Shock-fitting Solver User-manual

September 14, 2015

Shock-fitting Solver is a modular shock fitting algorithm that can be coupled to arbitrary unstructured CFD codes. It is composed of several libraries dynamically linked. Their setting is handled by the user through a configuration file.

The code implements a `Fortran` software code of Paciorri and Bonfiglioli [3] in an *object oriented* environment.

The description of the algorithm can be found in [1], [2] and [3],while the description of the new `C++` architecture can be found in [1].

The original `Fortran` algorithm works with several flow topologies. Up-to-date the `C++` algorithm has been tested for the circular cylinder case.

1 Installation instructions

The installation instructions are the following:

1. Install `cmake` (version > 2.8) if not already installed in your system.
2. Install the `COOLFluiD` platform following the installation instructions available on-line at:

   `https://github.com/andrealani/COOLFluiD/wiki/HOWTO`

3. Download the Shock-fitting Solver installation script from the website[1]:

   `https://github.com/andrealani/ShockFitting/wiki`

4. Create a directory `build` inside your `CouplingTools` home:

   `mkdir build`

5. Move into the directory:

   `cd build`

[1]use the link on the left-hand side

6. Configure by running the command:

```
cmake .. - DMPI_HOME=MPIDIR - DCMAKE_CXX_COMPILER=CXX - DCMAKE_INSTALL_
 PREFIX=INSTALLDIR - DCF_BUILD_Framework_API=ON - DCMAKE_BUILD_TYPE=DEBUG
```

   where:
   `CXX` : chosen `C++` compiler
   `MPIDIR` : directory of existing `MPI` installation
   `INSTALLDIR` : directory where CouplingTools libraries will be installed

7. Compile the libraries through the command

```
make install
```

   Upon successful completion, all shared libraries and include files (from the Framework only) can be found respectively inside:

```
INSTALLDIR/lib   INSTALLDIR/include/couplingtools/Framework
```

2 How to set up a test case

In order to setup a test case the following files are necessary:

1. shock-fitting files (section 2.1)
2. shock-fitting configuration file (*input.case*, section 2.2)
3. `COOLFluiD` file(s) (section 2.3)

2.1 Shock-fitting files

The required informations to initialize the shock-fitting algorithm are stored in the following files:

- mesh data files. Here the informations on the background grid are stored in terms of geometry and state. Up-to-date the `triangle mesh generator` format is used. The mesh informations are stored in five formats: *.node*, *.poly*, *.ele*, *.neigh*, *.edge*[2]. Each file format is described in [4].
- shock(s) data file: *sh00.dat*. It contains the the coordinates, the downstream and upstream states for each discontinuity(s) point:

  <# `discontinuities`>

[2]only the *.node* and *.poly* files are actually necessary. Starting from them, the other ones can be created by running `triangle` with the `-nep` switch.

then, for each discontinuity:

`<# points > < type of discontinuity (`S` or `D`)>`

for each discontinuity point (`NB_DIM`(=2)+`NB_EQ` entries per line):

`<x> <y>` $< Z_1^* >$ $< Z_2^* >$ $< Z_3^* >$ ... $< Z_{nb_{eq}}^* >$

where $Z_{1,\ldots,nb_{eq}}^*$ represent the non-dimensional Roe parameter vector variables.

The special points[3] are finally listed at the bottom of the file:

`<# special points>`

then, for each special point:

`<type special point>`

`<# discontinuity it belongs to> < # edge of the discontinuity it belongs to>`

WATCH OUT: up-to-date the `OP` and `IP` special points have been tested.

2.1.1 How to create the shock-fitting files

By counting out simple geometries, the files required to initialize the shock-fitting algorithm cannot be manually generated. They have to be created by starting from *captured* solutions.

The creation of the mesh data files is straightforward by starting from the file storing the CFD solution. It can be made by specifying the `CFmesh2StartingTriangle` object inside the shock-fitting configuration file, as described in section 2.2.9.

The creation of the discontinuities file can be made by manually extracting the discontinuity profile or by using a detection algorithm. Up-to-date the both two techniques have been tested when single bow shock appears within the domain.

- Manual extraction of discontinuity: the shock profile is extracted by loading the *.plt* captured solution with `Tecplot`. Then, by using the option:

  `Data/Extract/Points From Polyline`

  the shock profile is extracted by tracing a polyline. Finally, the data have to

[3] the *special* points represent intersection points between the shock edges and the domain boundaries or the shock edges and other discontinuities.

be written in an output file by specifying the shock points coordinates and the only variables corresponding to the `NB_EQs` used (for the *perfect gas* instance, the only `p u v T` variables have to be written; for the *thermochemical non-equilibrium* instance, the only $\rho_1 \;\; \rho_2 \;\; \ldots \;\; \rho_{nb_{species}}$ `u v T` T_{v_0} variables have to be written).

In order to create the shock-fitting discontinuity file, the `ShockFileConverter` object must be specified in the configuration file, as described in 2.2.9.

- Automatic detection of discontinuity: the shock is detected by using a detection algorithm implemented inside the *Shock Fitting Solver*. The `ShockDetector` object must be specified inside the shock-fitting configuration file. See section 2.2.8 for the details.

2.2 Shock-fitting configuration file

A configuration file, named as *input.case*, is used to state the objects assembling the shock-fitting algorithm and specify the main features of each shock-fitting simulation.

The configuration file is composed by several lines.
Each line is in the form `KEY = VALUE`. The `KEY` is an object or an object parameter and the `VALUE` is the quantity assigned to `KEY`.

The `VALUE` can be:

- an alpha-numeric string
- an integer
- a boolean (*true* or *false*)
- a floating point number
- an arbitrary complex analytical function
- an array of all the previous

The `VALUE`s can be broken in different lines by using the character backslash. Comments start with "#".

2.2.1 Model

```
.ShockFittingObj = StandardShockFitting
```

specifies the model of the Shock-fitting Solver and corresponds to a set of objects defined inside the code.
Up-to-date the `StandardShockFitting` and `StandardShockFittingBeta` models are defined.
The `StandardShockFittingBeta` has been created in order to use and test the shock detection feature. It uses the same set of objects of the `StandardShockFitting` except for the `ShockDetector` library that is called in place of the `ShockFileConverter` object.

2.2.2 Model setting

```
.StandardShockFitting = original
```

allows to choose the between different versions (if available) of the chosen `Model`. Up-to-date the `original` version (the `triangle mesh generator` library is called as executable files. The data are passed to it through `I/O` files.) and the `optimized` version (the `triangle mesh generator` library is called through it functions. The data are passed to it through arrays.) are implemented.

```
.StandardShockFitting.ResultsDir = ./Results_SF
```

specifies the path of the output files generated during the execution of the shock-fitting.

```
.StandardShockFitting.ComputeResidual = true
```

specifies if the shock-fitting residuals are computed during the execution. If `true`, the `ComputeResidual` object must be added to the `StateUpdaterSF` library list of section 2.2.12.

```
.StandardShockFitting.startFromCapturedFiles = true
```

specifies if the shock-fitting files have to be generated from a CFD solution (`true`) or if they are already available (`false`). If the `true` option is used, the `CFmesh2StartingTriangle` (section 2.2.9) and the object creating the discontinuity file must be specified (section 2.2.9 or section 2.2.8).

2.2.3 MeshData

```
.StandardShockFitting.MeshData.EPS = 0.20e-12
.StandardShockFitting.MeshData.SNDMIN = 0.05
.StandardShockFitting.MeshData.DXCELL = 0.0006
.StandardShockFitting.MeshData.SHRELAX = 0.9
```

define the distance between two shock faces, the maximum non-dimensional distance of phantom nodes, the length of the shock edges, the relax coefficient of shock points integration.

```
.StandardShockFitting.MeshData.Naddholes = 0
```

defines the number of hole points.

```
.StandardShockFitting.MeshData.CADDholes = 0
```

defines the coordinates of the hole points specified above.

```
.StandardShockFitting.MeshData.freezedWallCells = true
```

specifies if the connectivity of the wall cells must be freezed. This option is usually used for circular cylinder in viscous flows and requires specific converters with `Freez` options (see section 2.2.9).

```
.StandardShockFitting.MeshData.WithP0 = true
```

is used for backward compatibility. Choose `true` for the `2013.9 COOLFluiD` version and `false` for the `2014.11` one or higher.

```
.StandardShockFitting.MeshData.NPROC = 4
```

defines the number of processor used during the `COOLFluiD` execution.
With `NPROC = 1` it will be executed sequentially, with `NPROC = 2` or more, it will be executed in parallel.

```
.StandardShockFitting.MeshData.NBegin = 0
```

specifies the number of the first step. If `NBegin = 0` is chosen, the steps numbering will start from `0`.

```
.StandardShockFitting.MeshData.NSteps = 1000
```

specifies the maximum number of steps.

```
.StandardShockFitting.MeshData.IBAK = 100
```

defines every how many steps the solution will be saved. The files are saved inside directories named as *step* and the number of the current step (*e.g.*: step number 101 will be saved in the folder named as `step00101`).

2.2.4 PhysicsData

PhysicsInfo

```
.StandardShockFitting.PhysicsData.PhysicsInfo.NDIM = 2
.StandardShockFitting.PhysicsData.PhysicsInfo.NDOFMAX = 6
.StandardShockFitting.PhysicsData.PhysicsInfo.NSHMAX = 5
.StandardShockFitting.PhysicsData.PhysicsInfo.NPSHMAX = 1000
.StandardShockFitting.PhysicsData.PhysicsInfo.NESHMAX = 999
.StandardShockFitting.PhysicsData.PhysicsInfo.NADDHOLESMAX = 10
```

```
.StandardShockFitting.PhysicsData.PhysicsInfo.NSPMAX = 12
```

specify the space dimension, the maximum number of degrees of freedom, the maximum number of shocks, the maximum number of shock points for each shock, the maximum number of shock edges for each shocks[4], the maximum number of holes, the maximum number of special points.
At the first attempt these options are mostly stable and should be not be changed.

```
.StandardShockFitting.PhysicsData.PhysicsInfo.GAM = 1.40e0
```

defines the value of the free-stream heat capacity ratio [5].

ChemicalInfo

```
.StandardShockFitting.PhysicsData.ChemicalInfo.MODEL = TCneq
```

specifies the gas model. Up-to-date the `PG` (*Perfet Gas*), `Cneq` (*Chemical non equilibrium* with argon mixture) and `TCneq` (*Thermo-chemical non-equilibrium*) are implemented.

```
.StandardShockFitting.PhysicsData.ChemicalInfo.IE = 0
.StandardShockFitting.PhysicsData.ChemicalInfo.IX = 1
.StandardShockFitting.PhysicsData.ChemicalInfo.IY = 2
.StandardShockFitting.PhysicsData.ChemicalInfo.IEV = 3
```

Those options are most stable and should not be changed.
When `TCneq` model is chosen, the following options must be specified:

```
.StandardShockFitting.PhysicsData.ChemicalInfo.MIXTURE = nitrogen2
.StandardShockFitting.PhysicsData.ChemicalInfo.InputFiles = nitrogen2.dat
```

define the name of the the gas mixture and the file containing the gas mixture informations.
The mixture file template is shown hereafter:

```
!NAME           (name of the mixture)
!NSP            (number of the chemical species)
!SPECIES        (name of the species - IUPAC)
!MM             (molecular weight of the species [kg/mol])
```

[4]this values must always set equal to `NPSHMAX-1`

[5]this value is actually used only for the `PG` (*Perfect Gas*) and `Cneq` (*Chemical non equilibrium*) gas models.

```
!HF                 (formation enthalpy at 0 K of the species [J/kg])
!THEV               (characteristic vibrational temperature [K])
!GAMS               (specific heat ratio of each species)
!TYPE               (type of molecule:
                            A: atomic
                            B: di-atomic or aligned
                            T: tri-atomic non aligned)
```

some examples can be found inside the folder `src/data_template`.
When `Cneq` model is chosen, the following option must be specified:

```
.StandardShockFitting.PhysicsData.ChemicalInfo.Qref = 1.0
```

it defines the reference speed.

ReferenceInfo

```
.StandardShockFitting.PhysicsData.ReferenceInfo.gamma = 1.4
.StandardShockFitting.PhysicsData.ReferenceInfo.Rgas = 287.0e0
.StandardShockFitting.PhysicsData.ReferenceInfo.TempRef = 1833.0e0
.StandardShockFitting.PhysicsData.ReferenceInfo.PressRef = 57.65e0
.StandardShockFitting.PhysicsData.ReferenceInfo.VelocityRef = 5594.0e0
.StandardShockFitting.PhysicsData.ReferenceInfo.Lref = 1.0e0
```

are used by the `VariableTransformerSF` library.
Those options define the gas heat capacity ratio, the gas constant, the free-stream temperature, the free-stream pressure, the free-stream speed and the reference length. If `TCneq` and `Cneq` models are used, the species densities must be specified:

```
.StandardShockFitting.PhysicsData.ReferenceInfo.SpeciesDensities = \
  0.00036354 0.00461646
```

2.2.5 MeshGeneratorSF

```
.StandardShockFitting.MeshGeneratorList = ReadTriangle ReSdwInfo \
                                           TriangleExe Tricall
```

specifies the objects belonging to `MeshGeneratorSF` library that are called in the run model of the Shock-fitting Solver.

```
.StandardShockFitting.ReadTriangle = na00.1
.StandardShockFitting.ReadTriangle.FileTypes = node poly ele neigh edge
```

indicate the name and the formats of the shock-fitting mesh data files.

```
.StandardShockFitting.ReSdwInfo.InputFiles = sh00.dat
```

specifies the name of the discontinuity file.

If `StandardShockFittingBeta` is chosen in order to use the shock detection feature, the following option must be added:

```
.StandardShockFittingBeta.ReadTriangle.BCtypes = Wall Inlet Outlet
```

it specifies the strings assigned to the domain boundaries. They must be listed according to the boundary markers assigned inside the *.poly* file. The first string corresponds to the boundaries having the boundarymarker=1, the second string corresponds to the boundaries having the boundarymarker=2 and so on.

WATCH OUT: if the `freezedWallcell` option is active (section 2.2.2), the `ReadTriangleFreez` object must be used in place of `ReadTriangle`.

2.2.6 RemeshingSF

```
.StandardShockFitting.RemeshingList = \
 BndryNodePtr RdDpsEq FndPhPs ChangeBndryPtr \
 CoNorm4Pg CoPntDispl FixMshSps RdDps
```

specifies the objects of the `RemeshingSF` library called in the run model of the Shock-fitting Solver. The `CoNorm` object must be defined according to the gas model: `Pg` or `Cneq` or `TCneq` should be added to the string `CoNorm4`.

WATCH OUT: if the `freezedWallcell` option is active, the `BndryNodePtrFreez` object must be used in place of `BndryNodePtr` and the `BndryFacePtrFreez` object must be added at the end of the list:

```
.StandardShockFitting.RemeshingList = \
 BndryNodePtrFreez RdDpsEq FndPhPs ChangeBndryPtr CoNorm4Pg \
 CoPntDispl FixMshSps RdDps BndryFacePtrFreez
```

2.2.7 WritingMeshSF

```
.StandardShockFitting.WritingMeshList = \
 WriteTriangle WriteBackTriangle WriteSdwInfo
```

specifies the objects of the `WritingMeshSF` library called in the current model of the Shock-fitting Solver.

WATCH OUT: if the `freezedWallcell` option is active, the `WriteTriangleFreez` object must be used in place of `WriteTriangle`.

2.2.8 ShockDetectorSF

This library must be considered *only if* the automatic shock detection is chosen to extract the shock polyline. It means that the `StandardShockFittingBeta` has been chosen as shock-fitting model (section 2.2.1) and the `startFromCapturedFiles` option has been actived (section 2.2.2).

```
.StandardShockFittingBeta.ShockDetectorList = DetectorAlgorithm
```

specifies the objects of the `ShockDetector` library called in the run model of the Shock-fitting Solver.

```
.StandardShockFittingBeta.DetectorAlgorithm.From = Param
.StandardShockFittingBeta.DetectorAlgorithm.To = Prim
.StandardShockFittingBeta.DetectorAlgorithm.GasModel = Pg
.StandardShockFittingBeta.DetectorAlgorithm.AdditionalInfo = Dimensional
```

In order to better understand the following options, see chapter 5 of [**?**].

```
.StandardShockFittingBeta.DetectorAlgorithm.Detector = GnoffoShockSensor
```

specifies the detector method to extract the shock points from the CFD solution. Up-to-date the `GnoffoShockSensor` and the `NormalMachNumber` are implemented. The `GnoffoShockSensor` is the one best works with strong shocks.

Three techniques are implemented to fit the shock point distribution extracted by the detector methods: `Ellipse`, `Polynomial`, `SplittingCurves`.

```
.StandardShockFittingBeta.DetectorAlgorithm.fittingTechnique = Ellipse
```

if `Ellipse` is chosen, no additional options must be specified.

```
      .StandardShockFittingBeta.DetectorAlgorithm.fittingTechnique = Polynomial
.StandardShockFittingBeta.DetectorAlgorithm.polynomialOrder = 2
```

if `Polynomial` is chosen, the polynomial order must be specified.

```
.StandardShockFittingBeta.DetectorAlgorithm.fittingTechnique = SplittingCurves
.StandardShockFittingBeta.DetectorAlgorithm.nbXandYsegments = 1 2
.StandardShockFittingBeta.DetectorAlgorithm.segmPolynomialOrders = \
 5 5
```

if the SplittingCurves technique is chosen, the number of segments along the x-axis and the y-axis in addition to the order of the polynomial assigned to each segment must be specified.
Irrespective of the chosen fitting technique, the folloqing options must be specified:

```
.StandardShockFittingBeta.DetectorAlgorithm.smoothingOption = true
```

if it is active (true), it smooths the trend of the polyline.

```
.StandardShockFittingBeta.DetectorAlgorithm.shockLayerFactor = 1.5
```

specifies the distance used to extract the upstream and downstream points. The shockLayerFactor will be multiplied by the DXCELL value.

2.2.9 ConverterSF

```
.StandardShockFitting.ConverterList = \
 ShockFileConverter CFmesh2StartingTriangle Triangle2CFmesh CFmesh2Triangle
```

specifies the objects of the ConverterSF library called in the run model of the Shock-fitting Solver.
For each converter object in the list, the following lines must be added (in the example below are related to the CFmesh2Triangle object):

```
.StandardShockFitting.CFmesh2Triangle.From = Prim
.StandardShockFitting.CFmesh2Triangle.To = Param
.StandardShockFitting.CFmesh2Triangle.GasModel = TCneq
.StandardShockFitting.CFmesh2Triangle.AdditionalInfo = Dimensional
```

They define the strings that will create the name of the VariableTrasformerSF object asked to make the variables transformation.

Up-to-date the From and the To options have Prim and Param as possible values.
The GasModel can be Pg or Cneq or TCneq.
The AdditionalInfo specifies the CFD variables format (Dimensional or Adimensional).

If the startFromCapturedFiles option is active (section 2.2.2) and the manual extraction of the shock polyline is used (therefore the StandardShockFitting is chosen as model) some additional options must be specified in the definition of the ShockFileConverter.

```
.StandardShockFitting.ShockFileConverter.InputFile = FILE_PATH/shock.dat
```

defines the name of the tecplot file containing the shock points polyline and the corresponding informations.

```
.StandardShockFitting.ShockFileConverter.nbDof = 6
.StandardShockFitting.ShockFileConverter.nbShocks = 1
.StandardShockFitting.ShockFileConverter.nbSpecPoints = 2
.StandardShockFitting.ShockFileConverter.TypeSpecPoints = OPY
```

specify the options needed for the *sh00.dat* file creation: the number of degrees of freedom, the number of shocks, the number of special points, the type of the special points. Up-to-date only `OPY` are implemented as special points.

If the `startFromCapturedFiles` option is active (section 2.2.2), the `CFmesh2StartingTriangle` is used to create the *triangle* files from the captured solution. This additional line must be added in addtion to the options specified at the beginning of the section:

```
.StandardShockFitting.CFmesh2StartingTriangle.InputFile = FILE_PATH/start.CFmesh
```

specifies the name of the `COOLFluiD` file storing the captured solution.

If the `Triangle2CFmesh` object is used, an additional info must be specified:

```
.StandardShockFitting.Triangle2CFmesh.ShockBoundary = single
```

states if the shock boundary is *single* or it is *splitted* in a *subsonic* and a *supersonic* edges.

<u>*WATCH OUT*</u>: if the `freezedWallcell` option is set to `true`, `Triangle2CFmeshFreez` and `CFmesh2TriangleFreez` must be used in place of `Triangle2CFmesh` and `CFmesh2Triangle`.

Converters from Tecplot format to `triangle` format are defined inside the code. When using the Residual Distribution Methods, the `Triangle2CFmesh` and `CFmesh2Triangle` converters can be replaced with `Triangle2Tecplot` and `Tecplot2Triangle`.

<u>*WATCH OUT*</u>: when using the Finite Volume Method the converters must be `Triangle2Tecplot`, `TecplotFVM2StartingTriangle` and `TecplotFVM2Triangle`.

2.2.10 CFDSolverSF

```
.StandardShockFittingBeta.CFDSolverList = COOLFluiD
```

specifies the CFD solver called during the execution of the shock-fitting. Up-to-date the `COOLFluiD` solver can be used.

2.2.11 CopyMakerSF

```
.StandardShockFitting.CopyMakerList = \
 MeshBackup CopyRoeValues1 CopyRoeValues2 MeshRestoring
```

specifies the objects of the `CopyMakerSF` library called in the run model of the Shock-fitting Solver.

2.2.12 StateUpdaterSF

```
.StandardShockFitting.StateUpdaterList = \
 ComputeStateDps4Pg FixStateSps MoveDps4Pg Interp ComputeResidual
```

specifies the objects of the `StateUpdaterSF` library called in the run model of the Shock-fitting Solver.
The `ComputeStateDps` object must be defined according to the gas model: `Pg` or `Cneq` or `TCneq` should be added to the string `ComputeStateDps4`. Similarly for `MoveDps` object. The `ComputeResidual` object must be added only if the `ComputeResidual` option is active (section 2.2.2). If it is the case, some additional options must be specified:

```
.StandardShockFitting.ComputeResidual.whichNorm = L1
```

defines the norm of the discretization error used to compute the residual. Up-to-date the `L1` and `L2` norms are implemented.

```
.StandardShockFitting.ComputeResidual.isItWeighted = true
```

specifies if the norm is weighted on the first residual value.

```
.StandardShockFitting.ComputeResidual.gasModel = Pg
```

sets the gas model (`Pg` or `TCneq`) used to make the conversion to primitive variables.

2.3 The *COOLFluiD* input files

The files requested to run `COOLFluiD` are the following:

1. `.CFmesh` file, it is automatically generated during the Shock-fitting Solver execution
2. `.CFcase` configuration file
3. `coolfluid-solver.xml` file containing the link to the libraries

The `.CFcase` file description is available at:

```
https://github.com/andrealani/COOLFluiD/wiki
```

in the `HOWTO define a test case` section.

2.4 How to run

Once all the required files are collected, go to the folder containing all the required files and create a soft link to the executable of the shock fitting solver by writing the command:

```
ln -sf build/PATH_TO_THE_EXECUTABLE_FILE/EXECUTABLE_FILE_NAME .
```

run the test case through the command:

```
./EXECUTABLE_FILE_NAME input.case
```

For the `StandardShockFittingSF` instance, the two commands will be the following:

```
ln -sf build/src/TestStandardSF/TestStandardSF .

./TestStandardSF input.case
```

while, for the `StandardShockFittingBetaSF` instance, the command will be:

```
ln -sf build/src/TestStandardBetaSF/TestStandardBetaSF .

./TestStandardBetaSF input.case
```

The sections hereafter are aimed to the user intending to act on the code modifying it, adding new features and keying it to its requirements.

3 Common libraries issues

Each library has a corresponding vector defined inside the code. This vector is used to handle the objects of the library. Each library vector has the following notation:

```
std::vector<SConfig::StringT<SConfig::SharedPtr<LIBRARY_BASECLASS_NAME>>
                                               m_LIBRARY_VECTOR_NAME;
```

The library vector includes the pointers to the library objects as listed in the *input.case*. The library objects execution is handled through the object pointers and not through the objects them self. This allows to obtain the dynamic nature of the software. The pointed objects are defined each time by the user inside the *input.case*. The pointers act

inside the code on objects place.
The called objects are defined through the `List` option inside the *input.case*:

```
.StandardShockFitting.LIBRARY_BASECLASS_NAMEList = LIBRARYOBJECT1 LIBRARYOBJECT2
```

When a component is added to `List`, a pointer must be defined:

```
SConfig::SharedPtr<LIBRARY_BASECLASS_NAME> m_OBJECT_POINTER_NAME;
```

and it must be assigned to a position inside the library vector:

```
m_OBJECT_POINTER_NAME = m_LIBRARY_NAME[POSITION].ptr();
```

where `POSITION` is the place that the addressed `LIBRARYOBJECT` occupies in the `LIBRARY_NAMEList` of the *input.case*.
The pointer is then used on object's place:

```
m_OBJECT_POINTER_NAME→LIBRARY_MAINFUNCTION();
```

If the `LIBRARYOBJECT1` have be executed, it is enough to assign the position `0` to the pointer:

```
m_LIBRARY_OBJECT_POINTER = m_LIBRARY_NAME[0].ptr();
```

if the `LIBRARYOBJECT2` have to be executed instead, the position `1` must be assigned to the pointer:

```
m_LIBRARY_OBJECT_POINTER = m_LIBRARY_NAME[1].ptr();
```

however the execution command remains the same for the two objects, as written before:

```
m_LIBRARY_OBJECT_POINTER→LIBRARY_MAINFUNCTION();
```

Let us see an example using the `MeshGenratorSF` library. The scheme of the library is shown in Fig. 1.

The library vector can be defined as follows:

```
std::vector<SConfig::StringT<SConfig::SharedPtr<MeshGenerator>> m_mGenerator;
```

Inside the *input.case* the `List` option is the following:

```
.StandardShockFitting.MeshGneeratorList = ReadTriangle ReSdwInfo Tricall
```

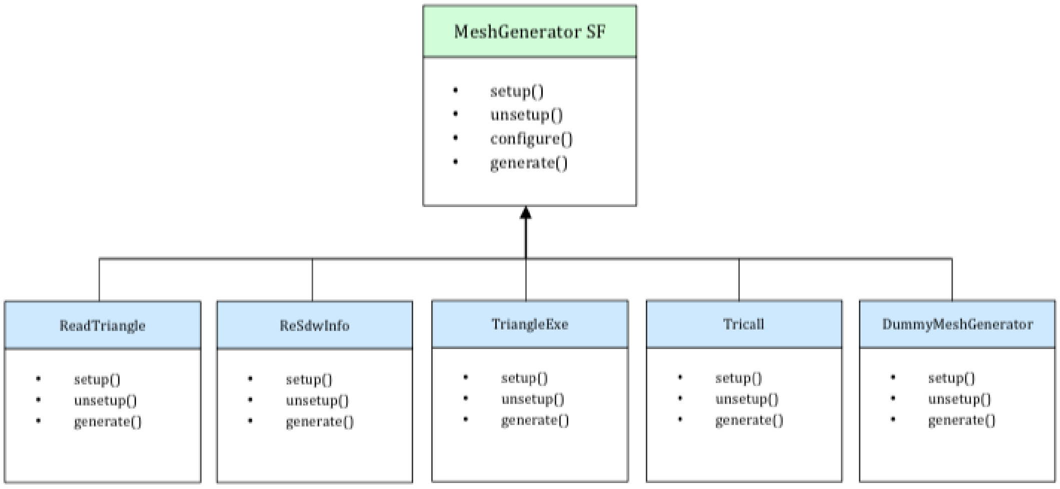

Figure 1: `MeshGenerator SF` library. The base object and its derived classes.

The pointer to the first object of the `List` is instantiated through the command:

```
SConfig::SharedPtr<MeshGenerator>m_readTriangle;
```

and it is assigned to the first entity of the `MeshGeneratorList`:

```
m_readTriangle = m_mGenerator[0].ptr();
```

and it is executed:

```
m_readTriangle→generate();
```

In the next section the creation of a new component inside an existing library is explained.

4 How to add a new component to an existing library

Each library has an own directory. The new member has to be placed in the folder of the library it belongs to.
The creation of a new library member is straightforward by following the "`Dummy`" template.
In each library's directory there is a *dummy* component (e.g: the `MeshGeneratorSF` library contains the `DummyMeshGenerator` member, the `RemeshingSF` library contains the `DummyRemeshing` and so on). By following the structure of the `Dummy` component, a new object can be easily created.

WATCH OUT: add always the new component to the *CMakeLists.txt* file of the library directory.
Once that the new component is created, it must be linked to the overall framework. These three steps can be follow:

1. define a new library pointer inside the `StandardShockFitting.hh` file. The new pointer must be related to the *base* class of the library:

   ```
   SConfig::SharedPtr<LIBRARY_BASECLASS_NAME> m_OBJECT_POINTER_NAME;
   ```

2. Inside the `StandardShockFitting.cxx` file, assign the pointer to a position of the library's vector, according to the *input.case* `List`:

   ```
   m_OBJECT_POINTER_NAME = m_LIBRARY_NAME.[POSITION].ptr();
   ```

3. use the pointer on object's place:

   ```
   m_OBJECT_POINTER_NAME→MAINFUNCTION();
   ```

In the next section the creation of a new library is explained.

5 How to add a new library

5.1 Creating a new library

As already explained in chapter 5 of [1], each library has a main component, the *base* class, and several members, the *derived* classes. Each *base* class has the `setup` and `unsetup` methods and a function identifying its main purpose. The members of the library inherits and customizes the main function according to their personal task.
Let us suppose to need a new library, e.g `DummyLibrarySF`. The main goal of this library is `create`.
The steps toward the creation of the library are the following:

1. insert the description of the *base* class in the `Framework` folder. Here all the *base* classes of the library with their functions are defined.

   WATCH OUT: add always the new members to the *CMakeLists.cxx* file of the `Framework` directory.

   In List 2 the `.hh` file is shown, while is List 3 the `.cxx` file is represented.

2. create a library directory in the `src` folder. The directory should be named as `DummyLibrarySF`. Here all the *derived* classes can be stored.

3. create the *CMakeLists.txt* file as appeared below:

   ```
   LIST (APPEND DummyLibrarySF_files
   ```

```
#ifndef ShockFitting_DummyLibrary_hh
#define ShockFitting_DummyLibrary_hh

#include "Framework/BaseShockFitting.hh"
#include "Framework/Field.hh"
#include "SConfig/SharedPtr.hh"

namespace ShockFitting {

class DummyLibrary : public BaseShockFitting {
public:

  /// typedef needed by the self-registration mechanism
  typedef SConfig::Provider<DummyLibrary> PROVIDER;

  /// Constructor
  // @param objectName the concrete class name
  DummyLibrary(const std::string& objectName);

  /// Destructor
  virtual ~DummyLibrary();

  /// Set up this object before its first use
  virtual void setup() = 0;

  /// Unset up this object after its last use
  virtual void unsetup() = 0;

  /// Configure the options for this object.
  /// To be extended by derived classes.
  /// @param args is the ConfgArgs with the arguments to be parsed.
  virtual void configure(SConfig::OptionMap& cmap, const std::string& prefix);

  /// create something
  virtual void create() = 0;

  /// Gets the Class name
  static std::string getClassName() {return "DummyLibrary";}

};

} // namespace ShockFitting

#endif // ShockFitting_DummyLibrary_hh
```

Figure 2: List showing `DummyLibrary.hh` file.

```
#include "Framework/DummyLibrary.hh"
#include "Framework/Log.hh"

//----------------------------------------------------------------------------//

using namespace std;
using namespace SConfig;

//----------------------------------------------------------------------------//

namespace ShockFitting{

//----------------------------------------------------------------------------//

DummyLibrary::DummyLibrary(const std::string& objectName) :
  BaseShockFitting(objectName)
{
  m_file = "dummyFile";
  addOption("FileToBeCreated",&m_file,
            "List of the creating files");
}

//----------------------------------------------------------------------------//

DummyLibrary::~DummyLibrary()
{
}

//----------------------------------------------------------------------------//

void DummyLibrary::configure(OptionMap& cmap, const std::string& prefix)
{
  LogToScreen(VERBOSE, "DummyLibrary::configure() => start\n");

  BaseShockFitting::configure(cmap, prefix);

  LogToScreen(VERBOSE, "DummyLibrary::configure() => end\n");
}

//----------------------------------------------------------------------------//

} // namespace ShockFitting
```

Figure 3: List showing `DummyLibrary.cxx` file.

```
DummyComponent.cxx
DummyComponent.hh

LIST (APPEND DummyLibrarySF_libs Framework SConfig MathTools)

SF_ADD_PLUGIN_LIBRARY (DummyLibrarySF)

#SF_WARN_ORPHAN_FILES()
```

After "LIST" all the library components (*e.g*: `DummyComponent`) come in succession.

4. start to create the libraries component beginning with the `Dummy` one. An example of a `DummyComponent .hh` and `.cxx` files is shown in List 4 and List 5.

```cpp
#ifndef ShockFitting_DummyComponent_hh
#define ShockFitting_DummyComponent_hh

#include "Framework/DummyLibrary.hh"

namespace ShockFitting {

class DummyComponent : public DummyLibrary {
public:

  /// Constructor
  /// @param objectName the concrete class name
  DummyComponent(const std::string& objectName);

  /// Destructor
  virtual ~DummyComponent();

  /// Set up this object before its first use
  virtual void setup();

  /// Unset up this object after its last use
  virtual void unsetup();

private: // data

  /// dummy variable
  double value;

  /// string read from the configuration file
  std::string m_file;
};

} // namespace ShockFitting

#endif // ShockFitting_DummyComponent_hh
```

Figure 4: List showing `DummyComponent.hh` file.

As appears in Lists 4 and 5, the `DummyComponent` inherits the main `DummyLibrary` functions (*setup* and *unsetup*) and customizes the key function (*create*) to its purposes (e.g: creating a file with a specified value written inside).
To the `m_file` variable is assigned, automatically, the value named `.CreatingFile`

```cpp
#include "DummyLibrarySF/DummyComponent.hh"
#include "Framework/Log.h"
#include "SConfig/ObjectProvider.hh"

//--------------------------------------------------------------------------//

using namespace std;
using namespace SConfig;

//--------------------------------------------------------------------------//

namespace ShockFitting {

//--------------------------------------------------------------------------//

// this variable instantiation activates the self-registration mechanism
ObjectProvider<DummyComponent, DummyLibrary>
dummyComponentProv("DummyComponent");

//--------------------------------------------------------------------------//

DummyComponent::DummyComponent(const std::string& objectName) :
  DummyLibrary(objectName)
{
  m_file = "dummyname";
  addOption("CreatingFile", &m_file,
            "Name of the file to be created");
}

//--------------------------------------------------------------------------//

DummyComponent::~DummyComponent()
{
}

//--------------------------------------------------------------------------//

void DummyComponent::setup()
{
  LogToScreen(VERBOSE,"DummyComponent::setup() => start\n");

  LogToScreen(VERBOSE,"DummyComponent::setup() => end\n");
}

//--------------------------------------------------------------------------//

void DummyComponent::unsetup()
{
  LogToScreen(VERBOSE,"DummyComponent::unsetup()\n");
}

//--------------------------------------------------------------------------//

void DummyComponent::create()
{
  LogToScreen(INFO,"DummyComponent::create()\n");

  ofstream outFile;
  outFile.open(m_file.c_str());

  outFile << value << endl;

  outfile.close();
}

//--------------------------------------------------------------------------//

} // namespace ShockFitting
```

Figure 5: List showing `DummyComponent.cxx` file.

and specified inside the *input.case*[6].
Following this example other `DummyLibrarySF` members can be generated.

5. add the new library to the *CMakeLists.txt* file of the folder in which the current model of the Shock-fitting Solver is called and tested. In List 6 is shown an example related to the `StandardShockFitting` model.

```
LIST ( APPEND TestStandardSF_files testStandardSF.cxx )

# AL: the following line assumes that the module dirs (SF_MODULES_LIST) share the same name
 as the corresponding libs
LIST ( APPEND TestStandardSF_libs ${SF_KERNEL_LIBS} ${SF_KERNEL_STATIC_LIBS}
CFDSolverSF
ConverterSF
CopyMakerSF
DummyLibrarySF
MeshGeneratorSF
RemeshingSF
StateUpdaterSF
VariableTransformerSF
WritingMeshSF
)

IF (SF_HAVE_TRILIBRARY)
LIST ( APPEND TestStandardSF_libs triangleLib TriangleSF )
ENDIF()

SF_ADD_PLUGIN_APP ( TestStandardSF )

##############################################################################

#SF_WARN_ORPHAN_FILES()
```

Figure 6: List showing `TestStandardSF` *CMakeLists.txt* file.

5.2 Linking the new library to the overall framework

Once that the new library is ready to operate, it must be inserted in the overall framework of the Shock-fitting Solver.
In order to accomplish it, follow the steps listed hereafter:

1. define the library vector inside the `ShockFittingObj.hh` file:

```
std::vector<PAIR_TYPE(DummyLibrary)> m_dummyLib;
```

2. make the library *configurable* through the following command (from here after the defined lines must be specified inside the `ShockFittingObj.cxx` file) :

```
m_dummyLib = vector<PAIR_TYPE(DummyLibrary)>();
addOption("DummyLibraryList",&m_dummyLib,
          "List of the DummyLibrary objects");
```

[6]in the *input.case* the KEY = VALUE format will be:
`.DummyLibrary.DummyComponent.CreatingFile = example.txt`

3. make the library *registrable* through the command [7]:

```
createList<DummyLibrary>(m_dummyLib);
```

4. in order to configure, set-up and unset-up the library components at run time, the following lines must be specified:

```
for(unsigned i=0;i< m_dummyLib.size(); i++) {
     configureDeps(cmap,&m_dummyLib[i].ptr()→get();
}

for(unsigned i=0;i< m_dummyLib.size(); i++) }
     m_dummyLib[i].ptr()→setup();
}

for(unsigned i=0;i< m_dummyLib.size(); i++)
     m_dummyLib[i].ptr()→unsetup();
}
```

5. create the library components by following section **??**.

References

[1] Valentina De Amicis, *Implementation and verification of a shock–fitting solver for hypersonic flows*, Master Thesis (2015).

[2] Valentina De Amicis, *An unstructured, two-dimensional, shock-fitting solver for hypersonic flows*, VKI Project Report (2015).

[3] R. Paciorri and A. Bonfiglioli, *Shock interaction computations on unstructured, two-dimensional grids using a shock-fitting technique*, Journal of Computational Physics, **230**, pag. 3155 − 3177, 2011.

[4] Jonathan Richard Shewchuk, *Triangle, a Two-Dimensional Quality Mesh Generator and Delaunay Triangulator.*, `https://www.cs.cmu.edu/~quake/triangle.html`.

[7] this line corresponds to a function already defined in the `ShockFittingObj.hh` file