multiSolver extension

multiSolver has been publicly released on github at https://github.com/Marupio/equationReader/
Refer to this website for the latest documentation and source code.

1 Introduction

1.1 What is it?

multiSolver is a master control class that allows you to create a superSolver composed of multiple solvers within a superLoop. All solvers operate on the same dataset in sequence. For example:

1. `icoFoam` - runs to completion;
2. data is handed over to `scalarTransportFoam`;
3. `scalarTransportFoam` - runs to completion;
4. data is handed back to `icoFoam`, and the superLoop repeats.

1.2 What it isn't

You should not use **multiSolver** to combine the physics of two solvers. For example, say you want to model bubbles in a porous medium. That's like combining `porousSimpleFoam` with `bubbleFoam`. You might be tempted to use **multiSolver** for that, but don't.

Each time it switches between solvers, **multiSolver** writes everything to disk, clears the memory, then reloads everything all over again. This is not efficient. If you plan to switch between solvers at every timestep, **multiSolver** is not a good idea. Rather, you should write a single solver that combines them within its algorithm.

1.3 Features

- **Multiple solvers** - multiple solvers can be used in sequence on the same data set.
- **Changing boundary conditions** - the boundary conditions can change at distinct time intervals.
- **Independent time** - each solver can operate with an independent time value, although universal time can still be used.
- **Single case directory** - the settings for all solvers are stored within a single case directory using a "multiDict" dictionary format.
- **Easy data management** - All the data output is sorted into subdirectories corresponding to the solver, and can be loaded / unloaded using the multiPost utility.
- **Store fields** - To save memory and hard drive space, not all solvers have to use all the fields. Rather, they can "store" any unneeded fields, leaving more memory and disk space. The next solver retrieves all stored fields, and no data is lost.

1.4 Why would you need this?

A fundamental assumption in the design of OpenFOAM is the existence of a universal time. Therefore the time object is the top-level objectRegistry (i.e. **runTime** hosts the database for your simulation). This design works for nearly all simulations imaginable, except for those that require more than one time frame. For these situations, **multiSolver** will come in handy.

1.5 When would you need this?

The capabilities of **multiSolver** are useful for:

- multi-step processes to be modelled within a single application, e.g. fluid injection, followed by settling;
- modelling of a flow problem characterized by two different timescales, e.g. stirring with biochemical reactions; and

- changing boundary conditions mid-run.

Basically, if you find yourself:

- frequently copying data between case directories;
- frequently stopping and changing the simulation details, then restarting; or
- using `runTime++` more than once in your solver,

then **multiSolver** might help you.

1.6 Mesh motion not fully supported

NOTE: At this time, multiSolver allows for full mesh motion, provided the mesh returns to its original position between solvers. This functionality is planned for the future.

1.7 Update info

- *2010-07-23*: Initial import
- *2011-03-29*: Minor bug fix for 1.6-ext and 1.7.1
- *2011-04-05*: Major upgrade - now works for parallel simulations
- *2011-06-03*: Minor bug fix - decompose nolonger omits the initial directories
- *2011-07-01*: Overhauled the documentation on the wiki page (no change to the code)
- *2012-10-25:* **Version 0.5.0**
    - Moved to git
    - Enabled boundary switching
    - Introduced version numbers to keep track of changes
- *2013-08-29:* **Version 0.6.0**
    - Uploaded to github and OpenFOAM-extend
    - Restructured applications and tutorials directories for consistency
    - Made opening splash optional

2 Installation

2.1 OpenFOAM-extend

If you have a recent version of OpenFOAM-extend, you may already have **multiSolver** installed. Type this command:

```
[ -d "$WM_PROJECT_DIR/src/multiSolver" ]&&echo "Yes"||echo "No"
```

If the response is "Yes" then you already have it.

2.2 Git Installation

A git installation will allow you to download the latest updates to **multiSolver**.

multiSolver is *git-tracked* seperately from OpenFOAM, so if your OpenFOAM installation is also *git-tracked*, it is advisable to put it in a seperate directory. Alternatively, if you use the latest version of OpenFOAM-extend, **multiSolver** is incorporated within the main git repository.

Therefore, choose a seperate directory, for example:

```
-OpenFOAM
 |-OpenFOAM-2.2.x
 | |-applications
 | |-src
 | '-etc, and so on
 '-multiSolver
```

To duplicate the structure above:

```
cd $WM_PROJECT_DIR
cd ..
git clone https://github.com/Marupio/multiSolver.git
cd multiSolver/src/multiSolver
wmake libso
cd ../../applications/solvers/multiSolver/multiSolverDemo
wmake
cd ../../../utilities/postProcessing/multiSolver
wmake
```

To later update multiSolver:

```
cd $WM_PROJECT_DIR
cd ..
git pull
cd multiSolver/src/multiSolver
wmake libso
cd ../../applications/solvers/multiSolver/multiSolverDemo
wmake
cd ../../../utilities/postProcessing/multiSolver
wmake
```

2.3 Manual installation

To manually install multiSolver:

1. Download the code:

- get the latest code from https://github.com/Marupio/multiSolver/archive/master.zip
- or use the zip file stored on the website linked-to from the DOI.

2. Open a terminal window and browse to the folder with your download.

3. Execute the following commands. You should be able to just copy and paste all 10 lines into your terminal:

```
unzip multiSolver-master.zip
mv multiSolver-master/README multiSolver-master/tutorials/multiSolver
cp multiSolver-master/* $WM_PROJECT_DIR
rm -rf multiSolver-master
cd $WM_PROJECT_DIR/src/multiSolver
wmake libso
cd $FOAM_APP/utilities/postProcessing/multiSolver
wmake
cd $FOAM_APP/solvers/multiSolver/multiSolverDemo
wmake
```

multiSolver should now be installed.

3 Testing the installation

The installation comes with a demo application and test case. First copy the test case into your `run/tutorials` directory:

```
cp -rf $WM_PROJECT_DIR/tutorials/multiSolver $FOAM_RUN/tutorials
```

To run the test case:

```
cd $FOAM_RUN/tutorials/multiSolver/multiSolverDemo/teeFitting2d
blockMesh
multiSolverDemo
```

To view the results:

```
multiSolver -load all
multiSolver -set icoFoam1
paraFoam
```

3.1 About the test case

The demo application is:

1. `icoFoam1` - i.e. `icoFoam` with boundary conditions 1;
2. `scalarTransportFoam`;
3. `icoFoam2` - i.e. icoFoam with boundary conditions 2;
4. `scalarTransportFoam` (again);
5. repeat.

The test case is a 2-dimensional tee fitting. The boundary conditions are:

1. `icoFoam1`:

```
        Inlet
         ||
         ||
Closed ====== Outlet
```

2. `scalarTransportFoam`:

```
        T = 1
         ||
         ||
 zeroG ====== zeroG
```

3. `icoFoam2`:

```
        Outlet
         ||
         ||
Outlet ====== Inlet
```

4. `scalarTransportFoam`:

```
        T = 1
         ||
         ||
 zeroG ====== zeroG
```

The test case also has storeFields defined to demonstrate their use:

- `icoFoam` doesn't need the `T` field, so it *stores* this field; and
- `scalarTransportFoam` doesn't need the `P` field.

Since `icoFoam1` is the first to run, it must have all fields defined in its `initial/0` directory, even though it is storing the `T` field.

4 Using multiSolver

4.1 Case directory structure

The case directory for a *superSolver* is different than a standard solver. The directory transforms itself while the solver runs. It contains transient files that are continuously modified by **multiSolver**, and static files that you use to describe the change.

4.1.1 Data

Now, you don't have to worry about this as it is handled automatically, but the most notable difference between a *superSolver* and a standard solver is where they keep their data.

A *typical* normal solver

```
[caseName]
|-system
|-constant
|-0
|-0.2 [timeDirectory]
|-0.4 [timeDirectory]
... more [timeDirectories]
```

A *typical* superSolver

```
[caseName]
|-system
|-constant
'-multiSolver
  |-[solverDomain1]
  | |-initial
  | |-0 [superLoop]
  | | |-0 [timeDirectory]
  | | |-0.2 [timeDirectory]
  | | |-0.4 [timeDirectory]
```

```
| | ... more [timeDirectories]
| |-1 [superLoop]
| | |-1.2 [timeDirectory]
| | |-1.4 [timeDirectory]
| | |-1.6 [timeDirectory]
| | ... more [timeDirectories]
| ... more [superLoops]
|-[solverDomain2]
| |-initial
| |-0 [superLoop]
| | |-0 [timeDirectory]
| | |-0.2 [timeDirectory]
| | |-0.4 [timeDirectory]
| | ... more [timeDirectories]
| |-1 [superLoop]
| | |-1.2 [timeDirectory]
| | |-1.4 [timeDirectory]
| | |-1.6 [timeDirectory]
| | ... more [timeDirectories]
| ... more [superLoops]
... more [solverDomains]
```

In short, standard solvers keep their data in:

```
[caseName]/[timeValue]
```

whereas *superSolvers* sort their data into superLoop subdirectories within subdirectories named after the solverDomain:

```
[caseName]/multiSolver/[solverDomainName]/[superLoopIndex]/[timeValue]
```

The standard location of `case/[timeValue]` is used as a temporary loading area, mostly for post-processing. You don't have to worry about this, as it is handled automatically by **multiSolver** and its associated post-processing application.

4.1.2 Boundary changes

Changes to the mesh are not yet supported by **multiSolver**, but boundary changes between `patch` and `wall` are supported. To do this, create create multiple copies of the `constant/boundary` file with their respective solver domain names as extensions. For instance:

```
-constant
 |-boundary
 |-boundary.domain1
 |-boundary.domain2
 |-boundary.domain3
 |-faces
 |-neighbour
 |-owner
 `-points
```

4.1.3 multiDicts

A regular solver reads information from various dictionary files, and these affect its behaviour. When using **multiSolver**, there will be dictionaries whose contents need to be *different* for each solverDomain. To specify this behaviour, a *multiDict* is used.

multiDicts:

- sit in the same directory as the dictionary they are managing;
- have the prefix `multi`, followed by the name of their child dictionary; and
- are not required if the dictionary doesn't need to change between solvers.

For example, a typical `constant` directory might look like:

```
constant
|-reactionProperties          standard dictionary (does not change)
|-multiTransportProperties   multiDictionary (describes change)
'-transportProperties        auto-generated dictionary (changes)
```

In this directory, there are three files:

- `reactionProperties` - this is a standard dictionary. (You can tell because there's no `multiReactionProperties`.) It is user-editable and will not change during a run. **multiSolver** ignores these files;
- `multiTransportProperties` - this is a multiDict. It is also user-editable and will not change during a run. **multiSolver** recognizes it by its prefix "`multi`". It describes how the dictionary `transportProperties` should change during a run;
- `transportProperties` - this dictionary is automatically generated by **multiSolver**. Its content changes during a run (and during post-processing). Editing this file is only useful for runTime modification.

A multiDict has the following structure:

```
dictionaryName   fvSchemes;

multiSolver
```

```
{
    sovlerDomainName1 // this is the solverDomain name
    {
         // settings for the first solver go here
    }
    solverDomainName2 // another solverDomain name
    {
        // settings for the second solver go here
    }
    solverDomainName3 // etc..
    {
    }
    default // optional
    {
        // default settings go here
        // these are loaded first, then overwritten by solverName (above)
        // solvers whose names are not listed above inherit only these settings,
        //    or none at all if default is absent
    }
}
```

Sometimes, two or more solverDomains will have identical dictionaries. Rather than write out their settings several times, the `sameAs` keyword is available:

```
solverDomainName1
{
    // settings for the first solver go here
}
solverDomainName2
{
    sameAs   solverDomainName1;
}
```

Exception! The `multiControlDict` contains the settings for **multiSolver**. It is not a standard multiDict. See the next section for details.

4.1.4 MultiControlDict

The `multiControlDict` is the **multiSolver** analogue of the `controlDict`. The `controlDict` is auto-generated based on the content of this file. The `multiControlDict` is the main control dictionary and therefore it does not conform to the format of a regular multiDict. It is located in `case/system` and is used to automatically generate the `controlDict`.

Since **multiSolver** has so many optional settings, the full list of settings for the `multiSolverControlDict` is long, but most of these keywords are not required. The full list is shown below:

```
multiSolverControl
```

```
{
    initialStartFrom
        firstTime
        firstTimeInStartDomain // *1
        firstTimeInStartDomainInStartSuperLoop // *1 *2
        startTime // *3
        startTimeInStartDomain // *1 *3
        startTimeInStartDomainInStartSuperLoop // *1 *2 *3
        latestTime // (default)
        latestTimeInStartDomain // *1
        latestTimeInStartDomainInStartSuperLoop // *1 *2
    startTime // (required with *3)
    startDomain // (required with *1)
    startSuperLoop // (required with *2)

    finalStopAt
        endTime // (default) *4
        endTimeInEndDomain // *4 *5
        endTimeInEndDomainInEndSuperLoop // *4 *5 *6
        superLoopEnd // *6
        writeNow
        noWriteNow
        nextWrite
    endTime // (required with *4)
    endDomain // (required with *5)
    endSuperLoop // (required with *6)

    multiDictsRunTimeModifiable // (default true)
    timeFormat
    timePrecision
}

solverDomains
{
    solverDomainName1
    {
        startFrom
            firstTime
            startTime // *7
            latestTimeThisDomain
            latestTimeAllDomains // (default)
        startTime // (required with *7)
        stopAt
            endTime // *8 (default)
            writeNow
            noWriteNow
            nextWrite
            iterations // *9 (cannot be used with adjustableTimeStep)
            solverSignal
            elapsedTime // *10
        endTime // (required with *8; default 0)
        iterations // (required with *9)
        elapsedTime // (required with *10)
        storeField
        purgeWriteSuperLoops // (default 0)
// * The rest are the standard controlDict entries, i.e.:
//        writeControl
//            timeStep
//            runTime
//            adjustableRunTime
//            cpuTime
//            clockTime
//        writeInterval
//        purgeWrite
//        writeFormat
//        writePrecision
//        writeCompression
//        runTimeModifiable
//        graphFormat
//        deltaT
//        maxCo
```

```
//        adjustTimeStep
//        maxDeltaT
// * Anything else entered here will automatically be merged verbatim into the controlDict
    }
    solverDomainName2
    {
        timeValueStartFrom
        // etc..
    }
    default // (optional, but all prefixes must be defined)
    {
        // values here are loaded first, then overwritten by the solverDomainName
    }
}
```

multiSolverControl

The multiSolverControl subdictionary contains all the settings that affect the superSolver globally.

Initial start settings

- **initialStartFrom** - this setting determines where the superSolver reads the initial data from and begins its run at:
    - **firstTime** - load data from `case/multiSolver/currentSolverDomainName/0/0`
    - **firstTimeInStartDomain** - load data from `case/multiSolver/startDomain/0/0`
    - **firstTimeInStartDomainInStartSuperLoop** - load data from `case/multiSolver/startDomain/startSuperLoop/0`
    - **startTime** - search `case/multiSolver/[allSolverDomains]/[allSuperLoops]` for the closest *globalTime* to *startTime*; load this data
    - **startTimeInStartDomain** - search `case/multiSolver/startDomain/[allSuperLoops]` for the closest *localTime* to *startTime*; load this data

    - **startTimeInStartDomainInStartSuperLoop** - search `case/multiSolver/startDomain/startSuperLoop` for the closest *localTime* to *startTime*; load this data
    - **latestTime** - search `case/multiSolver/[allSolverDomains]/[allSuperLoops]` for the latest *globalTime*; load this data
    - **latestTimeInStartDomain** - search `case/multiSolver/startDomain/[allSuperLoops]` for the latest *localTime*; load this data
    - **latestTimeInStartDomainInStartSuperLoop** - search `case/multiSolver/startDomain/startSuperLoop` for the latest **localTime**; load this data
- **startTime** - only if required (see above)
- **startDomain** - only if required (see above)
- **startSuperLoop** - only if required (see above)

Final stop at settings

- **finalStopAt** - this setting determines when the **multiSolver** will stop the simulation
    - **endTime** - stop when *globalTime* reaches *endTime*
    - **endTimeInEndDomain** - stop when *localTime* reaches *endTime* in sovlerDomain *endDomain*
    - **endTimeInEndDomainInEndSuperLoop** - stop when *localTime* reaches *endTime* in sovlerDomain *endDomain* at superLoop *endSuperLoop*
    - **endSuperLoop** - stop after the superLoop number reaches *endSuperLoop*

    - **writeNow** - stop and write after the next solver starts
    - **noWriteNow** - stop without writing after the next solver starts
    - **nextWrite** - stop at the next designated write time after the next solver starts
- **endTime** - only if required (see above)
- **endDomain** - only if required (see above)
- **endSuperLoop** - only if required (see above)

Other global settings

- **multiDictsRunTimeModifiable** - when set to *on*, **multiSolver** will scan and reread any changed multiDicts
- **timeFormat** - *fixed*, *scientific*, or *general*, the same as in a regular `controlDict`. The timeFormat must be applied globally - there cannot be two solvers using different timeFormats.
- **timePrecision** - again, the same as in the regular **controlDict**.

Solver domains

The solverDomains subdictionary contains all the solverDomainNames as subdictionaries, and also can include a *default*, but *all* solverDomainNames must be included. These subdictionaries contain all the settings that apply *locally* to a single solverDomain.

Local start settings

- **startFrom** - this setting determines the localTime value to start from each time this solverDomain is initialized in each superLoop.
    - **firstTime** - start from localTime = 0
    - **startTime** - start from localTime = *startTime*

    - **latestTimeThisDomain** - start from the localTime it left off at when it was last in this solverDomain
    - **latestTimeAllDomains** - start from the latest globalTime (i.e. localTime and globalTime are equal for this solverDomain)
- **startTime** - only if required (see above)

Local stop settings

- **stopAt** - this setting determines where the solver will stop within each superLoop.
    - **endTime** - stop when localTime = *endTime*
    - **writeNow** - stop now and write out the results
    - **noWriteNow** - stop now without writing out results
    - **nextWrite** - stop at the next scheduled write time
    - **iterations** - stop after *endIterations* have been achieved. This setting cannot be used if *adjustableTimeStep* is enabled for this solverDomain
    - **solverSignal** - leave it up to the solver to give the stop signal. This essentially sets *endTime* to a very large number
    - **elapsedTime** - stop after elapsedTime has passed for this solverDomain
- **endTime** - only if required (see above)
- **iterations** - only if required (see above)
- **elapsedTime** - only if required (see above)

Other local settings

- **storeField** - a wordList of any fields that this solverDomain doesn't need. This saves the field from having to be carried in memory and written out at every write time, but also allows the

latest values of the field to be passed on to the next solverDomain. This works by copying the field to the last write time of the solverDomain in question.

- **purgeWriteSuperLoops** - this works the same as the standard *purgeWrite*, except instead of overwriting `[timeValue]` directories, `[superLoop]` directories are overwritten.

Any other values placed in the `solverDomainName` subdictionary will be merged verbatim into its `controlDict`.

Default solver domain

Any values placed in the default solverDomain will be loaded first, and written over by any values specific to a solverDomain. Although this is a **default** subdictionary, **all solverDomainNames must be present in the solverDomains subdictionary**.

4.1.5 Boundary conditions and initial values

The boundary conditions and initial values are located in:

```
case/multiSolver/[solverDomainName]/initial/0
```

This is the analogue of the `case/0` directory, except there is one for every solverDomain. Unlike in a regular simulation, **multiSolver** will always refer to the boundary conditions located in `initial/0`.

4.1.5.1 Changing boundary conditions

multiSolver allows the boundary conditions to change between solverDomains. To accomplish this, the latest internalField is combined with the boundaryField from `initial/0`.

4.1.6 Advanced boundary condition settings

I built in some functionality into **multiSolver** that I don't think is actually necessary. It basically allows you to have the updated boundary field values skip solverDomains. The feature is also untested. I'm commenting out the documentation for this section.

4.2 Store fields

Some solvers may not need to use all the fields created by other solvers. On the other hand, these other solvers need the latest values for these fields. There are two options for handling this:

1. add the unneeded fields to the `createFields.H` of the solver. The extra fields will be carried in memory, and written out at every time step.
2. declare these fields as storeFields in the `multiControlDict`. With this option, for the solver that doesn't need them, these fields will not be loaded in memory, and will be written only to the first timestep in each run.

Note: The *first* solverDomain to run must have all fields from all solvers defined in its `initial/0` directory.

4.3 Local time and Global time

A fundamental principle of **multiSolver** is that time is independent between solverDomains. (If this causes you apprehension, don't worry, the default behaviour uses a standard global time.) Therefore there are two defined time values:

- *localTime* - the value known to the solver; and
- *globalTime* - the universal time, known only by **multiSolver**.

4.3.1 Initial start

The `multiControlDict` has settings for an initial start defined gloablly (`initialStartAt`), and an initial start defined for each solverDomain (`startAt`). Sometimes these settings may appear to come into conflict. What happens when `initialStartAt` is set to `latestTimeAllDomains`, but `startAt` is set to `startTime = 0`?

The `initialStartAt` settings are where the initial data is read from. The `startAt` settings are where the local time value starts from. Sometimes you may be trying to resume from the middle of a run, and the initial time value should pick up from where it left off. **multiSolver** tries to determine when this is the case. The rules are:

- `startAt` time is equal to the *localTime* of the `initialStartAt` data source;
- if the `initialStartAt` is from a different solverDomain than the initial solverDomain, the `startAt` time specified in the `multiControlDict` is used instead.

4.3.2 Switching domains

When switching domains:

- *globalTime* stays the same (i.e. time does not step when switching domains); and
- *localTime* is set to the value specified by the `startAt` settings in the `multiControlDict`.

4.3.3 End time

The local `stopAt` settings are always compared with the global `finalStopAt` settings. If the `finalStopAt` value occurs before the local `stopAt` value, the `finalStopAt` value is used, and the end condition is set.

4.3.4 Initial superLoop

The initial superLoop value is determined by the `initialStartAt` settings. It is set to be equal to the superLoop value of the `initialStartAt` data source. If the `initialStartAt` data source is from a different solverDomain than the initial solverDomain, the next superLoop is used.

4.4 Runtime Modification

There are two levels of runtime modification: within a solverDomain, and globally.

- Editting a standard dictionary (e.g. `controlDict`, or `transportProperties` applies to a solverDomain. Its effect depends on whether that solver has `runTimeModifiable` enabled. This happens at the end of a solver iteration. However, these changes will be lost in the next superLoop when the same solverDomain is initialized.
- Editting a multiDict dictionary applies globally. This is goverened by `multiDictsRunTimeModifiable` setting in the `multiControlDict`, but these modifications do not take place until the next solverDomain is initialized. However, these changes are permanent.

5 Parallel

Parallel processing is now available with **multiSolver**.

5.1 decomposePar

To decompose the case directory:

1. Set it up as a usual multiSolver-enabled case directory;
2. Create a `system/decomposeParDict` file as you would with a regular parallel solver;
3. Instead of `decomposePar`, use:

```
multiSolver -preDecompose && decomposePar && multiSolver -postDecompose
```

5.2 reconstructPar

To reconstruct the case directory, instead of `reconstructPar`, use:

```
multiSolver -preReconstruct && reconstructPar && multiSolver -postReconstruct
```

5.3 Aliasing

If you are going to be doing this regularly, it might be a good idea to create a shorter alias for these two commands. To do this, add:

```
alias multiDecomposePar='multiSolver -preDecompose && decomposePar && multiSolver -
postDecompose'
alias multiReconstructPar='multiSolver -preReconstruct && reconstructPar && multiSolver -
postReconstruct'
```

To the end of your `OpenFOAM-version/etc/aliases.sh` or `OpenFOAM-version/etc/aliases.csh` file.

5.4 Parallel post processing

Apparently post processors are available that work with the data split across the processor directories / drives. **multiSolver** can be post processed in this way as well. To achieve this, use the commands as described on the post processing page, except run them in parallel.

For example, instead of:

```
multiSolver -load all
```

use:

```
mpirun -n 4 multiSolver -load all -parallel
```

substituting the correct options for mpirun. Then run your fancy parallel post processor.

6 Programming

6.1 Programming basics

Programming a multiSolver-enabled application is almost as simple as pasting two solvers together.

6.1.1 Simple example

Often a simple example is enough to get started. Here's a simple multiSolver-enabled application, or "superSolver":

```
/*---------------------------------------------------------------------------*\
                              ... STANDARD HEADER ...
\*---------------------------------------------------------------------------*/

#include "fvCFD.H"
#include "multiSolver.H"
```

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

int main(int argc, char *argv[])
{

#   include "setRootCase.H"
#   include "createMultiSolver.H"

// * * * * * * * * * * * * * * * * icoFoam  * * * * * * * * * * * * * * * * //

    Info << "*** Switching to icoFoam ***\n" << endl;
    solverDomain = "icoFoam";
#   include "setSolverDomain.H"

// Paste everything from icoFoam.C, starting with #include "createTime.H",
// and ending just before (but not including) return 0;

// * * * * * * * * * * * * * * scalarTransportFoam * * * * * * * * * * * * * * //

    Info << "*** Switching to scalarTransportFoam ***\n" << endl;
    solverDomain = "scalarTransportFoam";
#   include "setSolverDomain.H"

// Paste everything from scalarTransportFoam.C, again, starting with
// #include "createTime.H", and ending just before (but not including)
// return 0;

#   include "endMultiSolver.H"
    return(0);
}

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

6.1.2 Basic strategy

- Write (or choose existing) solvers that you intend to use with your superSolver;
- The `createFields.H` files (and associate #include statements) have to be renamed if they differ between solvers;
- Add `#include "multiSolver.H"` to the top of the solver;
- Just after `#include "setRootCase.H"`, add: `#include "createMultiSolver.H"`
- Between solvers use:

```
    solverDomain = "nextSolverDomain";
#   include "setSolverDomain.H"
```

- End with:

```
#include "endMultiSolver.H"
return (0);
```

6.2 Advanced concepts

If the basic framework described above doesn't suit your needs, read on. This section also covers some semantics that may be useful to know.

6.2.1 More on solverDomains

A solverDomain is an individual solver loop. It is assigned a name, and the list of names is static. A solverDomainName cannot be:

- `all`;
- `constant`;
- `default`; or
- `root`.

All solverDomains must appear in the `case/system/multiControlDict` file, although declaring additional names is not a problem.

6.2.2 Order of execution

In the simple example above, all the solverDomains execute in sequence, once per superLoop. This is not necessary: you can enclose them in conditionals; they can execute in any order; they can miss entire superLoops; however, they cannot execute more than once per superLoop. Use: `multiRun++` between solvers to force the superLoop number to increment if necessary.

Note: Using a `multiRun++` statement may lead to user-confusion with the `endSuperLoop` condition for `finalStopAt`.

6.2.3 runTime must go out of scope

Looking at the include files specified in the simple example, you will notice that the entire solver loop is enclosed in its own set of braces { }, starting before `#include "createTime.H"`. This is necessary

because runTime, the mesh, and all fields must go out of scope before **multiSolver** initializes another solverDomain.

6.2.4 End condition

multiSolver will detect the end condition automatically during the `setSolver` function. It will archive the last `case/time` directory, and exit the superLoop. This is achieved using the `#include` framework described in the simple example. If you are deviating from this framework, the requirements for correctly ending the **multiSolver** are:

- `setSolver` will automatically detect an end condition;
- to force an end condition, use `setFinished();`
- once the end condition is met, the function `setSolverDomain` must be encountered at least once more (although it may be encountered any number of times) to perform the final data clean-up;
- the function `multiRun()` returns true if another `setSolverDomain` still must be encountered; false means the run has finished, and `setSolverDomain` has completed the final clean-up;
- the inidivual solverDomain loops cannot be encountered after the final clean-up has taken place. Enclose them each in:

```
if (multiRun.run())
{
    // solverDomain loop
}
```

6.2.5 #undef directives

Sometimes there will be conflicts with `#define` directives across solverDomains. For instance, if you have more than one solver using `#include "createPhi.H"`, only the first solver will recognize it. This is caused by the fact that the `createPhi.H` file has this structure:

```
#ifndef createPhi_H
```

```
#define createPhi_H

// createPhi code

#endif
```

These preprocessor directives ensure the code for `createPhi` is read only once, regardless of how many times it is included. This prevents the compiler from complaining that something is being redefined. The problem is, when we switch solver domains, `phi` goes out of scope, and it is not recreated. To overcome this, use an `#undef` directive between solver domains: `#undef createPhi.H`

This, and a few other `#undef` directives are already included by default in the `setSolverDomain.H` file. You may encounter others that need to be undefined. Please let me know if you do, and I will add it to the `setSolverDomain.H` file. In the mean time, to add an `#undef` directive:

- using `#include "setSolverDomain.H"`:

```
    Info << "*** Switching to icoFoam ***\n" << endl;
#   undef [conflicting definition]
    solverDomain = "icoFoam";
#   include "setSolverDomain.H"
```

- without using `#include "setSolverDomain.H"`:

```
} // previous solver domain goes out of scope
multiRun.setSolverDomain(solverDomain);

#undef [conflicting definition]

if (multiRun.run())
{ // next solver domain comes into scope
```

where **`[conflicting definition]`** is the definition that needs to be removed.

6.3 How does it work?

OpenFOAM is incredibly flexible, and easily extensible, but implementing a change of this kind challenged its founding assumptions. Therefore, the flexibility was not there on level it needed to be, leaving little option but to use a top-level wrapper implementation.

A wrapper encloses the targeted object in a class that gives it the environment it expects to operate, while simultaneously presenting a different environment to other objects interfacing with it. At the top-level, the "other objects" are users. (Strictly speaking, at the code-level, multiSolver is not a true wrapper since it doesn't include an "OpenFOAM solver" as a member variable, but it is in principle.) **multiSolver** works by mutating the case directory into what each solver requires. A transient solver will see the correct `ddtSchemes` setting in `fvSchemes`; likewise a steady state solver will see `steadyState` for `ddtSchemes`. This is the purpose of the multiDict dictionary format.

The data output and input are hard-coded to the `case/[timeValue]` directory. Therefore, when **multiSolver** initializes the next solverDomain, it archives the existing output into the correct directory at `case/multiSolver/[solverDomainName]/[superLoopIndex]/[timeValue]`, and copies the latest field values to the initial time the next solver expects.

6.4 Reference

This reference section gives an overview of the functions available to you. You don't need to know any of this. It might be better just to look at the source.

6.4.1 Solver interface functions

Functions designed for use within a solver.

setSolverDomain

This function mutates the case directory into what the next solverDomain expects. It:

- rereads any modified multiDicts;
- archives the existing data to `case/multiSolver/solverDomain/superLoop/time`;

- copies the current field data to the `case/time/` directory, swapping the boundary conditions if necessary;
- creates and writes the new `controlDict`;
- swaps all multiDicts to the next solverDomain; and
- checks for the end condition.

setFinished

This function tells multiSolver that once the current solverDomain is finihsed, the full superSolver run is finished. (Technical, it tells multiSolver to save the last data and clean-up at the next `setSolverDomain()`, then the full superSolver run is finished.)

operator++

This function increments the superLoop number. It must be used if the same solverDomain is visited more than once in the same superLoop (otherwise it will overwrite its previous data). It can be used once between any pair of `setSolverDomain()` functions.

run and end

The `run()` and `end()` functions are analogous to those of the same name in the `Time` class. The first is true when the run should continue; the latter is true when the run should end.

6.4.2 Post processing functions

There are several functions designed for post-processing. Many of them depend on the use of the `timeCluster` and `timeClusterList` classes.

setSolverDomainPostProcessing

This mutates the case directory to that expected by the specified solverDomain. This is necessary for post-processors that read the controlDict.

timeFunctions

There are several searching / cataloging functions available for post-processing. Their function is described in the **multiSolver.H** header file. These include:

- **`findSuperLoops`** - list all the superLoop directories in a given path;
- **`findClosestGlobalTime`** - find the closest globalTime to a given value in a `timeClusterList`;
- **`findClosestLocalTime`** - find the closest localTime to a given value in a `timeClusterList`. If timeClusters are overlapping, this function only uses the those from the latest superLoop;
- **`findInstancePath`** - return the path to a given `timeCluster` and index;
- **`findMaxSuperLoopValue`** - maximum superLoop by value;
- **`findMaxSuperLoopIndices`** - return a labelList of the `timeClusters` with the maximum superLoop value;
- **`nonOverlapping`** - if the `timeClusters` overlap in time, return false. `timeClusters` that share starting points, or share ending points are non-overlapping;
- **`readSuperLoopTimes`** - catalog the time directories in `case/multiSolver/givenSolverDomain/givenSuperLoop`;
- **`readSolverDomainTimes`** - catalog the time directories in `case/multiSolver/givenSolverDomain/allSuperLoops`;

- **`readAllTimes`** - catalog the time directories in `case/multiSolver/allSolverDomains/allSuperLoops`;
- **`loadTimeClusterList`** - copy / move time directories in a `timeclusterList` to `case/time`;
- **`archiveTimeDirectories`** - copy / move time directories from sourcePath to destinationPath; and
- **`purgeTimeDirectories`** - delete all time directories in a given path.

6.4.3 Support classes

There are a few additional classes that were written to support **multiSolver**. These include:

- `tuple2List` class;
- `timeCluster` class;
- `timeClusterList` class; and
- `dummyControlDict` class.

tuple2List

This is a sortable list of paired values was created. It is sortable by first or second value, and currently can be any combination of `scalar` or `label`.

timeCluster

`timeCluster` is to **multiSolver** what `instant` is to runTime. This object holds all the information necessary to catalog the data within a single `solverDomain/superLoop` directory. It holds:

- the solverDomain name;
- the superLoop number;

- the globalTimeOffset; and
- an instantList, cataloging all the time directories within the directory.

Sometimes, a `timeCluster` is used to identify a single time directory within a solverDomain/superLoop. This is useful for functions such as: `findClosestGlobalTime`, which needs to identify a single time directory. To assist in this operation, `operator()` has been defined. It creates just such a `timeCluster`, given the index of the instant.

(This is a bit messy, it might have been smarter to create a different class to distinguish between single time directories, and time directory lists.)

timeClusterList

Again, similar to `instantList`, for **multiSolver**. A `timeClusterList` can catalog all the data in the case directory. Unlike `instantList`, this class has some functions of its own:

- **`append`** - add another `timeCluster` or `timeClusterList` to its collective;
- **`globalSort`** - sort its constituent `timeClusters` by their minimum globalTime;
- **`purgeEmpties`** - remove any `timeClusters` that have an empty `instantList`. Returns false if none remain. Many functions depend on a non-empty `instantList`.
- **`selectiveSubList`** - returns a `timeClusterList` that is composed of a subList of the original `timeClusterList`. The sublist is seleted by index using a labelList. The sublist is not a true sublist like in other classes; rather it is simply another timeClusterList.

Note: Use **`purgeEmpties`** to ensure there are no empty `timeClusters`. Many of the timeFunctions will throw a fatal error if passed an empty `timeClusterList`.

dummyControlDict

In order to allow runTimeModification of **multiSolver**'s multiDicts, **multiSolver** required an objectRegistry that doesn't dissappear between solverDomains, when runTime goes out of scope. Therefore it needed its own objectRegistry. Hence, `multiDictRegistry_` is a `Time` object. `Time` was never intended to be a member variable, therefore its constructors do not allow initialization without a `controlDict`. The object `dummyControlDict` was introduced as a self-initializing, minimal `controlDict`.

`dummycontrolDict` also has constructors that take a `multiControlDict`; or its name. These constructors look for the `timeFormat` and `timePrecision` keywords. If found, it includes these in its settings. These settings are static variables owned by `Time`; and to simplify implementation, it was made universal - i.e. they can only be set once (at initialization) in **multiSolver**.

Ultimately, the `dummyControlDict` was necessary for global runTimeModification.

7 Post processing

multiSolver has a post processing tool with the same name, which is required to make data available to the standard post processors.

7.1 Overview

OpenFOAM is hard-coded to look for data in the `case/[time]` directories. In order to post-process (including sampling, and data conversion) the data needs to be there. To accomplish this, a post processing utility is available. Typing the command `multiSolver` in the terminal accesses this. It has four main commands:

- `-load` - copy data files from their storage location to `case/time` (i.e. load the data for post-processing);
- `-purge` - delete data files;
- `-list` - display the contents found in the case directory storage location; and
- `-set` - make the case directory appear as required for the given solver name.

7.2 Syntax

```
multiSolver -command 'options' [-global] [-local] [-noPurge] [-noSet] [-noStore]
```

7.3 -command

Choose one of:

- **list** - list data available
- **load** - copy specified data to case/time
- **purge** - delete specified data
- **set** - change case directory to match supplied solver domain

7.3.1 list

Takes no options. Giving options will produce an error.

```
multiSolver -list
```

7.3.2 load

Loads the data specified in options. Additional options:

- `-global`, `-local` - By default, `load` will copy by *localTime*, unless the times overlap, in which case it loads by *globalTime*. These options force it to load by the specified time.
- `-noPurge` - By default, load will purge the `case/time` directories before copying the new data in. This option disables this behaviour.

- `-noSet` - By default, if all the specified load data is from a single solverDomain, load will automatically set the case directory to this solverDomain. This option disables this behaviour.
- `-noStore` - By default, if any *storeFields* exist, they will be copied into every time directory in which they are missing - this is for ease of post-processing. This option disables this copying, leaving only the data that actually exists in the `multiSolver` directory.

Load all data:

```
multiSolver -load all
```

Load all data from icoFoam solverDomain, using globalTime.

```
multiSolver -load icoFoam -global
```

Load all data from scalarTransportFoam, superLoops 1, 2, and 5 through 9, but do not delete the `case/time` directories first:

```
multiSolver -load 'scalarTransportFoam 1 2 5:9' -noPurge
```

7.3.3 purge

Purges the data specified in options.

Delete all the `case/time` directories:

```
multiSolver -purge root
```

Delete superloops 5 through 9 in all solverDomains:

```
multiSolver -purge '5:9'
```

Delete all data in all solverDomains (except for initial directories):

```
multiSolver -purge all
```

7.3.4 set

Sets the case directory to a given solverDomain. That is, all multiDicts are changed to the correct solverDomain; and a `controlDict` is written. The `controlDict` doesn't have all the data for the given solverDomain, but it (importantly) points to the first `case/time` directory as its `startFrom`. `paraFoam` uses this to initialize.

Set can only take a single solverDomain in options. Any superLoop specifications, or additional solverDomain names will result in error.

7.4 'options'

The options to the command. If the options have whitespace characters (i.e. are more than 1 word), they must be enclosed in apostrophes. Options come in two parts: solverDomains, followed by superLoops. Each can have any number of entries, including zero, but options cannot be empty unless using `-list`.

7.4.1 solverDomainNames

A simple word list, space delimited. List any solverDomains you want to load data from. e.g.:

```
scalarTransportFoam icoFoam customSolver
```

or

```
icoFoam
```

Omitting solverDomainNames indicates that *all* solverDomains will be used.

7.4.2 superLoop numbers

A space delimited number list in any order. Can include ranges using a : character. A value of `-1` indicates the initial directory. e.g.:

```
3 5 6 9:12
```

or

```
-1 5
```

Omitting superLoop numbers entirely indicates that *all* superLoops will be used.

7.4.3 Special words

There are two special words reserved for some of the commands:

- **`all`** - indicates all solverDomains and all superLoops. Useable by `-load` and `-purge`.

- **`root`** - indicates all case/time directories. Useable by -purge only.

e.g.:

```
multiSolver -load all
multiSolver -purge root
```

7.5 Examples

Show all available data:

```
multiSolver -list
```

Load all available data:

```
multiSolver -load all
```

Load all data from solverDomain icoFoam:

```
multiSolver -load icoFoam
```

Load all data from superLoops 8 and 9 from all solverDomains:

```
multiSolver -load '8 9'
```

Load all data from superLoops 4, 6, 7, 8, and 9 from solverDomain scalarTransportFoam, but do not delete any existing case/time directories.

```
multiSolver -load 'scalarTransportFoam 4 6:9' -noPurge
```

Load all data from superLoops 4, 5, 7, 8 and 9 from solverDomains icoFoam and scalarTransportFoam, but force the directory names to globalTime:

```
multiSolver -load 'icoFoam scalarTransportFoam 4 5 7:9' -global
```

Delete all case/time directories:

```
multiSolver -purge root
```

Delete all time directories in case/multiSolver/allSolverDomains/allSuperLoops, but do not delete any initial directories:

```
multiSolver -purge all
```

Delete superLoops 5, 6, 7, 8, and 9 from all solverDomains:

```
multiSolver -purge '5:9'
```

This instruction says to delete the initial directory from solverDomain icoFoam, but the initial directory is never deleted, therefore nothing is done:

```
multiSolver -purge 'icoFoam -1'
```

Set the case directory to scalarTransportFoam's settings:

```
multiSolver -set scalarTransportFoam
```