Cite as: Sankar Raju.N : Evoking existing function objects and creating new user-defined function objects for Post- Processing. In Proceedings of CFD with OpenSource Software, 2017, Edited by Nilsson. H., `http://dx.doi.org/10.17196/OS_CFD#YEAR_2017`

CFD WITH OPENSOURCE SOFTWARE

A COURSE AT CHALMERS UNIVERSITY OF TECHNOLOGY
TAUGHT BY HÅKAN NILSSON

Evoking existing function objects and creating new user-defined function objects for Post- Processing

Developed for OpenFOAM-17.06

Author:
SANKAR RAJU . N

Peer reviewed by:
SURYA KAUNDINYA ORUGANTI
EBRAHIM GHAHRAMANI (CTH)

January 8, 2018

Acknowledgements

First, I wish to convey my sincere thanks to Prof.Dr. Håkan Nilsson for organizing the Open Source CFD course and providing it as an effective platform for spreading the knowledge of Open-FOAM for users from versatile arenas.

Special thanks to scientists from DESY for formulating such an interesting problem statement. Also I wish to thank my friends from MPI, DESY, Chalmers , IITM & AU for their continuous support and encouragement.

Last but not the least, I would like to thank my friends of this course from Juelich and Lyon. Indeed their perseverence and hard work are worthy of admiration and gratitude.

Learning outcomes

The reader will learn:

- An explanatory guide for the field and force function objects for post processing in OpenFOAM
- A brief description of the library classification depending on the inheritance of function objects.
- Procedure to evoke Evoke the existing function objects which are present in the OpenFOAM library by default for post-processing and to control them on run time.
- To create a new user-defined (customary)function object for Post Processing using a detailed step by step methodology and creating a template for creating newer customary ones on demand.
- A sample tutorial to utilize the created function object.

Prerequisites

The reader is expected to know the following in order to get maximum benefit out of this report:

- Skansholm, Jan ”C++ from the Beginning” Pearson Education (2004).
- White, Frank M. ”Fluid mechanics. 5th.” Boston: McGraw-Hill Book Company (2003).

Contents

Chapter 1

Evoking Existing Function Objects

1.1 Function Object

This section initially discusses briefly on the various types of function objects available for post-processing. And consequently, the detailed procedure of evoking existing function objects is explained with a some examples.

1.1.1 Function object libraries for post processing

The post processing in OpenFOAM is performed using the function objects. The function objects are options that are run time selectable. The results generated by function objects can be both obtained in the due course of simulation and after the calculation.

The current report primarily deals with the explanation for the run time - result generation for post-processing. The various function objects are classified as follows:

1. Field
2. Forces
3. Graphics
4. Lagrangian
5. Sampling
6. Solvers
7. Utilities

Of all the library categories the function objects that is of interests are Fields and Forces, as these are the type types of libraries that are repeatedly utilized in all the simulations performed in OpenFOAM.

1.1.2 Fields

The field function objects can be also called as Reaction sensitivity analysis function objects, as they act as a relative performance analysis functions between the thermophysical function objects and the analysis (in Figure 1.1). The fields library category consists of various functions that can be represented as scalar fields.

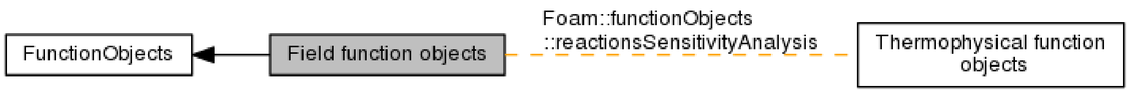
FunctionObjects
Field function objects
Foam::functionObjects ::reactionsSensitivityAnalysis
Thermophysical function objects

Figure 1.1: Collaboration diagram of field function object

The function objects that are inherited from the field expression with/without writeFile is mentioned in the following section.The function object which inherits a write file, writes it's corresponding file in each of the solution folder for every time step. Though the outline of the function object code is same among the function objects, the dependencies highly vary and hence a deeper understanding of the classification is important.

1. components

   This is used to calculate the components of a field

2. CourantNo

   This function object calculates and outputs the Courant number as a volScalarField. The field is stored on the mesh database so that it can be retrieved and used for other applications.

3. Curle

   This function object calculates the acoustic pressure based on Curle's analogy.

4. div

   This function object calculates the divergence of a field. The operation is limited to surfaceScalarFields and volVectorFields, and the output is a volScalarField.

5. enstrophy

   This function object calculates the enstrophy of the velocity.

6. flowType

   This function object calculates and writes the flowType of a velocity field.

7. flux

   This function object calculates and writes the flux of a field. The operation is limited to surfaceVectorFields and volVectorFields, and the output is a surfaceScalarField.

8. grad

   This function object calculates the gradient of a field.

9. Lambda2

   This function object calculates and outputs the second largest eigenvalue of the sum of the square of the symmetrical and anti-symmetrical parts of the velocity gradient tensor.

10. blendingFactor

    This function object calculates and outputs the blendingFactor as used by the bended convection schemes. The output is a volume field (cells) whose value is calculated via the maximum blending factor for any cell face.

11. MachNo

    This function object calculates and writes the Mach number as a volScalarField.

12. mag

    This function object calculates the magnitude of a field.

13. magSqr

    This function object calculates the magnitude of the sqr of a field.

14. PecletNo

    This function object calculates and outputs the Peclet number as a surfaceScalarField.

15. pressure

    This function object includes the tools to manipulate the pressure into different forms.

16. Q

    This function object calculates and outputs the second invariant of the velocity gradient tensor [1/s^2].

17. randomise

    This function object adds a random component to a field, with a specified perturbation magnitude.

18. streamFunction

    This function object calculates and outputs the stream-function as a pointScalarField.

19. subtract

    This one,from the first field subtract the remaining fields in the list. The operation can be applied to any volume or surface fields generating a volume or surface scalar field.

20. vorticity

    This function object calculates the vorticity, the curl of the velocity.

It is then sufficient to mention the respective field function object when the function object being called in the control dict. More explanation can be found in the evoking existing function object section.

The function objects that are inherited from fvMeshFunctionObject with/without writeFile are,

1. ddt2

    This function object calculates the magnitude squared of d(scalarField)/dt.

2. DESModelRegions

    This function object writes out an indicator field for DES turbulence calculations.

3. extractEulerianParticles

    This function object generates particle size information from Eulerian calculations. Example: Volume of Fluids

4. fieldAverage

    This function object calculates average quantities for a user-specified selection of volumetric and surface fields.

5. fieldCoordinateSystemTransform

    This function object transforms a user-specified selection of fields from global Cartesian to a local system. Also to note that the fields are run-time modifiable.

6. fieldMinMax

    This function object calculates the value and location of scalar minimum and maximum for a list of user-specified fields.

7. fieldValueDelta

   This function object provides an operation between two 'field value' function objects.

8. fluxSummary

   This function object calculates the flux across selections of faces.

9. histogram

   This function object writes the volume-weighted histogram of a volScalarField

10. mapFields

    This function object calculates the vorticity, the curl of the velocity.

11. nearWallFields

    This function object map fields from local mesh to secondary mesh at run-time.

12. particleDistribution

    This function object generates a particle distribution for lagrangian data at a given time.

13. processorField

    This function object writes a scalar field whose value is the local processor ID. The output field name is 'processorID'.

14. readFields

    This function object reads fields from the time directories and adds them to the mesh database for further post-processing.

15. regionSizeDistribution

    This function object creates a size distribution via interrogating a continuous phase fraction field

16. setFlow

    This function object provides options to set the velocity and flux fields as a function of time.

17. surfaceInterpolate

    This function object linearly interpolates volume fields to generate surface fields.

18. turbulenceFields

    This function object stores turbulence fields on the mesh database for further manipulation.

19. writeCellCentres

    This function object writes the cell-centres volVectorField and the three component fields as volScalarFields.

20. writeCellVolumes

    This function object writes the cell-volumes volScalarField.

21. XiReactionRate

    This function object writes the turbulent flame-speed and reaction-rate volScalarFields for the Xi-based combustion models.

22. yPlus

    This function object evaluates and outputs turbulence y+ for turbulence models.

23. zeroGradient

    This function object creates a volume field with zero-gradient boundary conditions from another volume field.

24. specieReactionRates

    This function object writes the domain averaged reaction rates for each specie for each reaction into the file <timeDir> /specieReactionRates.dat.

1.2 Evoking

All the pre-available or existing function objects are evoked or called from the controlDict.
Inorder to understand the evoking of the existing function objects,the classification made in the previous section is considered.

1.2.1 Function objects depended on the field expression

For the function objects which are just dependent on the function object field expression, it's mandatory to call the type of function object that one wants to compute and the library on which it is dependent on. If one wants to calculate only in a specific region, then the line referring the region can be mentioned (example,using patches).

The base template can be written as,

```
fieldExpressionFunctionObjectName
{
type        fieldExpressionName;
libs        ("libfieldFunctionObjects.so");
}
```

Example 1
Let's revoke a the Courant Number function object:

```
CourantNo1
{
   type        CourantNo;
   libs        ("libfieldFunctionObjects.so");
}
```

Here the function object type is mentioned as CourantNo. The `libfieldFunctionObjects.so` shared object library of the function object is utilized by the Courant No. function object. Hence Courant number calculated provides a measure of the rate at which information is transported under the influence of a flux field.

Example 2
Let's revoke the Q criterion:

```
QCriterion
{
type  Q;
libs ("libfieldFunctionObjects.so");
field U;
result myQ;
}
```

In the above Example 2, one could find that the particular field (Velocity) that needs to be computed is mentioned. Secondly the result is asked to be displayed with the name myQ.

Example 3
Let's calculate the y+ of the specific regions of a test case (say dam Break case- in this example)

```
yPlus1
{
type yPlus;
libs ("libfieldFunctionObjects.so");
patches (leftWall rightWall lowerWall);
}
```

In the y+ example, the particular sections have been mentioned were the calculation needs to be performed.

If the particular regions of interest aren't mentioned, then the calculation is made for the entire region. Hence, in order to save computational resources, one needs to act wisely on selection of the various sections that are to be calculated.

Example 4
Let's compute a particular turbulence field parameter

```
turbulenceFields1
{
   type             turbulenceFields;
   libs             ("libfieldFunctionObjects.so");
   field            R;
}
```

Here the stress tensor parameter`R` is alone calculated using the `turbulenceFields` function object. The other turbulence field parameters that can be measured are turbulence kinetic energy (`k`), dissipation rate of turbulence kinetic energy (`epsilon`), specific dissipation rate (`omega`). For incompressible cases, one can additionally measure eddy kinematic viscosity (`nut`), effective kinematic viscosity (`nuEff`), deviatoric stress tensor (`devReff`). And for the compressible cases, one can additionally measure eddy dynamic viscosity (`mut`), effective dynamic viscosity (`muEff`), thermal eddy diffusivity (`alphat`), effective eddy thermal diffusivity (`alphaEff`), deviatoric stress tensor (`devRhoReff`).

Chapter 2

User-defined Post-Processing function object

This chapter deals with the detail description of the steps to be followed for creating a new function object for post- processing. And a greater is given for the creation of a new scalar field function for calculations. The initial section deals with the steps to be followed for creating a new function object. Following which a sample template with a function object named ”helicity” is created.

2.1 Steps

The basic steps for creating a new function object (here considered for fields and forces) for post-processing are described as follows.
Overall Procedure

1. Create .C and .H files for the function object that is intended to be created
2. Copy one set of .C and .H files created into the ”lnInclude” folder
3. Follow the compiling procedure for the custom developed function object. A detailed methodology to compile a custom defined function object has been described in the last section of this chapter.

Procedure to create *.C file

The .C file created consists of 5 major sections, which are described as follows.

1. Header files
    - In this section one needs to include all the header files necessary for computation by the Private Member Functions. As the post-processing results are intended to be computed during the run time, the run time header file and the header file of the current function object are mandatorily included, which are represented as,

    ```
    #include "helicity.H"
    #include "addToRunTimeSelectionTable.H"
    ```

    - The other header files are the ones necessary for computation.

2. Static Data Members

   - The general outline of the static data member function can be written as:

     ```
     1 namespace Foam
     2 {
     3 namespace functionObjects
     4 {
     5 defineTypeNameAndDebug(userDefFnObjName, 0);
     6 addToRunTimeSelectionTable(functionObject,userDefFnObjName,dictionary);
     7 }
     8 }
     ```

   - The line 5 links to the first 0 file of the computation case intended. The line 6 mentions that functionObject is added to AddToTable and thereby adding the user Defined Function Object's constructor to the hash table.

3. Private Member Functions

   - The private Member functions include the definition of the function object to be computed. It is suggested to note that the member functions have direct access to all data members and member function of the class. The general outline of the private Member function can be written as:

     ```
     bool Foam::functionObjects::userDefFnObjName::calc()
     {
         return calcAllTypes(*this);
     }
     ```

4. Constructors

   - The constructors are the ones that are used to initialize the attributes. The general outline of the Constructors can be written as:

     ```
     Foam::functionObjects::userDefFnObjName::userDefFnObjName
     (
     const word& name,
     const Time& runTime,
     const dictionary& dict
     )
     :
     fieldsExpression(name, runTime, dict)
     {
          setResultName("userDefFnObjName");
      }
     ```

5. Destructor

   - A destructor is a member function without parameters, with the same name as the class, but with a ˜(tilda) in front of it. The destructor is defined explicitly, so as to ensure that all the memory is returned.
   - The general outline of the destructor can be written as:

     ```
     Foam::functionObjects::userDefFnObjName::~userDefFnObjName()
        {}
     ```

Procedure to create *.H file

1. Inclusion of header files

   - The ifndef is used to check if the header file has been defined earlier in the include file or in the file. And the command is hence written as:

     ```
     #ifndef functionObjects_userDefFnObjName_H
     ```

     next the header file is defined as follows:

     ```
     #define functionObjects_add_H
     ```

     And Finally the inherited header files are mentioned. Say, the field Expression header file or the fvMeshFunction header file

     ```
     #include "fieldsExpression.H"
     ```

2. Declaration of the user defined function object for Post-Processing

   This section should contain the member function (either public or private), with their corresponding constructors and destructor.

Chapter 3

Helicity

The creation of the helicity.C and helicity.H files are discussed as follows.

3.1 Creation of helicity.C

Based on the methodology stated in the section 2.1 for creation of new *.C file, the helicity.C is created as,

```
\*---------------------------------------------------------------------------*/

#include "helicity.H"
#include "fvcCurl.H"

#include "addToRunTimeSelectionTable.H"



// * * * * * * * * * * * * * * Static Data Members * * * * * * * * * * * * * //

namespace Foam
{
namespace functionObjects
{
   defineTypeNameAndDebug(helicity, 0);

   addToRunTimeSelectionTable
   (
       functionObject,
       helicity,
       dictionary
   );
}
}


// * * * * * * * * * * * * * Private Member Functions  * * * * * * * * * * * //

bool Foam::functionObjects::helicity::calc()
{
if (foundObject<volVectorField>(fieldName_))
33 {
34    return store
35    (
36    resultName_,
37    fvc::curl(lookupObject<volVectorField>(fieldName_)) &
      lookupObject<volVectorField>(fieldName_)
38    );
39  }
40    else
41    {
42        return false;
43    }
44
45    return true;
46 }
47
48
49 // * * * * * * * * * * * * * * * * Constructors  * * * * * * * * * * * * * * //
50
51 Foam::functionObjects::helicity::helicity
52 (
53    const word& name,
54    const Time& runTime,
55    const dictionary& dict
56 )
57 :
58    fieldExpression(name, runTime, dict, "U")
59 {
60    setResultName(typeName, fieldName_);
61 }
62
63
64 // * * * * * * * * * * * * * * * * Destructor  * * * * * * * * * * * * * * * //
65
66 Foam::functionObjects::helicity::~helicity()
67 {}
69
70
71 // ************************************************************************* //
```

Explanation

- In the section 1 of the file, the header files are included. In the line 3, the user defined header file, helicity.H is included. In the line 6, the header file "addToRunTimeSelectionTable" is included, which adds the derived constructor("helicity" in this case) to the hash table held by base. The only extra necessary header file is the "fvCurl.H". This function object's header file is hence utilized in the Private Member Function Section.

- In the Static Data Member section, the type name of the function object is specified. Hence this forms the type name to be called for in the controlDict of the simulated case 18-23 is used to mention that the simulation needs to be computed on the due course of simulation, where the helicity file is created in every directory file created (for every time step). In the template of the helicity.C attached in the appendix, one could see the header comments too.

3.2 Creation of helicity.H

Based on the methodology stated in the section 2.1 for creation of new *.H file, the helicity.H is created as follows:

```
 \*---------------------------------------------------------------------------*/

#ifndef functionObjects_helicity_H
#define functionObjects_helicity_H

#include "fieldExpression.H"

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

namespace Foam
{
namespace functionObjects
{

/*---------------------------------------------------------------------------*\
                         Class helicity Declaration
\*---------------------------------------------------------------------------*/

class helicity
:
   public fieldExpression
{
   // Private Member Functions

       //- Calculate the helicity field and return true if successful
       virtual bool calc();


public:

   //- Runtime type information
   TypeName("helicity");


   // Constructors

       //- Construct from Time and dictionary
       helicity
       (
           const word& name,
           const Time& runTime,
           const dictionary& dict
       );


   //- Destructor
   virtual ~helicity();
};


49 // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
50
51 } // End namespace functionObjects
52 } // End namespace Foam
53
54 // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
55
56 #endif
57
58 // ************************************************************************* //
```

Explanation

- Line 1 with the "ifndef" is used to check if the header file "helicity.H" has been defined earlier in the include file of function object directory or in the file. Line 2 with the "define" is used to define the header file "helicity.H". In the helicity class declaration section, as the helicity function object just depends on the field expression, the member function is defined to calculate the field expression. And this has been set as public.
- The type name to be used to evoke it in the controlDict is mentioned as "helicity" in the line 30. Following which, helicity contructor is mentioned. Here one can find that values that needs to be displayed are significantly mentioned with the location at which the files need to be saved. As the base class is polymorphic, the virtual destructor has been utilized in the line 45.

3.3 Compiling

This section deals with the complete compiling procedure of a custom defined function object for post processing in OpenFOAM. Here the procedure is based on the helicity function object that is intended to be performed.

Step 1:

- Since, there might be a good number of dependency files, it's suggested to create a new folder named "helicity" in the function object file of the source folder of OpenFOAM installation.

Step 2:

- The .C and .H files created for helicity are copied in the helicity folder. Also, the Make folder is copied from an existing one.

Step 3:

- The file named "file" is opened inside the Make folder. In this document, change the "executable name" to "helicity". If one is utilizing the make folder available inside the function object directory under source directory(field or force directory), it's sufficient to include a new executable name for helicity.

```
helicity/helicity.C // in helicity case
```

Step 4:

- This step is optional,while this can be beneficial for the ones who have copied the templates from existing ones. In the .C file created/copied, one can change the application name and the description in the comment header. This step might be beneficial if this helicity folder files are used as a template to create your own function objects.

Step 5:

- Copy the .C and .H files that has been created for the new function object into the lnInclude folder.

Step 6:

- Go to the directory of helicity, i.e, function objects, fields

Step 7:

- In the terminal type:

```
wclean
```

Step 8:

- This is an effective step for compiling, as this avoids permission errors. Type :

```
sudo bash
```

- Enter your system password. Now one can compile as a super user.

Step 9:

- Now, inorder to compile, type:

  ```
  wmake
  ```

  Thus we compile all the necessary. Here, one compiles as a super user.

Step 10:

- Inorder to check if OpenFOAM recognizes the function object that we have compiled, type:

  ```
  helicity -help
  ```

  The word helicity can be replaced with the user defined function object. To exit the super user authorization, type:

  ```
  ls
  exit
  ```

Thus, our new function object - ”helicity” has be created successfully. And now one can evoke the helicity function object in the controlDict as,

```
helicity1
{
   type helicity;
   libs ("libfieldFunctionObjects.so");
}
```

Hence on running the simulation, the helicity function object is computed, similar to the other function objects.

3.4 Tutorial

3.4.1 Test Case

A new tutorial has been created to evaluate the helicity. The tutorial is based on the damBreak tutorial case of interFoam solver. It is suggested to follow the steps to create .H and .C files for helicity using the methodology suggested in the previous sections. And hence this tutorial primarily demonstrates the evoking the function object (Helicity) created in a test case (damBreak tutorial case). First copy the tutorial to run folder and rename it as damBreak$_helicity$:

```
cp -r $FOAM_TUTORIAL/multiphase/interFoam/RAS/damBreak/
      $FOAM_RUN/damBreak_helicity
cd $FOAM_RUN/damBreak_helicity
```

Since the aim of this tutorial is to showcase the procedure to use helicity function object, the same application is followed for damBreak-helicity without any change in geometry or physical parameters.

To block Mesh the geometry, type:

```
blockMesh
```

Add the following in the `system/controlDict` and save it.

```
function
{
   helicity1
     {
        type helicity;
        libs ("libfieldFunctionObjects.so");
     }
}
```

It's sufficient to add the other function objects continuously if needed. (Refer the evoking tutorial)

Ensure to compile the helicity function object before running the case(follow the steps in Section 3.3). And finally to run the case type `interFoam`

As a result one could see that the results for `helicity` has been calculated in each time step.

Chapter 4

Study Questions

1. What is the necessity of creating new function objects for post-processing, in the due course of simulation ?
2. What is the ultimate use of the helicity ?
3. Which header file is the most mandatory one in the *.C file created for the function object?
4. How to save the computational time, without compromising on the utilization of the run time post-processing function objects ?
5. What are the 3 main files that are needed to create a new user-defined function?
6. Name some applications where the helicity function object is of prime importance

REFERENCE

1. Skansholm, Jan "C++ from the Beginning" Pearson Education (2004).
2. OpenFOAM v1706, "The open source CFD toolbox".
3. White, Frank M. "Fluid mechanics. 5th." Boston: McGraw-Hill Book Company (2003).

Chapter 5

APPENDIX

5.1 Helicity.C

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     |
    \\  /    A nd           | Copyright (C) 2014-2016 OpenFOAM Foundation
     \\/     M anipulation  |
-------------------------------------------------------------------------------
License
    This file is part of OpenFOAM.

    OpenFOAM is free software: you can redistribute it and/or modify it
    under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    OpenFOAM is distributed in the hope that it will be useful, but WITHOUT
    ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
    FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License
    for more details.

    You should have received a copy of the GNU General Public License
    along with OpenFOAM.  If not, see <http://www.gnu.org/licenses/>.

\*---------------------------------------------------------------------------*/

#include "helicity.H"
#include "fvcCurl.H"

#include "addToRunTimeSelectionTable.H"
```

```
// * * * * * * * * * * * * * * Static Data Members * * * * * * * * * * * * * //

namespace Foam
{
namespace functionObjects
{
    defineTypeNameAndDebug(helicity, 0);

    addToRunTimeSelectionTable
    (
        functionObject,
        helicity,
        dictionary
    );
}
}
```

```
// * * * * * * * * * * * * * Private Member Functions  * * * * * * * * * * * //
```

```
bool Foam::functionObjects::helicity::calc()
{
    if (foundObject<volVectorField>(fieldName_))
    {
     return store
     (
      resultName_,
      fvc::curl(lookupObject<volVectorField>(fieldName_)) &
      lookupObject<volVectorField>(fieldName_)
     );
    }
    else
    {
        return false;
    }

    return true;
}
```

```
// * * * * * * * * * * * * * * * * Constructors  * * * * * * * * * * * * * * //




Foam::functionObjects::helicity::helicity
(
    const word& name,
    const Time& runTime,
    const dictionary& dict
)
:





    fieldExpression(name, runTime, dict, "U")
{
    setResultName(typeName, fieldName_);
}


// * * * * * * * * * * * * * * * * Destructor  * * * * * * * * * * * * * * * //
Foam::functionObjects::helicity::~helicity()
{}
// ************************************************************************* //
```

5.2 Helicity.H

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     |
    \\  /    A nd           | Copyright (C) 2014-2016 OpenFOAM Foundation
     \\/     M anipulation  | Copyright (C) 2016 OpenCFD Ltd.
-------------------------------------------------------------------------------
License
    This file is part of OpenFOAM.

    OpenFOAM is free software: you can redistribute it and/or modify it
    under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    OpenFOAM is distributed in the hope that it will be useful, but WITHOUT
    ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
    FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License
    for more details.

    You should have received a copy of the GNU General Public License
    along with OpenFOAM.  If not, see <http://www.gnu.org/licenses/>.

Class
    Foam::functionObjects::helicity

Group
    grpFieldFunctionObjects

Description
    Calculates the helicity, dot product the curl of the velocity and velocity.

    The field is stored on the mesh database so that it can be retrieved
    and used for other applications.

Usage
    \verbatim
    helicity1
    {
        type        helicity;
        libs        ("libfieldFunctionObjects.so");
        ...
    }
    \endverbatim
    Where the entries comprise:
    \table
        Property     | Description             | Required    | Default value
        type         | Type name: helicity     | yes         |
        U            | Name of velocity field  | no          | U
        result       | Name of Courant number field | no    | \<function name\>
        log          | Log to standard output  | no          | yes
    \endtable

See also
    Foam::functionObjects::fieldExpression
    Foam::functionObjects::fvMeshFunctionObject

SourceFiles
    helicity.C

\*---------------------------------------------------------------------------*/

#ifndef functionObjects_helicity_H
#define functionObjects_helicity_H

#include "fieldExpression.H"

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

namespace Foam
{
namespace functionObjects
{
```

```
/*---------------------------------------------------------------------------*\
                         Class helicity Declaration
\*---------------------------------------------------------------------------*/

class helicity
:
    public fieldExpression
{
    // Private Member Functions

        //- Calculate the helicity field and return true if successful
        virtual bool calc();


public:

    //- Runtime type information
    TypeName("helicity");


    // Constructors

        //- Construct from Time and dictionary
        helicity
        (
            const word& name,
            const Time& runTime,
            const dictionary& dict
        );


    //- Destructor
    virtual ~helicity();
};


// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

} // End namespace functionObjects
} // End namespace Foam

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

#endif

// ************************************************************************* //
```