# Felisce Documentation Documentation

***Release 1.0.0***

**Jérémie Foulon David Froger**

**Mar 18, 2025**

# CONTENTS

Contents:

CHAPTER

ONE

# TUTORIAL: LAPLACE EQUATION

## 1.1 Introduction

This example presents one the most famous partial differential equation: heat equation.

$$\frac{\partial \mathrm{u}}{\partial \mathrm{t}} - \Delta u = 0 \quad \text{in } \Omega \tag{1.1}$$

$$\frac{\partial \mathrm{u}}{\partial \mathrm{n}} = 0 \quad \text{on } \mathrm{d}\Omega \tag{1.2}$$

$$u(x,0) = u_0(x) \quad \text{in } \Omega \tag{1.3}$$

- test configuration
- geometry
- mesh
- bondaay condition
- initial condition

## 1.2 Laplace

**Go to the `$FELISCE_ROOT/Tests/Example/Laplace`, you will find 5 files:**

- `main.cpp`: contain the main progam which is executed.
- `userHeat.hpp` and `userHeat.cpp`: where we specify how to build the linear system.
- `Makefile`: to build the programm.
- `data`: configuration file.

Consider the file `main.cpp`:

```
#include "Model/heatModel.hpp"
#include <userHeat.hpp>
using namespace felisce;

int main(const int argc, const char** argv)
{
  HeatModel model;
  model.initializeModel(argc, argv);
```

(continues on next page)

(continued from previous page)

```
  model.initializeLinearProblem( new UserHeat());

  //Loop time
  while (!model.hasFinished()){
    model.forward();
  }
}
```

```
#include "Model/heatModel.hpp"
#include <userHeat.hpp>
```

The first included header is the one of the model you will use to modelize the variationnal problem.

The second included header is for the definition of our particular problem, in this case for boundary conditions.

```
using namespace felisce;
```

To use the objects define of the felisce library without preffixing with the namespace `felisce::`.

```
HeatModel model;
```

Here the class Model is instanciated (so the default constructor is used).

```
model.initializeModel(argc, argv);
```

The model is initialized. The arguments `argc` and `argv` allow to pass command line options to overide default values. For example, the default configuration file name is `data`. To use a different file, use: "–file='databis'".

```
model.initializeLinearProblem( new UserHeat());
```

Here we speficy which to the model the linear system we are going to solve. The `UserHeat` class is explained just bellow. After this step we are ready to run the time loop:

```
while (!model.hasFinished()){
  model.forward();
}
```

Compile and run the program:

```
make
./main
```

The solution files will be written in the `Solution/` directory, that you can visualize with ensight:

```
ensight Solution/temperature.case
```

## 1.3 userHeat

Now that we have run the program, let's look at the `UserHeat` class defined in the `userHeat.hpp`:

```
#ifndef _USERHEAT_HPP
#define _USERHEAT_HPP
#include <Solver/linearProblemHeat.hpp>

namespace felisce
{
  class UserHeat:
  public LinearProblemHeat
  {
  public:
    UserHeat();
    ~UserHeat(){};
    void defineBC();
    void finalizeBCConstant();
  };
}

#endif
```

The `linearProblemHeat` class is reponsible to construct the system $A x = b$. Its methods relative to boundary condition are virtual and we have implemented them is `userHeat` class, which derive from `linearProblemHeat`.

```
   UserHeat();
   ~UserHeat(){};
   void defineBC();
   void finalizeBCConstant();


The two  first methods are the default constructor and destructor.
In ``defineBC`` the boundary conditions are enumerated, and their values are set in
``finalizeBCConstant``.
```

The `userheat.cpp` file implement these methods:

```
#include <userHeat.hpp>

namespace felisce
{
  UserHeat::UserHeat():
  LinearProblemHeat()
  {}

  void UserHeat::defineBC()
  {
    int iTemperature = _listVariable.getVariableIdList(temperature);
```

(continues on next page)

(continued from previous page)

```
    Variable temperature = _listVariable.listVariable()[iTemperature];

    _BoundaryConditionList.add(new BoundaryCondition(Dirichlet, Constant, 1, temperature,
↪ Comp1));
    _BoundaryConditionList.add(new BoundaryCondition(Dirichlet, Constant, 0, temperature,
↪ Comp1));
  }

  void UserHeat::finalizeBCConstant()
  {
    _BoundaryConditionList.getList()[0]->setValue(1.);
    _BoundaryConditionList.getList()[1]->setValue(0.);
  }
}
```

```
int iTemperature = _listVariable.getVariableIdList(temperature);
Variable temperature = _listVariable.listVariable()[iTemperature];
```

We select the variable on which the boundary contion will apply: the temperature.

```
_BoundaryConditionList.add(new BoundaryCondition(Dirichlet, Constant, 1, temperature,␣
↪Comp1));
```

The boundary condition is a constant Dirchlet condition that apply to the mesh region with label 1, on the first component of the physical variable temperature.

```
_BoundaryConditionList.add(new BoundaryCondition(Dirichlet, Constant, 0, temperature,␣
↪Comp1));
```

The similar boundary condition now apply on region of label 0.

```
_BoundaryConditionList.getList()[0]->setValue(1.);
_BoundaryConditionList.getList()[1]->setValue(0.);
```

We now specify the values corresponding to the two boundary conditions.

```
[transient]

timeStep = 0.01
timeMax = 0.2
time = 0.

[variable]

variable = 'temperature'
typeOfFiniteElement = '0'
degreeOfExactness = '2'
domain = 'all'

[mesh]

meshDir = ./../../../../data/mesh/
inputMesh = square_tria_4kv.mesh
```

(continues on next page)

(continued from previous page)

```
outputMesh = temperature.geo
resultDir = ./Solution/
prefixName = temperature

# list of eltType keyword:
# Nodes Segments2 Segments3 Triangles3 Triangles6 Quadrangles4
# Quadrangles8 Quadrangles9 Tetrahedra4 Tetrahedra10 Pyramids5
# Pyramids13 Prisms6 Prisms15 Hexahedra8 Hexahedra20
# Hexahedra26 Hexahedra27

[Triangles3]

label = '0'
stringRef = 'domain'

[Segments2]

label = '0 1 2 3'
stringRef = 'up down left right'

[petsc]

# List of solver :
# chebychev cg gmres preonly bicg python
#To use mumps: solver = preonly

solver = gmres

# List of preconditioner :
# none jacobi sor lu bjacobi ilu asm cholesky
#To use mumps: preconditioner = lu

preconditioner = ilu
relativeTolerance = 10.e-6
absoluteTolerance = 10.e-10
maxIteration = 10000
```

The configuration file is divided into sections. For now, interessing sections are:

```
[mesh]

meshDir = ./../../../../data/mesh/
inputMesh = square_tria_4kv.mesh
outputMesh = temperature.geo
resultDir = ./Solution/
prefixName = temperature
```

In the, `mesh` section can be found:

- meshDir: the directory of the mesh file.
- inputMesh: the mesh file with extension determing the file format (.mesh for medit and .geo for ensight).
- ouputMesh: the output mesh with extension.

- resultDir: where the solution is written.
- prefixName: the prefix of the solution file names.

```
# list of eltType keyword:
# Nodes Segments2 Segments3 Triangles3 Triangles6 Quadrangles4
# Quadrangles8 Quadrangles9 Tetrahedra4 Tetrahedra10 Pyramids5
# Pyramids13 Prisms6 Prisms15 Hexahedra8 Hexahedra20
# Hexahedra26 Hexahedra27
```

Here is the list of the element types manageable and defined in Felisce.

```
[Triangles3]

label = '0'
stringRef = 'domain'

[Segments2]

label = '0 1 2 3'
stringRef = 'up down left right'
```

These sections define a mapping between the integer element labels in medit format and the string element labels in ensight format (this is optional). The string label are used during visualization.

CHAPTER

TWO

# TUTORIAL: POISSON EQUATION

## 2.1 Introduction

- mathematics

$$\Delta U = f$$

- test configuration
- geometry
- mesh
- bondaay condition
- initial condition
- source term: f

## 2.2 Poisson

This case is just an extension of Laplace example. We present here the protocol to add a source term f.

**Go to the `$FELISCE_ROOT/Tests/Example/Heat`, you will find 5 files:**

- `main.cpp`: contain the main progam which is executed.
- `userHeat.hpp` and `userHeat.cpp`: where we specify how to build the linear system.
- `Makefile`: to build the programm.
- `data`: configuration file.

Consider the file `main.cpp`:

```
#include "Model/heatModel.hpp"
#include <userHeat.hpp>
using namespace felisce;

int main(const int argc, const char** argv)
{
  HeatModel model;
  model.initializeModel(argc, argv);

  model.initializeLinearProblem( new UserHeat());
```

(continues on next page)

(continued from previous page)

```
  //Loop time
  while (!model.hasFinished()){
    model.forward();
  }
}
```

This new physical problem doesn't imply any changes in the main program with the Felisce structure.

The UserHeat class gives some possibilities at the user to change: boundary conditions, source term, etc. With these modification we complete the system $Ax = b$ to be consistent with user formulation.

```
#include <Solver/linearProblemHeat.hpp>
#include <math.h>

namespace felisce
{
 class MyFunctionTime{
 public:
   double operator() (double x,double y,double z, double t) const
   {
                return (1.-x)*x*(1.-y)*y*cos(t)+2.*sin(t)*(y*(1.-y) + x*(1.-x));
   }
 };

 class UserHeat:
 public LinearProblemHeat
 {
 public:
   UserHeat();
   ~UserHeat();
   void initPerElementType(ElementType eltType, const int&␣
↪flagMatrixRHS=FlagMatrixRHS::matrix_and_rhs);
   void computeElementArray(vector<Point*>& elemPoint, vector<int>& elemIdPoint, int&␣
↪iel, const int& flagMatrixRHS=FlagMatrixRHS::matrix_and_rhs);
   void defineBC();
   void finalizeBCConstant();
 private:
   ElementField* _elemFieldTemperature;
   MyFunctionTime _fctTime;
 };
}
```

We use a functor to define the function $f$ and to use specific mathematic function you must include the `math.h` file.

```
class MyFunctionTime{
public:
  double operator() (double x,double y,double z, double t) const
  {
        return (1.-x)*x*(1.-y)*y*cos(t)+2.*sin(t)*(y*(1.-y) + x*(1.-x));
  }
};
```

After that just declare a function in the class:

```
MyFunctionTime _fctTime;
```

To use this function we are working with a pointer on the Felisce object: **ElemField**. An ElementField contains the values of a scalar field, for example a diffusion coefficient or a source term, at the element level. It can be either constant, defined on the degree of freedom of the finite element or defined on the quadrature points.

```
ElementField* _elemFieldTemperature;
```

In the matrix and second member assembling procedure (usually call: assembly loop) the two virtual functions **initPerElementType** and **computeElementArray** are call in the class **LinearProblem**. To use source term we add some details on it that why we declare the functions.

```
void initPerElementType(ElementType eltType, const int&␣
↪flagMatrixRHS=FlagMatrixRHS::matrix_and_rhs);
void computeElementArray(vector<Point*>& elemPoint, vector<int>& elemIdPoint, int& iel,␣
↪const int& flagMatrixRHS=FlagMatrixRHS::matrix_and_rhs);
```

The `userheat.cpp` file implement these methods:

```
#include <userHeat.hpp>
namespace felisce
{
  UserHeat::UserHeat():
  LinearProblemHeat(),
  _elemFieldTemperature(NULL)
  {}

  UserHeat::~UserHeat()
  {
    if (_elemFieldTemperature)
            delete _elemFieldTemperature;
  }

  void UserHeat::initPerElementType(ElementType eltType, const int& flagMatrixRHS)
  {
    LinearProblemHeat::initPerElementType(eltType);
    int iTemperature = _listVariable.getVariableIdList(temperature);
    CurrentFiniteElement& fe = *_listCurrentFiniteElement.
↪listCurrentFiniteElement()[iTemperature];

    //ElementField initialisation
    _elemFieldTemperature = new ElementField(ElementField::QUAD_POINT_FIELD, fe);
  }

  void UserHeat::computeElementArray(std::vector<Point*>& elemPoint, std::vector<int>&␣
↪elemIdPoint, int& iel, const int& flagMatrixRHS)
  {
    LinearProblemHeat::computeElementArray(elemPoint,elemIdPoint, iel);

    int iTemperature = _listVariable.getVariableIdList(temperature);
    CurrentFiniteElement& fe = *_listCurrentFiniteElement.
↪listCurrentFiniteElement()[iTemperature];
```

(continues on next page)

(continued from previous page)

```
    _elemFieldTemperature->setValue(_fctTime, fe,_fstransient->time,0);
        _elemVec->source(1.,fe,*_elemFieldTemperature,0,1);
  }

  void UserHeat::defineBC()
  {
    int iTemperature = _listVariable.getVariableIdList(temperature);
    Variable temperature = _listVariable.listVariable()[iTemperature];

    _BoundaryConditionList.add(new BoundaryCondition(Dirichlet, Constant, 0, temperature,
↪ Comp1));
        _BoundaryConditionList.add(new BoundaryCondition(Dirichlet, Constant, 2,␣
↪temperature, Comp1));
  }

  void UserHeat::finalizeBCConstant()
  {
    _BoundaryConditionList.getList()[0]->setValue(1.);
    _BoundaryConditionList.getList()[1]->setValue(0.);
  }
}
```

### 2.2.1 Constructor/Destructor

Initialize value in the constructor.

```
UserHeat::UserHeat():
 LinearProblemHeat(),
 _elemFieldTemperature(NULL)
 {}
```

We add a command in the destructor to suppress pointer *_elemFieldTemperature* to free memory.

```
UserHeat::~UserHeat()
 {
   if (_elemFieldTemperature)
    delete _elemFieldTemperature;
 }
```

### 2.2.2 Function initPerElementType

Call function in the mother class.

```
LinearProblemHeat::initPerElementType(eltType);
```

The object **CurrentFiniteElement** contain all information (geometric finite element, reference finite element, quadrature point) about the finite element choose to support *temperature* variable. *fe* is a reference on this finite element and is used to allocate elementary array (elementary matrix, elemtary vector, elementary field).

```
int iTemperature = _listVariable.getVariableIdList(temperature);
CurrentFiniteElement& fe = *_listCurrentFiniteElement.
↪listCurrentFiniteElement()[iTemperature];
_elemFieldTemperature = new ElementField(ElementField::QUAD_POINT_FIELD, fe);
```

### 2.2.3 Function computeElementArray

Call function in the mother class and take a reference on finite element associate to *temperature*.

```
LinearProblemHeat::computeElementArray(elemPoint,elemIdPoint, iel);
int iTemperature = _listVariable.getVariableIdList(temperature);
CurrentFiniteElement& fe = *_listCurrentFiniteElement.
↪listCurrentFiniteElement()[iTemperature];
```

The function **setValue** fill *_elemFieldTemperature* with value calculate from *_fctTime* (source term function define on `userHeat.hpp`)`.

```
_elemFieldTemperature->setValue(_fctTime, fe,_fstransient->time,0);
```

The function **source** evaluate the term $\int_T f.u$.

```
_elemVec->source(1.,fe,*_elemFieldTemperature,0,1);
```

### 2.2.4 Boundary Conditions

We just apply constant Dirichlet boundary conditions.

```
void UserHeat::defineBC()
{
  int iTemperature = _listVariable.getVariableIdList(temperature);
  Variable temperature = _listVariable.listVariable()[iTemperature];

  _BoundaryConditionList.add(new BoundaryCondition(Dirichlet, Constant, 0, temperature,␣
↪Comp1));
  _BoundaryConditionList.add(new BoundaryCondition(Dirichlet, Constant, 1, temperature,␣
↪Comp1));
  _BoundaryConditionList.add(new BoundaryCondition(Dirichlet, Constant, 2, temperature,␣
↪Comp1));
  _BoundaryConditionList.add(new BoundaryCondition(Dirichlet, Constant, 3, temperature,␣
↪Comp1));
}

void UserHeat::finalizeBCConstant()
{
  _BoundaryConditionList.getList()[0]->setValue(0.);
  _BoundaryConditionList.getList()[1]->setValue(0.);
  _BoundaryConditionList.getList()[2]->setValue(0.);
  _BoundaryConditionList.getList()[3]->setValue(0.);
}
```

CHAPTER

THREE

# INDICES AND TABLES

- genindex
- modindex
- search