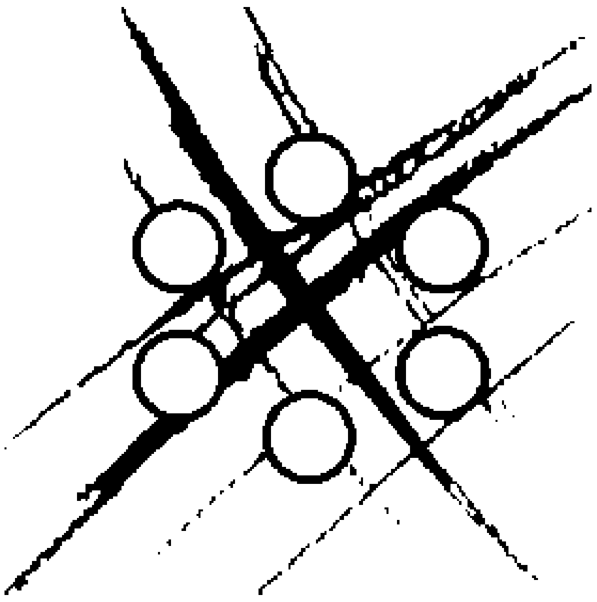
DIRECTION DE L’ÉNERGIE NUCLEAIRE

DÉPARTEMENT MODÉLISATION DE SYSTÈMES ET STRUCTURES

SERVICE FLUIDES NUMÉRIQUES, MODÉLISATION ET ÉTUDES



RAPPORT DM2S

SFME/LGLS/RT/06-005/A

SALOME component integration tutorial

Vincent Bergeaud

COMMISARIAT A L'ENERGIE ATOMIQUE

DIRECTION DE L'ENERGIE NUCLEAIRE
DEPARTEMENT MODELISATION DE SYSTEMES ET STRUCTURES
SERVICE FLUIDES NUMÉRIQUES, MODÉLISATION ET ÉTUDES
LABORATOIRE DE GÉNIE LOGICIEL ET SIMULATION

RAPPORT DM2S

REFERENCES : SFME/LGLS/RT/06-005/A

TITRE : SALOME component integration tutorial

AUTEURS	SIGNATURES	AUTEURS	SIGNATURES
Vincent Bergeaud	[illegible]		

RESUME : Ce document est un didacticiel permettant à un développeur de prendre en main rapidement la plate-forme Salomé et la procédure d'intégration d'un code extérieur.
Il est fondé sur l'exemple de l'intégration du code OpenSource RHEO-LEF.

MOTS CLES : Salome, intégration, composants, python, C++, codes de calcul, didacticiel, tutorial
AFFAIRE : DSOE/Simulation Projet : NPPAL EOTP : A-NPPAL-02-01
Titre de l'action : Environnement PAL

			Visa	[illegible]		[illegible]
			Nom	M. Tajchman		L. Dada
A	21/07/2006	37	Date	21/07/2006		21/07/2006
Indice	**Date**	**Nb. Pages**		**Vérificateur**	**Autre visa**	**Approbateur**

Les informations contenues dans ce document sont réservéees aux destinataires nommément désignés et ne peuvent recevoir aucune diffusion sans l'autorisation expresse du DM2S

Liste des modifications

Indice	Date	Motif et description de la modification
A	21/07/2006	Document initial

Table des matières

1 Introduction

1.1 Outline of the tutorial

This tutorial presents the integration of a new computational component in the SALOME platform. Rather than a complete description of the functionalities of the platform, it focuses mainly on the interfacing of SALOME with an external code/library. More detailed information about the different SALOME modules (CAD, Meshing, Supervision, PostProcessing) can be found in [DR3], [DR4].

The tutorial is divided in five steps, which are detailed in the following sections :
– the development of an interface between SALOME structures and the external code (section 2);
– the creation of a new SALOME component via the HXX2SALOME wrapper generator (section 3);
– the use of SALOME via an interactive session (section 4);
– the use of SALOME via a Python script (section 5);
– the creation of a GUI for the new SALOME component (section 6).
These steps will be illustrated through an example. As an external code, we will use RHEOLEF, an open source finite-element library [DR5]. The sources corresponding to this tutorial can be found on the Web Site [DR8].

1.2 Reference documents

[DR1]	User's Guide of Med Memory, P. Goldbronn, E. Fayolle, N. Bouhamou, J. Roy and N. Crouzet, 26th February 2004
[DR2]	hxx2salome : a Salome component generator, N. Crouzet
[DR3]	SALOME Platform Web documentation, http://www.salome-platform.org/docfaq/
[DR4]	SALOME online help
[DR5]	RHEOLEF Web Site, http://www-lmc.imag.fr/lmc-edp/Pierre.Saramito/rheolef/
[DR6]	Projet PAL : Definition du modele d'échange de donnees MED V2.2, EDF R&D, V. Lefebvre, E. Fayolle, HI-26/03/012, December 2003
[DR7]	SALOME GUI Tutorial, to be published
[DR8]	Sources of the tutorial, HAMMI Web Site, https://hammi.extra.cea.fr

1.3 Prerequisites

The execution of the following tutorial requires an installation of **Salome v 3.2.0** and **RHEOLEF v 5.17**. As the `dl_open` mechanisms in RHEOLEF made it incompatible with SWIG Python wrappings, it was necessary to modify a few `Makefile.am` in the SRHEOLEF distribution, so that RHEOLEF modules are linked against libSRHEOLEF.so. In addition, we derive classes from RHEOLEF classes, so a header file that is not normally installed is required (`geo-connectivity.h`). Therefore, the modified sources are given for convenience in the tutorial archive. RHEOLEF installation process is unchanged. It should be noted that RHEOLEF requires `boost` with version higher than 1.33 and `automake` with version higher than 1.9.6. The two following lines suffice to configure and install the library :

```
./configure --prefix=${RHEOLEF_INSTALL_DIR}
make install
```

2 Development of a standalone component

2.1 Presentation of the functionalities of the component

The RHEOLEF library is written in C++ and provides implementation of high level finite element concepts so that tens of lines of source code are sufficient to create and solve a variety of problems. The main notions used by RHEOLEF are geometries (meshes and regions), spaces (polynomial basis), fields (series of values defining a function for a particular space) , and forms (linear forms for a given space). As an example, the following code uses RHEOLEF to create a Poisson problem with Dirichlet boundary conditions :

```
#include "rheolef.h"
using namespace std;

int main(int argc, char**argv) {
    geo omega (argv[1]);
    space Vh (omega, argv[2]);
    Vh.block ("boundary");
    form a (Vh, Vh, "grad_grad");
    form m (Vh, Vh, "mass");
    field fh (Vh, 1);
    field uh (Vh);
    uh ["boundary"] = 0;
    ssk<Float> fact = ldlt(a.uu);
    uh.u = fact.solve (m.uu*fh.u + m.ub*fh.b - a.ub*uh.b);
    cout << uh;
    return 0;
}
```

In order to illustrate different aspects of the SALOME platform, we use RHEOLEF for solving a transitory heat equation problem, a stationary heat equation problem and an elastic displacement problem. Therefore, our interface is made of the following class :

```
#ifndef _SRHEOLEF_HXX_
#define _SRHEOLEF_HXX_


#include "MEDMEM_Field.hxx"
```

```
#include "MEDMEM_Mesh.hxx"
#include <map>

class HeatEquation;
class FieldCollection;

class SRHEOLEF
{

// Méthodes publiques
public:

  SRHEOLEF();

  virtual ~SRHEOLEF();

  //Sets the Mesh
  void setMesh(const MEDMEM::MESH* mesh);

  //Sets the Mesh from a filename and a mesh name
  void setMeshFromFile(const std::string& filename, const std::string& meshname);

 //Computation of a stationary heat equation with Dirichlet bc
  MEDMEM::FIELD<double>& computeDirichlet();

  //Computation of an elasticity problem
  void computeThermoElasticity(const MEDMEM::FIELD<double>* temperature);

  //sets a Dirichlet boundary condition on boundary named bc
  void setDirichlet (const char* bc, double alpha);

  //sets a blocked displacement condition on boundary bc
  void setBlockedDisplacement (const char* bc);

  //creates the objects for the heat equation
  void heatEquationInit();

  //advances heat equation by one time  step
  void heatEquationTimeStep(double dt);

  //destroys the problems
  void heatEquationDestroy();

  //! stores the result in a file
  void dump(const std::string& name);
private:
  //Mesh Structure
  MEDMEM::MESH* m_mesh;

  //interface between MEDMEM field and RHEOLEF field
  FieldCollection* m_temperature_field;

  //interface between MEDMEM field and RHEOLEF field
  //for displacement fields
  FieldCollection* m_displacement_field;

  //list of Dirichlet boundary conditions for temperature
  std::map<string,double> m_dirichlet_bc;

  //list of Neumann boundary conditions for temperature
  //(with zero flux)
  std::set<string> m_neumann_bc;

  //list of blocked displacement boundary conditions for
  //thermoelastic displacement computation
  std::set<string> m_blocked_displacement_bc;

  //Heat Equation algorithm
  HeatEquation* m_heat_equation;

  //flag for memory management
  bool m_owns_mesh;
};

#endif
```

With this interface, two classes of problems can be addressed : heat diffusion and elasticity. We will use this to illustrate the coupling capabilities of SALOME.

This interface enables the user to :
– load a mesh definition from a SALOME MED-file (MED stands for Data Exchange Format);
– set Dirichlet boundary conditions on some regions of the mesh for the heat equation problem;
– set blocked displacement boundary conditions for the elasticity problem;
– compute a time step for the heat equation;
– compute the solution of the stationary heat equation problem;
– compute the displacements for the elasticity problem;
– write resulting fields in a MED-file.

The next subsections focus on the first and last steps, which involve coupling of SALOME structures to RHEOLEF structures, and which are typical of the work that has to be done for the integration of an external code to SALOME.

2.2 Loading meshes with SALOME

SALOME modules communicate via a dedicated format named MED (Data Exchange Format). This format comes with two flavours : MED-file and MED-memory, whether the interaction is made via a file or directly through memory. Here, we will not consider the possibility of directly reading MED-files into the external code, and we will explain how to load data via a MED-memory object.

2.2.1 Loading a MED-file in memory

On this topic, more information can be found in document [DR1] (MEDMEM user guide).
Here follows the implementation of our `setMeshFromFile` method.

```
 // Sets the Mesh from a file
void SRHEOLEF::setMeshFromFile(const std::string& filename,
const std::string& meshname)
{
try {
  m_mesh=new MEDMEM::MESH(MEDMEM::MED_DRIVER,
  filename,
  meshname);
}
catch(MEDMEM::MEDEXCEPTION& ex){
 MESSAGE(ex.what());
                 throw;
}
m_owns_mesh=true;
}
```

Note :
– the namespace MEDMEM for access to MED-memory structures;
– the specification of a mesh name in addition to the filename (MED files can contain several meshes);
– re-throwing an exception for management by the SALOME kernel;
– in this case, memory management ensured by the SRHEOLEF interface.

2.2.2 Exploiting MED-memory query methods

In the RHEOLEF code, meshes are contained in a class named `geo`. For the standalone use of RHEOLEF, a set of drivers is provided for reading files in different formats and filling up the `geo` structures (coordinates, connectivity, regions,...).
In our case, we directly use the data of the `MEDMEM::MESH` object. We created a new class which derives from the RHEOLEF `geo` class and which is named `geo_read_from_med_mem`. Its constructor has a `MEDMEM::MESH` structure as an argument.
The queries to the `MEDMEM::MESH` object are described in document [DR1]. In our example, we made the building of `geo_read_from_med_mem` only available for 3D meshes made out of tetraheadra. The RHEOLEF

format requires nodal connectivities for the tetrahedra, the triangles and the segments of the mesh. However, MEDMEM gives only access to the nodal connectivities for elements of dimension d and $d - 1$ (where d is the mesh dimension). Therefore, we had to reconstruct the edge nodal connectivity.

In addition, the connectivity that is used in MEDMEM does not fit the one used by RHEOLEF, so that we had to revert tetrahedra in order to retrieve positive element volumes. Connectivities used in MEDMEM are described in [DR6], which describes the MED Data Exchange Format.

Finally, connectivity tables start at 1 in MED, whereas they start in a C fashion at 0 in RHEOLEF.

Our code is given in appendix.

The constructor of our class comes in six steps, which illustrate typical use of the `MEDMEM::MESH` query methods :

1. general information are read (mesh dimension, number of elements), and we check that the mesh contains nothing but tetrahedra;
2. node coordinates are retrieved;
3. tetrahedra and triangle nodal connectivities are retrieved;
4. edge nodal connectivity is reconstructed;
5. RHEOLEF elements are built from these informations;
6. MEDMEM families are browsed to create corresponding RHEOLEF domains.

2.2.3 Creating MED-memory result fields

After computation, result fields are available from RHEOLEF. Let us now see how to transfer this data in a MED structure. The `memfield_from_rheoleffield` class takes care of this aspect.

Here follows the code of the method that copies the content of a RHEOLEF field in a new MED-memory `double` field structure :

```
void memfield_from_rheoleffield::addNewField(double t)
{
  MEDMEM::FIELD<double>* new_field = new MEDMEM::FIELD<double>(&m_support,1);
  int nb_elements = m_support.getNumberOfElements(MED_EN::MED_ALL_ELEMENTS);

  string Units("K");
  string Description("Temperature");
  string ComponentsNames("T");
  new_field->setMEDComponentsUnits(&Units);
  new_field->setComponentsDescriptions(&Description);
  new_field->setComponentsNames(&ComponentsNames);
  new_field->setName(m_name);
  new_field->setIterationNumber(++m_iteration_number);
  new_field->setTime(t);
  new_field->setOrderNumber(0);

 for (int j=1; j<=nb_elements; j++)
```

```
    new_field->setValueIJ(j,1,m_rheolef_field->at(j-1));

  SCRUTE(m_name);
  m_field_collection.push_back(new_field);

}
```

Please note :
– The use of the class template `MEDMEM::FIELD<T>`
– The constructor of this template which takes as argument a `support` (the description of a region of the mesh, see document [DR1]) and the number of components per support element. Here, our support has been defined on nodes and we consider the representation of a scalar value so our second argument is 1.
– The description of the field with name, description, units.
– The setting of an iteration number (iteration number).
– The setting of a sub-loop iteration number (order number, always set to zero in our case).
– The copying of the data through the `setValueIJ` inline method. The call takes into account the fact that MED numbering starts at 1 instead of 0 for RHEOLEF.

2.3 Compiling the implementation of the interface

In order to prepare the creation of the SALOME component, it is best to work in a predefined frame that is given by the SA_build_new_cpp_component script. This script, the use of which is fully described in [DR2], creates a directory structure, configuration files and outlines for the C++ and SWIG headers. After using the script, the developer should :
– put the C++ implementation of the interface in the `src/SRHEOLEF/SRHEOLEF_CXX` directory
– implement a C++ test file in `src/SRHEOLEF/SRHEOLEF_CXX/main.cxx`
– put the SWIG Python wrapper `SRHEOLEF.i` file in `src/SRHEOLEF/SRHEOLEF_SWIG`
– implement a Python test in `src/SRHEOLEF/SRHEOLEF_TEST` directory
– modify the `Makefile.am` files to reflect the changes.

2.4 Testing the standalone component

In our case, the standalone component is tested via a C++ program written in main.cxx and a Python program written in `SRHEOLEF_test.py`. Both perform the following actions thanks to the SRHEOLEF interface :
– read a mesh from a MED file
– set Dirichlet boundary conditions on regions “bottom” and “boundary”
– init heat equation problem
– perform 100 time steps of the heat equation
– write results in a MED file
– exit after cleanup
Here comes the code for the python test program :

```
from os import getenv

#loading SWIG modules
if getenv("SALOMEPATH"):
    import salome
    import SRHEOLEF_ORB
    my_SRHEOLEF = salome.lcc.FindOrLoadComponent("FactoryServer", "SRHEOLEF")

    IN_SALOME_GUI = 1
else:
    from SRHEOLEFSWIG import *
    my_SRHEOLEF = SRHEOLEF()
pass

#
#
print "Test Program of SRHEOLEF component"

# twocubes.med consists of a tetrahedric mesh
# of a small box on top of a large box
# ‘‘bottom’’ group is the group of faces corresponding
#   to the bottom of the big box
# ‘‘boundary’’ group is the group of all the other external
# faces

MedFile="/home/fsalome4/RHEOLEF/rheolef_demo_salome_vb/MED_files/twocubes.med"

#’’Mesh_1’’ is the default name given by SMESH to
# a newly created mesh

MeshName="Mesh_1"

my_SRHEOLEF.setMeshFromFile(MedFile,MeshName)

#setting temperature boundary conditions on groups defined
#in the mesh

my_SRHEOLEF.setDirichlet(’bottom’,1000.0)
my_SRHEOLEF.setDirichlet(’boundary’,100.0)

#computation of the heat problem on the mesh
my_SRHEOLEF.heatEquationInit()

dt = 1.0
while i in range(10):
    myCalc.heatEquationTimeStep(dt)
```

```
#stores the mesh and the resulting fields in a file
my_SRHEOLEF.dump("/export/home/result_twocubes_py.med")
```

The first few lines are of particular interest. In the case of an existing “SALOMEPATH” variable, the script is running in the SALOME framework, and the SRHEOLEF component is loaded. The wrapping of SRHEOLEF methods is done by CORBA. This will be available after the HXX2SALOME tool has been used as explained in the next section. In the second case, we use the standalone SRHEOLEF interface, and Python methods are made available by SWIG wrappings.

2.5 Creation of a standalone component - step by step

1. unzip `tutorial.tgz` and set the absolute path of the `tutorial` directory as the `$TUTORIAL_ROOT_DIR` environment variable
2. set `$HXX2SALOME_ROOT_DIR` in your `$PATH` environment variable
3. set the install path for RHEOLEF as `$RHEOLEF_INSTALL_DIR`
4. run `SA_new_cpp_component SRHEOLEF` to create the skeleton corresponding to the C++ interface of the RHEOLEF code
5. copy `SRHEOLEF_SRC` from the tutorial.tgz to fill the skeleton created by the previous step
6. run `build_configure` in the `SRHEOLEF_SRC` directory
7. create a `SRHEOLEF_BUILD` directory at the same level as `SRHEOLEF_SRC`
8. `cd SRHEOLEF_BUILD`
9. `../SRHEOLEF_SRC/configure -prefix=‘pwd‘/../SRHEOLEF_INSTALL`
10. `make all install` to install the library and test programs
11. `cd ../SRHEOLEF_INSTALL/bin/salome`
12. run `./SRHEOLEF_test` to run the C++ test program
13. add `../SRHEOLEF_INSTALL/lib/salome` to `$PYTHONPATH`
14. run `python SRHEOLEF_test.py` to run the Python script using SWIG wrapping of the C++ interface

3 Integration of the new component in SALOME

3.1 Creation of a SALOME component

Once the standalone interface has been tested, the SALOME component can be created. This step is performed by the HXX2SALOME tool, which takes as an input the SRHEOLEF.hxx header and the libSRHEOLEFCXX.so object file created by the compilation of our interface. The whole procedure is detailed in document [DR2].

As pointed out in document [DR2], the header has to be compatible with CORBA interfacing, which puts some constraints on the signatures of the methods. In particular, no method overloading is allowed, and constness of

types is of special significance, as it will correspond to CORBA argument being declared in or out. Only some types (basic types, MEDMEM meshes and fields, std::vector) are compatible with the tool.
The command writes :

```
hxx2salome -c -e /home/fsalome4/salome_install/env_products_3_2_0.sh
/home/fsalome4/SRHEOLEF/SRHEOLEFCPP_SRC/SRHEOLEF_INSTALL
SRHEOLEF.hxx libSRHEOLEFCXX.so
/home/fsalome4/RHEOLEF/rheolef_demo_salome_vb/SRHEOLEFCPP_SRC
```

SALOME can now be run with the new component :

```
source /home/fsalome4/salome_install/env_products.sh
runSalome --modules=GEOM,SMESH,VISU,SRHEOLEF,SUPERV
```

3.2 Life cycle of objects created by HXX2SALOME

Whereas in the standalone component, the interface is entirely responsible for the memory management of the objects it manipulates, in the SALOME framework, the developer must pay a particular attention to the ownership of the MEDMEM pointers. The MEDMEM objects that are passed between different methods are transformed by HXX2SALOME in CORBA clients. HXX2SALOME chooses to give the ownership of the underlying C++ object to the CORBA client or not according to the way the object was passed in the header.
For instance, let us consider the coupling of the temperature diffusion and the displacement computation in our SRHEOLEF.hxx interface:

```
  MEDMEM::FIELD<double>& computeDirichlet();

  void computeThermoElasticity(const MEDMEM::FIELD<double>* temperature);
```

By choosing to return a reference to a `MEDMEM::FIELD`, we specify that the `SRHEOLEF` servant keeps the control over the `MEDMEM::FIELD` object that is created. Had we chosen to return a pointer, `hxx2salome` would relinquish the ownership to the CORBA client, the field would have been destroyed with the CORBA client (i.e. after the call to `computeThermoElasticity`).

3.3 Creation of a Salome component - step by step

1. run `hxx2salome` to create the SRHEOLEF component for SALOME
2. source the environment file passed as an argument to `hxx2salome`
3. set `SALOME_trace` environment variable to `local`
4. run Salome with all the desired modules (`runSalome -modules=GEOM,SMESH,VISU,SRHEOLEF,SUPERV`)

4 Use via an interactive SALOME session

4.1 Outline

The SRHEOLEF component being available, the user can :
- create a 3D CAD geometry with GEOM component;
- define CAD face groups that will eventually become supports for boundary conditions;
- mesh the geometry with the SMESH component;
- construct mesh groups from the CAD groups;
- create a MED-file with the geometry;
- create a supervisor dataflow that performs a computation chain corresponding to a transitory heat equation problem;
- perform the computation;
- visualize results with the VISU component.

4.2 CAD and meshing

4.2.1 Outline

Those functionalities are described in the online help [DR4], which presents in depth the GUI and TUI for the CAD tool and the Meshing tool. In Salome, the CAD component is called GEOM, while the Meshing component is called SMESH.

The resulting mesh can be exported to a MED-file and be used as an input for the SRHEOLEF script.

As an example, we propose here to create a mesh that represents a hot blade that is immersed in a cold fluid. On this mesh, we will use the SRHEOLEF component to perform a heat propagation computation followed by an elastic displacement with elastic coefficients depending on the temperature. On our model, we will therefore define three faces for setting boundary conditions :
- an internal one where we will set a fixed hot temperature boundary condition for the heat transport problem and on which we set zero displacement for the elasticity problem,
- the face that is plunged in the fluid where we will set fixed cold temperature boundary condition for the transport,
- the other faces where we will set zero flux boundary condition for the heat equation.

The following subsections take the user step by step in the process of building a CAD model and a corresponding mesh. The capabilities of Salome CAD module are considerably richer than the basic features that are used in this tutorial.

4.2.2 CAD modelling - step by step

1. Run Salome.
2. Switch to the GEOM module by clicking on the appropriate icon.
3. Create four vertices that will be used to set the position of the shapes. To do so, choose the vertex icon,

or use **New Entity** > **Basic** > **Point** to open the dialog box. Create vertices at locations specified by the following table. Use 'Apply' button to avoid closing the dialog box after each vertex definition.

Vertex name	X	Y	Z
Vertex_1	100	-50	0
Vertex_2	100	50	0
Vertex_3	120	50	0
Vertex_4	120	-50	0
Vertex_5	100	-20	3.3333
Vertex_6	100	-15	5
Vertex_7	100	0	10
Vertex_8	100	30	0
Vertex_9	100	25	-10
Vertex_10	100	0	0
Vertex_11	100	-10	0
Vertex_12	100	-20	0
Vertex_13	100	-30	0

4. Create a quadrangle that will represent the section of the rotor. To do so, use **New Entity** > **Blocks** > **Quadrangle Face** and specify vertices 1,2,3 and 4.
5. Create a vector along direction y for the direction of the revolution of the rotor section. To do so, choose the vector icon or use **New Entity** > **Basic** > **Vector** to select the DXDYDZ mode and choose (0,100,0).
6. Create the rotor by a revolution of the section. To do so, choose **New Entity** > **Generation** > **Revolution** and specify **Quadrangle_Face_1** as the base shape. Use **Vector_1** as a direction and 30 degrees as an angle. At this stage, you should have a view similar to figure 1.
7. Rotate the obtained shape so that the shape is centered on the x axis. To do so, use **Operations** > **Transformation** > **Rotation** and select an angle of -15 degrees applied on shape **Revolution_1**.
8. Create a vector that defines the direction of the blade. To do so, choose the vector icon or use **New Entity** > **Basic** > **Vector** to select DXDYDZ mode and choose (100,0,0).
9. Create a curve that will define the shape of the section of the blade. To do so, choose the curve icon or use **New Entity** > **Basic** > **Curve**. Select vertices 5,6,7,8,9,10,11,12 on the Object Browser.
10. To close the curve, define two edges with **New Entity** > **Basic** > **Edge** linking vertices (5,13) and (12,13).
11. We now have to define a face based on the curve and the edges. GEOM distinguishes **edges** that are arcs or segments from **wires** that are collections of edges (a wire can be closed or open). To create a face, we need to create a closed wire from the curve and the two edges. To do so, select **New Entity** > **Build** > **Wire** and select the curve and the two edges.
12. We will now rotate the wire to give it the right orientation. To do so, use **Operations** > **Transformation** > **Rotation** and select an angle of -45 degrees applied on shape **Wire_1**, **Vector_2** being the rotation direction.

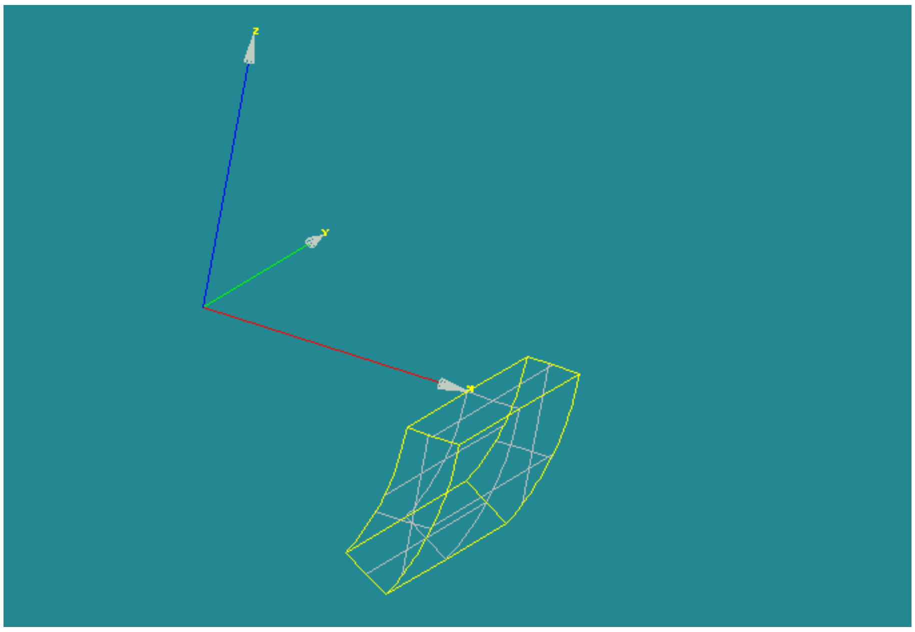


FIG. 1 – creation of the rotor

13. We can now create a face from the wire. To do so, select **New Entity** > **Build** > **Face** and select the wire.
14. We can now create the blade by extrusion of the face. To do so, select **New Entity** > **Generation** > **Extrusion**. Select the face **Face_1** as a shape and **Vector_2** as a direction. Use 100 as a length for extrusion.
15. Let us now fuse the two shapes (blade and rotor). To do so, select **Operations** > **Boolean** > **Fuse** and fuse the rotor **Revolution_1** and the blade **Prism_1**.
16. Right click the **Fuse_1** shape in the Object Browser window and select 'Display Only'. Right click the wireframe view in the OCC window and select **Display Mode** > **Shading**. You should obtain a view similar to figure 2.

17. Our volume CAD model is now ready. Let us now turn to the setting of face groups that will be the support of our boundary conditions. This is achieved by the **New Entity** > **Group** > **Create** command that can be used to select groups of vertices, edges, faces or volumes. Here, we select **Face** in the window, choose `fluid` in the group name, **Fuse_1** as the main shape. At this stage, the faces can be selected on the OCC view by clicking the shapes. After each selection, the face is added to the group by using the **Add** button. For `fluid`, select five faces, the face at the end of the blade, thethree faces making up the blade and the plate supporting the blade.
18. Repeat the previous operation with group `internal` which is made of one face, the inner face at $r = 100$.
19. Repeat the previous operation with group `vessel` which is made of the four remaining external faces.

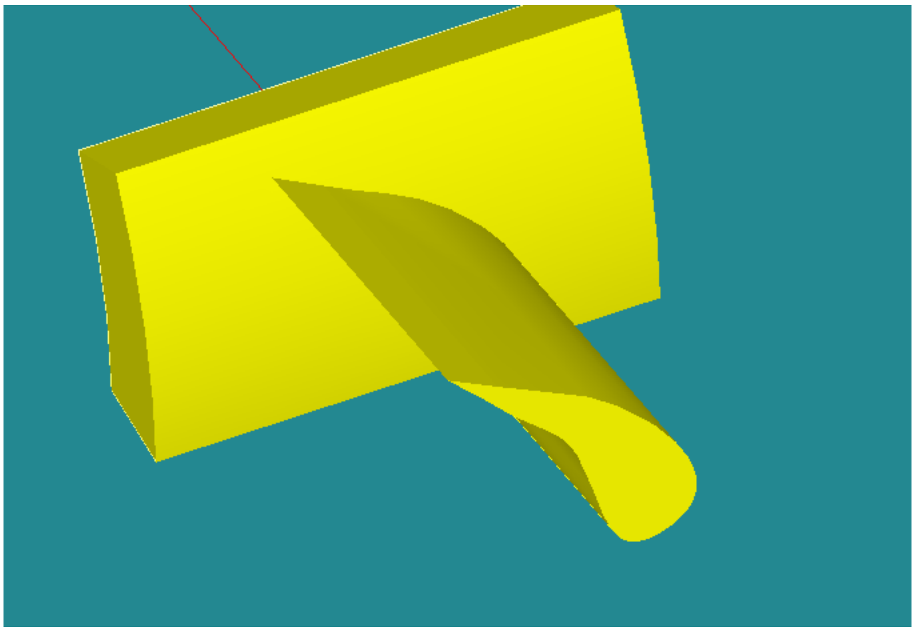


FIG. 2 – creation of the object

20. The CAD model is ready, and we can now turn to the mesh. Before doing so, we will use a powerful tool provided by Salome : the possibility to convert all the operations performed in the GUI mode to a Python script. Use **File > Dump study** to create a Python script that reproduces all the steps performed in the GUI mode. Thanks to this script, it is possible to rebuild the object with the possibility of changing the location of the different objects in TUI mode.

4.2.3 Meshing - step by step

Meshing is a straightforward operation. The user is asked to specify algorithms and hypotheses for meshing wires, faces and volumes.

1. Switch to the SMESH mode. The display is automatically switched to the VTK viewer, which is more appropriate for the Meshing phase.
2. Create a mesh with **Mesh > Create mesh**. A dialog box opens with three different tabs corresponding to 1D, 2D and 3D algorithms. Select **Fuse_1** as the geometry on which the mesh is based. Select **Wire Discretization** with **Average length** equals to 5 for 1D. Select **Triangle (MEFISTO)** with **Length from Edges** for 2D. Select **Tetrahedra (Netgen)** with **Max Element Volume** equals to 10000 for 3D.
3. Compute the mesh by right-clicking the mesh in the Object Browser and selecting **Compute**. You should obtain a view similar to figure 3.

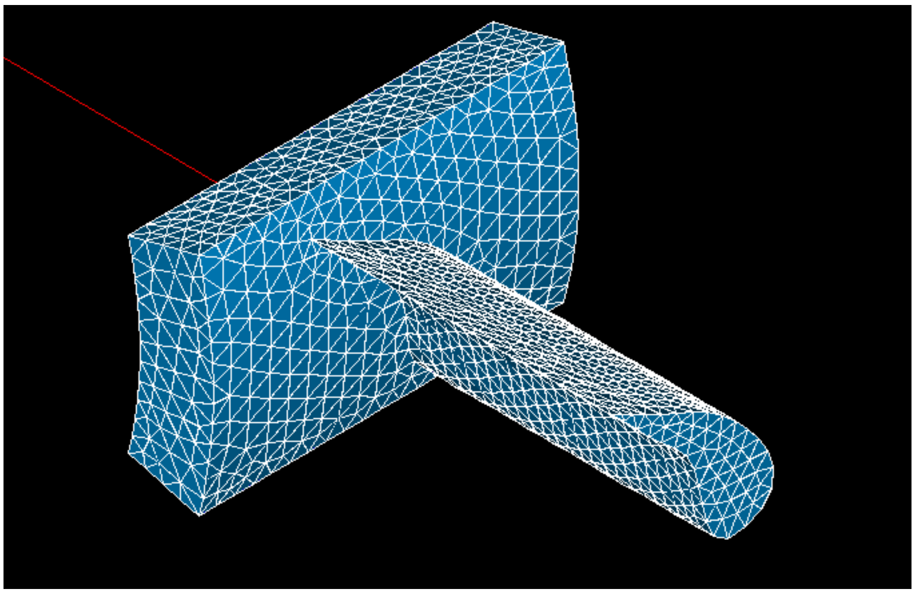


FIG. 3 – Meshing the object

4. We must now create mesh groups that correspond to the CAD groups created in the CAD phase. To do so, select **Mesh > Create Group**. A dialog box opens where you must specify a mesh on which the mesh group is based, a name for the mesh group, the components of the group (node, edge, face, volumes) and the selection mode. Choose a group called 'fluid', and select 'Group based on geometry', selecting geometry 'fluid'. Right clicking on mesh group 'fluid' in the Object Browser,you should now see a figure like the one given on figure 4.

5. Repeat the operation for other face groups 'vessel' and 'internal'.
6. Create a MED-file containing the mesh and the mesh groups by right-clicking the mesh and choosing 'Export to MED file'.
7. It is possible to dump the GUI operations in a Python script as was done for the GEOM module.

4.3 Supervision

4.3.1 Outline

In Salome, the supervision component is called SUPERV. It enables the user to dispatch different computational steps which form a “dataflow”. The platform then determines independent nodes in the flow graph, and can instantiate computational components on different machines, so that concurrent computation is performed.

The supervision also offers graphical display of the progress of a computation chain. A complete description of the SUPERV tool can be found in [DR3].

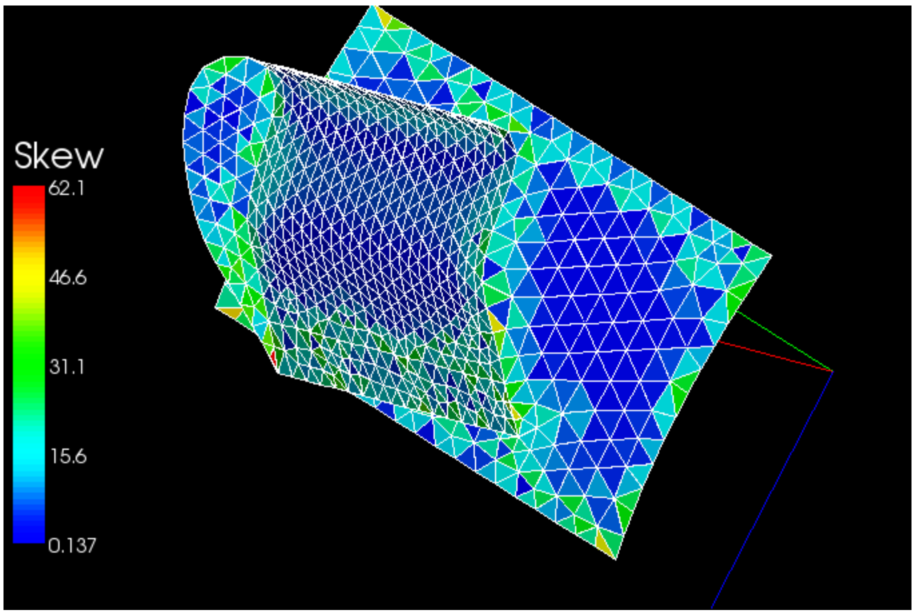


FIG. 4 – Defining the fluid surface

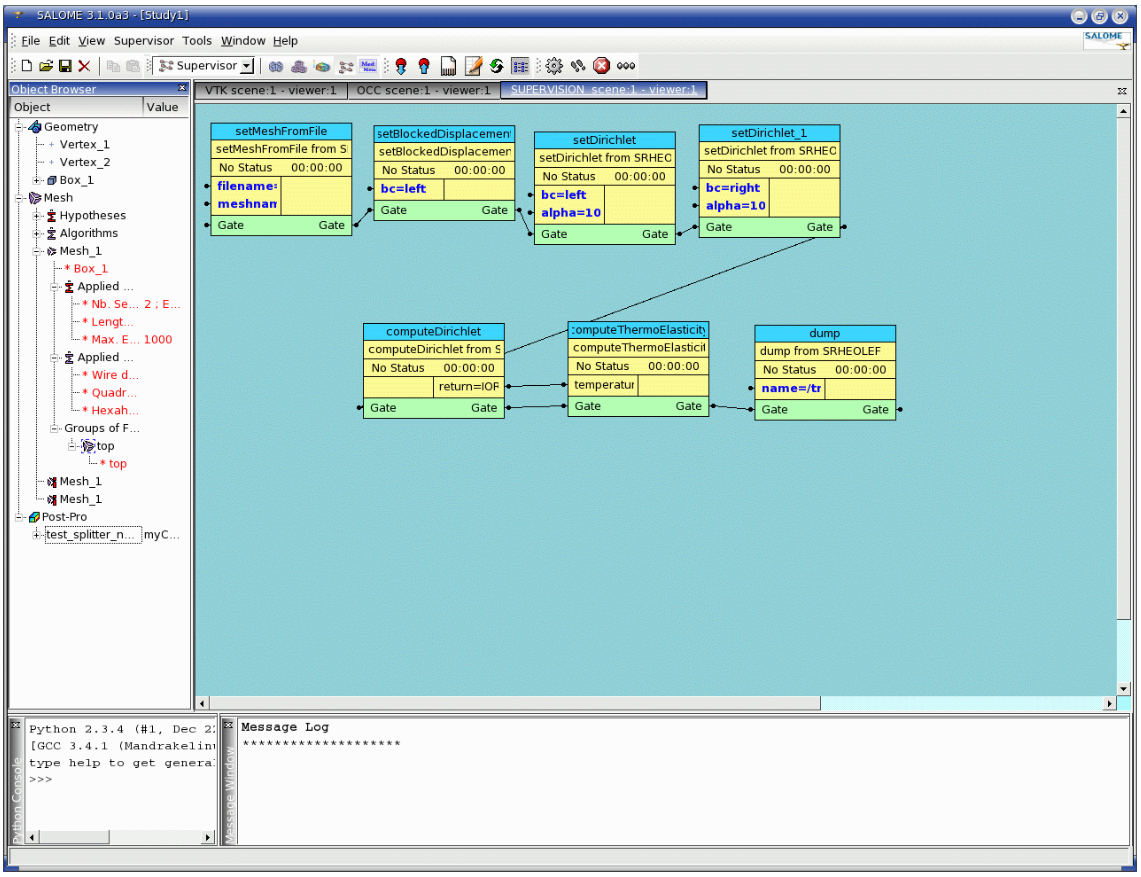
The different methods that have been implemented in the SRHEOLEF interface are now available as nodes in the computation graph. For each node in the graph, input and output data are determined, so that each input has either to be linked to the output of another node or to be set by the user.

It is possible to create special nodes which implement loop, while, switch statements. In our TUI example, we use loop nodes so that the heat equation time step is performed ten times (see section 5).

4.3.2 Using supervision module - step by step

This example illustrates the possibility to couple different codes via the MED-memory exchange format. Here, the temperature field is exchanged between two different computational nodes.

1. Let us first add a computation node to the graph. To do so, right-click on the empty view and select **Add node**. A dialog box opens, on which the node can be selected in a variety of modules. Select the **SRHEOLEF > setMeshFromFile** node which was created by our interface in section 3. A node appears with a name, an execution status (undefined), three input ports (on the left) and one output port (on the right). The three input ports correspond to the two input arguments specified in the interface and a synchronisation gate. The output port is a synchronisation gate.
2. To create our computation chain, let us add other ports :
   - two **setDirichlet** nodes
   - one **setFixedDisplacement** node
   - one **computeDirichlet** node
   - one **computeThermoElasticity** node



```
Python 2.3.4 (#1, Dec 2
[GCC 3.4.1 (Mandrakelin
type help to get genera
>>>
```

```
Message Log
********************
```

FIG. 5 – Defining the SUPERV dataflow

– one **dump** node.

3. Let us now link the different nodes. Using synchronization gates, draw links between nodes so that they are placed in the order of the previous list. To do so, right click on the gate and use **Sketch Link**.
4. The two computation nodes must also be linked via the ouput/input nodes. The MEDMEM field that is the output of **computeDirichlet** must be the input of **computeThermoElasticity**.
5. The remaining input nodes are strings or values that enable the user to specify the filenames and the value of the boundary conditions. The strings specified in setMeshFromFile should fit the file name and the mesh name given during the export of MED file at the meshing step. Boundary conditions strings should fit the names given to the face mesh groups of the meshing step.
6. You should now obtain a window resembling that of figure 5.

7. It is now possible to run the computation graph by clicking the **Run Dataflow** icon. The MED-file containing the resulting field is then available at the specified location.

4.4 Visualization

The fields which are written by the `dump(filename)` method of our test script can be visualized with the VISU component. It can be used to create quickly 3D views of the domain, movies with all available frames. Different representations (cut planes, scalarmap, iso contours) are supported. More information can be found in [DR3], but the use of this tool is rather straightforward and the user can create presentations of the results quite easily.

4.4.1 Visualizing - step by step

We propose to illustrate a few features of the VISU module.

1. Switch to the VISU module.
2. Import the MED-file that was created by the previous computation. To do so, select **File > Import > MED File**. The structure of the MED file appears in the object browsers.
3. Select one of the resulting field. Right-click on one of the time steps of the field and select **Scalar Map**. You should now obtain a window resembling that of figure 6.

4. To see inside the structure, right-click on the scalar map in the Object Browser and choose **Clipping Planes**. It is then possible to define a plane which defines a half-space in which elements are visible.
5. Import the MED-file called $TUTORIAL_ROOT_DIR/resources/heat_transport_result.med. It contains a transitory heat diffusion problem. On this one, it is possible to create a movie displaying the evolution of the temperature.
6. Right-click on the (T,K) field and select **Animation**. Setup the animation choosing cut planes, generate frames and run the movie.

5 Use via Python scripts

The Python scripts can reproduce the actions performed in the GUI mode. The user can thus parametrize the domain mesh generation and can easily modify an unsatisfying study. The scripts described in this section are available from the `tutorial.tgz` archive.

5.1 CAD and meshing

5.1.1 Creation of a Python script

The CAD and meshing steps are presented in file `make_two_cubes.py` which creates a MED-file containing geometries and groups.

In this example, the CAD domain consists of two superposed boxes. CAD Face groups are created on the external faces of the shape, the bottom one being labeled ’bottom’, while all the other ones are labeled ’boundary’.

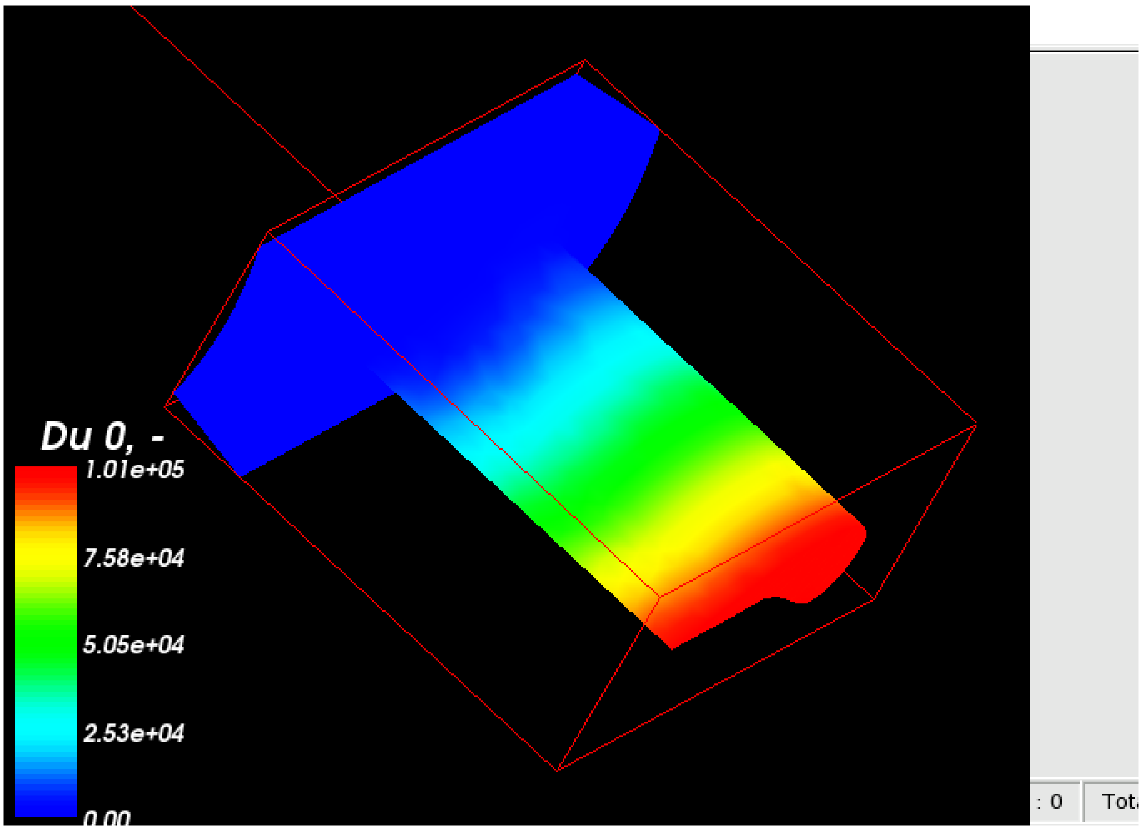


FIG. 6 – View of the modulus of the displacement field for the blade

The meshing is performed by choosing 1D, 2D and 3D meshing algorithm. We select meshing meethods that create triangles in 2D and tetrahedra in 3D, and we choose the parameters so that we specify the typical length of the edges.

Eventually, the mesh is exported to a MED file, for reuse by other SALOME components. This MED-file is compatible with execution of the stationary temperature computation of our SRHEOLEF component.

5.1.2 Loading a previously dumped Python script

As it was said earlier, it is possible to dump in a Python script all the operations which are performed in the GUI. As an example, dumping the CAD operations described in section 4.2.2 will result in the creation of a file postfixed by GEOM. Such a file is available in the scripts directory of our tutorial archive. It can be loaded using the **File** > **Load Script** menu. After that, typing **RebuildData(salome.myStudy)** will "play" the operations in the current study.

5.2 Supervision

Two dataflow examples are given with the tutorial. The first one, `superv_heat_equation.py` corresponds to a transitory heat equation. The solution is advanced in time for ten iterations. This supervision dataflow can be generated from the `superv_heat_equation.py` script. This dataflow reproduces the different steps of the heat equation example.

The second one corresponds to a coupling case : a stationary heat equation problem is first solved, then an elasticity problem with coefficients dependent on temperature is solved. This illustrates the capability of SALOME to integrate multiple physics codes in the same platform, via MED-files or via MED-memory structures. In our case, the two computation nodes `computeTemperature` and `computeThermoElasticity` communicate via the MEDMEM::FIELD structure containing the temperature.

6 GUI creation

6.1 Outline

It is possible to exploit the graphical capabilities of SALOME and create a GUI for the newly built component. For doing so, `hxx2salome` offers an option that creates a GUI template for the component it creates.

6.2 Template created by hxx2salome

In order to create a GUI template, it is sufficient to add option `-g` to the `hxx2salome` command line that was given in subsection 3.1 Thus, in addition to the Corba wrappings generated by `hxx2salome`, a new directory containing a basic GUI is generated.
The command writes :

```
hxx2salome -g -c -e /home/fsalome4/salome_install/env_products_3_2_0.sh
```

```
/home/fsalome4/SRHEOLEF/SRHEOLEFCPP_SRC/SRHEOLEF_INSTALL
SRHEOLEF.hxx libSRHEOLEFCXX.so
/home/fsalome4/RHEOLEF/rheolef_demo_salome_vb/SRHEOLEFCPP_SRC
```

This command generates different types of files :

SRHEOLEFGUI.cxx the file that implements the GUI

SRHEOLEFGUI.h the corresponding header

SRHEOLEF_icons.po a mapping of strings used in SRHEOLEFGUI.cxx to PNG files containing icons

SRHEOLEF_msg_en.po a mapping of strings used in SRHEOLEFGUI.cxx to messages in English

SRHEOLEF_msg_en.po a mapping of strings used in SRHEOLEFGUI.cxx to messages in French

This template generates a GUI that :
– gives the possibility to switch to SRHEOLEF mode,
– implements a submenu in the File menu,
– implement a new menu named SRHEOLEF,
– implements an action icon on the toolbar.

Unfortunately, `hxx2salome` does not declare the new component to SALOME, so you have to do it by yourself. To do so, edit `$GUI_ROOT_DIR/share/salome/resources/SalomeApp.xml` and add the SRHEOLEF component root dir at the required places.

6.3 Implementing the GUI

The possibilities of the GUI are vast, ranging from simple dialog boxes to graphical selection of elements on a mesh display. The web page [DR7] proposes an in-depth tutorial to create a GUI. It describes step by step the creation of a data model for the GUI component, the browsing of this model with the Object Browser window and the implementation of XML models that enable the user to load/save data models.

A very basic example is given in the present tutorial. The implementation that can be found in SRHEOLEFGUI :
– implements two items in the File menu to import and export files
– implements a new menu with two items that enable the user to set boundary conditions and compute a stationary heat equation problem,
– creates an icon that runs the stationary heat equation problem.

6.4 GUI creation - step by step

1. run `hxx2salome` with the `-g` option to create a GUI template,
2. copy SRHEOLEFGUI from the tutorial archive in the `./SRHEOLEF_CPP/src/` directory
3. cd `SRHEOLEF_BUILD`
4. make all install
5. add SRHEOLEF to `SalomeApp.xml`
6. runSalome –modules=GEOM,SMESH,MED,VISU,SUPERV,SRHEOLEF

7 Prospects

This tutorial is intended as a companion to document [DR2] that describes the integration of an external code with `hxx2salome`. In the future, this tutorial could be upgraded to include :
– the use of XDATA as a datafile manager with the `xdata2cpp` tool,
– a more elaborate GUI with Object Browsing and significant checks.

A geo_read_from_med_mem.cxx

```
//================================================================================
// File      : geo_read_from_med_mem
// Project   : SALOME
// Author    : V. Bergeaud, CEA/DEN/DM2S/SFME/LGLS
//==============================================================================
//
// A class that derives from RHEOLEF::georep and
// uses a MEDMEM sructure as an input argument of
// the constructor
//
// Only tetrahedric meshes are accepted
//

# include "geo_read_from_med_mem.hxx"
# include "domain_read_from_med.hxx"
# ifdef __cplusplus

extern "C"
{
# endif
# include <med.h>
# include <stdio.h>
# ifdef __cplusplus
}
# endif

# include <iostream>
# include <iomanip>
# include <fstream>
# include <string>
# include <typeinfo>
#include <string>

# include <vector>
#include <set>
#include "MEDMEM_Exception.hxx"
# include "MEDMEM_CellModel.hxx"
# include "MEDMEM_Mesh.hxx"
# include "MEDMEM_Group.hxx"

#include "/export/home/vb/rheolef-5.18/nfem/lib/geo-connectivity.h"
# include "utilities.h"


// inspired from rheolef load_subgeo_numbering
// where connectivity is given by the conmed
// array

template<class RandomIterator, class SetRandomIterator,
 class RandomIterator2, class RandomIterator3,
 class Size1, class Size2>
void
load_subgeo_numbering2 (
// input
RandomIterator             element,
SetRandomIterator          ball,
Size1                      subgeo_dim,
Size2                      n_subgeo,
const int**                conmed,
const int*                 face_offset,
// in-output
RandomIterator2            count_geo,
RandomIterator3            count_element)
{
  typedef typename iterator_traits<RandomIterator>::value_type    element_type;
  typedef reference_element::size_type                        size_type;
  typedef set<size_type>                                 set_type;
  typedef set<size_type>::const_iterator                 const_iterator_set;

  SCRUTE(subgeo_dim);
  SCRUTE(n_subgeo);
  for (size_type idx = 0; idx < (unsigned)n_subgeo; idx++)
    {
      element_type S;
      string str_buf;
      ostringstream stro;

      //construction of the string that would have been
      //read in a geo file
      int tmp[3];


      //dimension 2 : using connectivityIndex
      if (subgeo_dim==2)
{
  for (int l= face_offset[idx]-1; l < face_offset[idx+1]-1; l++)
    tmp [l-face_offset[idx]+1]=conmed[2][l]-1;
  stro<<tmp[0]<<" "<<tmp[1]<<" "<<tmp[2]<<" ";
}
      else
{
  for (int l=0; l<=1;l++)
```

```
    tmp[l]=conmed[1][idx*2+l] -1;
  stro<<tmp[0]<<" "<<tmp[1]<<" ";
}

      stro<<flush;
      str_buf = stro.str();
      istringstream strinput(str_buf);

      //putting the connectivity
      //in the S element
      strinput >> S;
      count_geo     [S.dimension()] ++;
      count_element [S.type()] ++;

      set_type contains_S;
      build_set_that_contains_S (S, ball, element, contains_S);

      for (const_iterator_set p = contains_S.begin();
   p != contains_S.end(); p++)
{
  element [*p].set_subgeo(S, idx);
        }
    }
}


const int MED_SUCCESS = 0 ;
const int num_version = 2 ;
const int num_version2 = 1 ;

/*! constructs a rheolef::georep object from a MEDMEM::MESH

The method is implemented only for meshes made out of
tetrahedra, with regions of tetrahedra or triangles
*/

geo_from_medmem::geo_from_medmem(const MEDMEM::MESH& myMesh ) :georep()
{
  BEGIN_OF("geo_read_from_med_mem ") ;

  //***********************************************************
  // Step 1 : reading general information
  //***********************************************************

  //RHEOLEF data format version
  _version=2;
```

```
//MEDMEM call to get the name of the mesh
_name=myMesh.getName();
SCRUTE(_name);

//Reading the mesh dimension
//and checks that it is 3
int gdim = myMesh.getMeshDimension();
ASSERT( gdim == 3);
_dim=gdim;
SCRUTE(_dim);


//integer array that contains the number of nodes, edges, faces and cells
int num_elem[4];

// definition of the Entities and Geometric elements
// that are supported in this version

MED_EN::medEntityMesh typent[4] = {MED_EN::MED_NODE,
   MED_EN::MED_EDGE,
   MED_EN::MED_FACE,
   MED_EN::MED_CELL};
MED_EN::medGeometryElement typgeo[4] = {MED_EN::MED_POINT1,
MED_EN::MED_SEG2,
MED_EN::MED_TRIA3,
MED_EN::MED_TETRA4};

//Reading the number of nodes
int k;
num_elem[0]=myMesh.getNumberOfNodes() ;
SCRUTE(num_elem[0]);

//Reading the other number of elements
for (k=2; k<4; k++)
  {
    SCRUTE(typent[k]);
    SCRUTE(typgeo[k]);
    num_elem[k] = myMesh.getNumberOfElements(typent[k],typgeo[k]);
    SCRUTE(num_elem[k]);
  }

long n_vert=num_elem[0];
int n_elt=num_elem[3];
int n_fac=num_elem[2];
int n_edg=num_elem[1];
```

```
  //*************************************************************
  // Step 2 : reading node coordinates
  //*************************************************************


  //georep derives from std::vector<cell>
  //we resize it so that we can fit the tetrahedra in.
  resize(n_elt);

  //the vertices are stored in the _x vector
  _x.resize(n_vert);

  //Coordinate table allocation
  //MED_FULL_INTERLACE is used because coordinates are
  // stored in x1, y1, z1, x2, y2, ... order
  const double* coomed = myMesh.getCoordinates(MED_EN::MED_FULL_INTERLACE);

  // copying the coordinates in the rheolef structure
  // and computation of _xmax, _xmin
  for (k=0; (unsigned)k< _dim ; k++)
    {
      _xmin[k] = numeric_limits<Float>::max();
      _xmax[k] = -numeric_limits<Float>::max();
    }
  iterator_node iter = begin_node();
  for ( k=0 ; k<n_vert ; k++ )
    {
      float  *vertice ;
      point& p=*iter++;
      for ( int j=0 ; j<gdim ; j++ )
{
  p[j]=coomed[k*gdim+j] ;
  _xmin[j] = min(p[j],_xmin[j]);
  _xmax[j] = max(p[j],_xmax[j]);
}
    }


  //*************************************************************
  // Step 3 : reading connectivities
  //*************************************************************


  // Reading connectivities in MEDMEM

  //conmed[k] points to  the nodal connectivities for element of dimension k
  // conmed[0] points to nothing
  // conmed[1] points to edge connectivity
  // conmed[2] points to triangle connectivity
  // conmed[3] points to tetrahedra connectivity

  const int*  conmed[4] ;

  //MEDMEM gives access to nodal connectivities
  //for objects of dimension gdim and gdim-1 only !!
  //
  for (k = gdim-1; k<= gdim ; k++)// loop over dimensions for nodal connectivity
    {
      SCRUTE(k);
      try
{
  conmed[k] = myMesh.getConnectivity(MED_EN::MED_FULL_INTERLACE,
   MED_EN::MED_NODAL,
     typent[k],typgeo[k]);
}
      catch (MEDMEM::MEDEXCEPTION m) {
cout << m.what() << endl ;
throw;
      }
    }

  //getting the face connectivity offset
  //the nodes of triangle k
  //are between conmed[2][face_offset[k-1]] and conmed[2][face_offset[k]]
  const int* face_offset =  myMesh.getConnectivityIndex(MED_EN::MED_NODAL,
MED_EN::MED_FACE);

  // setting some rheolef internal counters
  size_type idx = 0;
  reset_counters();
  _count_geo [0] = n_node();
  _count_element [0] = n_node();
  geo::iterator last_elt = end();
  int j=0;
  const int m = 3;

  // loop over elements to fill rheolef connecivity

  for (geo::iterator i = begin();i<last_elt; i++,idx++)
    {
      (*i).set_name('T');
      int tmp[m+1];
      for (int l=0; l<=m; l++)
{
  tmp[l]=conmed[m][j*(m+1)+l] - 1 ;
}
      //reversing the tetrahedra
      (*i)[0]=tmp[0];
      (*i)[1]=tmp[1];
      (*i)[2]=tmp[3];
      (*i)[3]=tmp[2];
      j++;
      (*i).set_index (idx);
      _count_geo     [(*i).dimension()]++;
      _count_element [(*i).type()]++;

    }


  //**************************************************************
  // Step 4 : Reconstruction of the edge connectivity
  //**************************************************************


    //Reconstruction of edge connectivity
    //the conmed_index set stores couples of vertices (v1,v2)
    //under the n_vert*v1+v2 form with v1<v2 to avoid storing
    //twice the same pair

    std::set<long> conmed_index;
    for (geo::iterator i = begin(); i< last_elt; i++)
      {
for (int iv1=0; iv1<4; iv1++)
  for (int iv2=0; iv2<iv1; iv2++)
    {
      int v1=(*i)[iv1];
      int v2=(*i)[iv2];
      if (v1<v2)
conmed_index.insert((n_vert)*(v1)+(v2));
      else
conmed_index.insert((n_vert)*(v2)+(v1));
    }
      }
    n_edg=conmed_index.size();
    SCRUTE(n_edg);

    std::set<long>::iterator conmed_iter;
    int* conedge=new int[2*conmed_index.size()];
    int conmed_ind=0;
    for (conmed_iter=conmed_index.begin(); conmed_iter !=conmed_index.end(); ++conmed_iter)
      {
ldiv_t integer_division;
integer_division=div(*conmed_iter,n_vert);

// the connectivity is extracted from the v1*n_vert+v2
// storage
// 1 is added to the rheolef connectivity to fit MEDMEM
// indices
conedge[conmed_ind]=integer_division.quot + 1;
conmed_ind++;
conedge[conmed_ind]=integer_division.rem + 1;
conmed_ind++;
      }
    ///////////////////

      //conmed[1] is pointed to the connectivity vector
      const int* con(conedge);
      conmed[1]=con;

  //**********************************************************
  // Step 5 : filling RHEOLEF structures with infomation
  //**********************************************************

      //rheolef internal management of elements
      size_type map_d = map_dimension();
      vector<set<size_type> > ball (n_node());
      ball.resize(n_node());
      build_point_to_element_sets (begin(), end(), ball.begin());

      //put faces in rheolef
      MESSAGE("--------------TRIANGLES -------------------------");
      load_subgeo_numbering2 (begin(), ball.begin(), 2, n_fac, conmed, face_offset, _count_

      //put edges in rheolef
      MESSAGE("--------------EDGES-------------------------");

      load_subgeo_numbering2 (begin(), ball.begin(), 1, n_edg, conmed, face_offset,  _count


  //**********************************************************
  // Step 6 : reading groups to create RHEOLEF domains
  //**********************************************************


      // Reading MESH Groups
      // on faces and tetrahedra
      for (int dim=2; dim <=3; dim++)
{
  int type_entity = typent[dim];
  int type_elem = typgeo[dim];

  int nbfam=myMesh.getNumberOfGroups(type_entity);

  SCRUTE(nbfam);

  //request for the families of type type_entity
  vector<MEDMEM::GROUP *> mesh_regions=myMesh.getGroups(type_entity) ;

  for (k=0;  k < nbfam; k++)
    {
      SCRUTE(k);

      const string nomFam(mesh_regions[k]->getName());
      SCRUTE(nomFam);
      int nb=mesh_regions[k]->getNumberOfElements(type_elem);
      const int* myseq= mesh_regions[k]->getNumber(type_elem);

      //creation of a rheolef::domain from the MEDMEM group
      domain_read_from_med d(nomFam,nb,conmed,face_offset,myseq,type_elem);

      //propagation of numberings to the regions
      MESSAGE("ici");
      propagate_subgeo_numbering (d.begin(), d.end(), begin(), ball.begin());
      MESSAGE("la");

      //updating rheolef domains list
      _domlist.push_back(d);
    }
}

      //deallocating edge connectivity
      delete[] conedge;

      END_OF ("geo_read_from_med_mem ") ;
}
```