User/Developer's Manual for ShockFitting (SF):

A modular shock-fitting code for unstructured grids.

Valentina De Amicis, valentina.deamicis91@gmail.com
Dr. Andrea Lani, lani@vki.ac.be
Prof. Renato Paciorri, paciorri@dma.ing.uniroma1.it
Prof. Aldo Bonfiglioli, aldo.bonfiglioli@unibas.it

Introduction

SF is a modular shock fitting solver that can couple to arbitrary unstructured CFD codes.

1 Configuration file

The input file for the code is in a readable text format, where keywords are associated to values according to a dynamic configuration language, which is described hereafter.

1.1 Format description

1.1.1 Full format (discouraged)

The format consists of lines in the form:

```
.OptionA = Value1                # use Value1 as value for OptionA
.Value1.OptionB = Value2         # use Value2 as value for OptionB
.Value1.Value2.OptionC = Value3 # use Value3 as value for OptionC
.Value1.Value2.OptionD = Value4 # use Value4 as value for OptionD
```

where, in each line, the whole LHS is the keyword and the RHS is the value. The latter can be:

- an alpha-numerical string
- an integer
- a boolean (`true` or `false`)
- a floating point number
- an arbitrarily complex analytical function
- an array of all the previous.

The keyword is composed of literal strings separated by ".", corresponding to different *entities* (configurable objects or parameters) defined inside the actual code. The configuration is hierarchical and recursive from top to lowest level. The order in which the options are declared in the file are irrelevant. If needed, the value can be broken into different lines by using the continuation character (back slash) at the end of each line (note that the number of spaces at the end or before the line is irrelevant):

```
.Example.arrays = 4 4 10 \
                  10 4 \
                  4 3
```

Comments start with "#": they can occupy full lines or be placed at the end of the line. Full blocks of comments can be enclosed between a starting and ending **single** "$", without needing a "#" in each line, as in the example below:

```
$
block of comments can be conveniently
written in this way without needing
a "#" character in each line
$
```

1.1.2 Compact format (standard)

The full format example in Sec. 1.1.2 can be significantly simplified:

```
.OptionA = Value1
.Value1 {   # start block of nested options for Value1 (note the ".")
 .OptionB = Value2
 .Value2 {  # start block of nested options for Value2 (note the ".")
 .OptionC = Value3 ; OptionD = Value4   # semicolon separates options
 }    # this brace ends the block for OptionB (Value2)
}     # this brace ends the block for OptionA (Value1)
```

where, in each line, the LHS of the "=" is the keyword and the RHS is the value. Each block of options starts with "" right after the last value and ends with "". The latter must be the only character on the line, a part from comments, if any. Note that with the compact format, different options can be put **within the same line**, separated by ";". While reading the configuration file (see implementation details in SConfig/src/ConfigFileReader.cxx), the compact format is internally converted into the full format.

1.1.3 Errors handling

The file parser is able to detect errors of different kind. It will raise an exception and abort the code if one the following situations occur:

- a keyword (explicitly indicated within "<>") is duplicated;
- a keyword (explicitly indicated within "<>") doesn't exist;
- a value is not satisfying some predefined validating conditions.

If the keyword includes (between two successive ".") the name of a polymorphic object (base class or derived class), the corresponding option, if defined in the parent class (and therefore inherited by the derived ones) can be addressed by both the derived class name or the base class name. **All fields previously defined correspond to base classes**, where most of the options are defined.

1.1.4 Warnings

Let's consider the following example:

```
.BaseObject = DerivedObject  # DerivedObject is a derived class of BaseObject
.BaseObject.pnr = 70         # pnr is an option defined in BaseObject
.DerivedObject.pnr = 40      # but it is also inherited (and "visible") by DerivedObjec
```

If you specify two options for the same parameter (`pnr` in our case), one with the parent `BaseObject` and another with the actual (derived) name `DerivedObject`, the parser will not detect an error but will notify a self-explanatory warning:

```
WARNING: Parameter < pnr > is DUPLICATED: is this what you really expect?
```

In our example, there is actually an error to fix, since it is not clear which value the user wants to assign to `pnr`. If not properly fixed, the error could lead to undefined behaviour.
Let's make another example where the warning would actually be harmless:

```
.DerivedObject.Inner.Funs = 181.38 \
                          1936.18*cos(25./180*pi) \
                          0. \
                          1936.18*sin(25./180*pi) \
                          259.96

.DerivedObject.BcField.Funs = 181.38 \
                          1936.18*cos(25./180*pi) \
                          0. \
                          1936.18*sin(25./180*pi) \
                          259.96
```

In this case, two distinct fields `Inner` and `BcField` (not deriving one from each other!) have been defined, each one specifying an option named `Funs`, corresponding to an array of user-defined functions, one per flow variable. Even though `Funs` is used twice and all corresponding values are the same, the two keywords refer to different entities and is perfectly normal to get the above mentioned warning.

1.1.5 Log files

After parsing the configuration file, two output files are written (only by the process with rank 0 in a paralllel simulation):

- config.log, where all detected options key and values are listed;
- options.dat, which dumps all descriptions associated to the option keys corresponding to the selected simulation fields.

1.1.6 Interactive parameters

All parameters that have been declared interactive (see following section) will be updated after a number of iterations indicated by the user (which is also by itself interactively modifiable):

```
.config_rate = 1000 # every 1000 iterations the code updates the configuration
```

2 SConfig: a configuration library (by Andrea Lani)

The parsing of the configuration file and its interpretation is performed by a newly implemented library, that we have denominated *SConfig*. The latter combines concepts inspired by [1] (self-registering objects) and [2] (object configuration) and more sophisticated techniques described in [4, 3] (self-registering and self-configuring objects) into a new implementation.
The SConfig library is completely independent from ADPDIS3D and can be linked statically or dynamically to whatever other code to take profit of the very general functionalities it provides.
Herein, while the C++ implementation details are left out of the present document, the usage of the library from a developer point of view will now be addressed.

2.1 Self-configuring objects

In order to take profit of the self-configuration capability, the developer must define **objects deriving from `ConfigObject`** or from other objects already deriving from it. The interface of `ConfigObject` is shown in listing 1. Here follows some guidelines for activating the self-configuration capability.

Listing 1: ConfigObject interface.

```
class ConfigObject : public NamedObject {
  public:

   // constructor
   ConfigObject(const string& name);

   // virtual destructor
   virtual ~ConfigObject();

   // set the parameter corresponding to the following inputs
   template <typename TYPE>
   void addOption(const string& optionName,
                  TYPE* optionPtr,
                  const string& optionDescription,
                  bool isDynamic = false,
                  OptionValidation<TYPE>* condition = NULL);

   // configure all parameters from file
   virtual void configure(OptionMap& cmap, string* prefix = NULL);

   // configure dependent objects
   virtual void configureDeps(OptionMap& cmap, ConfigObject *const other);

protected:
    // get the name of the parent
    virtual string getParentName() const = 0;

 ... // private data follows
};
```

1. An object deriving from `ConfigObject` must implement the pure virtual function `getParentName()` specifying the name of the parent class. This allows users to specify one of both the parent class name or the derived class name for configurable options defined in the parent (see

Sec. 1.1.4 for an example), in order not to have to introduce too many changes in new configuration files when adapting them for new cases.

2. A `ConfigObject` child needing to declare some options must **call explicitly the `addOption()` function for each single option** inside its own constructor, specifying the appropriate arguments to it, namely:
   - the option name;
   - a pointer to the local (in the class) variable to be linked to the option;
   - the option description (indicating the purpose of the option);
   - a boolean (`true` or `false`) to specify if the option is eligible to be modified interactively during the simulation (default value is `false`);
   - a pointer to an object deriving from `OptionValidation` responsible for checking if the option value is valid.

   The last two function arguments are optional and can be omitted. However, the fifth argument cannot be specified if the fourth isn't present (this is due to C++ treatment of default arguments).
3. A `ConfigObject` child can in general omit to implement the function `configure()`, unless something special is needed during the configuration phase, like configuring nested objects (to which the object in question holds pointer) via `configureNested()`. **Inside the `configure()` implementation, a call to `configure()` on the closest parent class must be executed first**.
4. The `configureNested()` function defined in `ConfigObject` should not be overridden by derived classes (even though such a possibility is left open in case a SConfig user really need to do something extraordinary atypical).

2.2 Self-registering objects

The self-registration technique pioneered in [1] allows polymorphic objects (i.e. deriving from some existing base classes) to be dynamically integrated into an existing code, without having to modify a single line of the latter. The technique effectively replaces the use of conditional constructs `if-else-if` or `switch` in a more elegant and flexible way. SConfig provides a generic self-registration capability which can be enabled as explained hereafter.

1. The self-registration affects all objects belonging to the same hierarchy, or, in other words, with the same polymorphic type, which is the same as the base class type. In a generic base class `Base` of a hierarchy of potentially self-registering objects, one must define the type `PROVIDER`, the type `ARGS` (constructor arguments) and the class name, as shown in listing 2. In our implementation, `Base` constructor must

Listing 2: How to enable Self-registration.

```
class Base {
  public:
    typedef Provider<Base> PROVIDER;   // definition of the type PROVIDER
    typedef ArgumentsType ARGS;        // definition of the type ARGS

    Base(ArgumentsType args);          // constructor taking ArgumentsType
    static string getClassName(){return "Base";}    // define class name

    // ... whole class definition follows
};
```

   accept one single argument which can be of generic type. In case more than one argument is needed, `ArgumentsType` can be a `std::pair` or tuple object grouping multiple arguments (or even none) of whatever type, for instance:

Listing 3: Examples of argument tuples.

```
template <typename T1, typename T2, typename T3>
struct Args0 {};

template <typename T1, typename T2, typename T3>
struct Args3 {T1 arg1; T2 arg2; T3 arg3;};
```

2. All objects to be made self-registering must derive from a base class satisfying the previous requirement and must define a global variable of type `ObjectProvider` parameterized with the derived class static and polymorphic types in a source file (not on an header!). An illustrative example is shown in 4.

Listing 4: ObjectProvider defined in DerivedClass.cxx.

```
ObjectProvider<MyDerived, Base> myDerivedProvider("MyDerived1");
```

The provider associates a name to a polymorphic object (in this case "MyDerived1" to `MyDerived`) allowing the code to construct such an object on demand, just by knowing its name.

2.3 A clarifying example

After having explained how to make an object self-configurable or self-registering, we are now ready to analyze a case where the two properties are combined together, in order to get maximum profit of the two techniques. Let's consider the base class `BaseObject` (actually defined inside ADPDIS3D), whose interface is presented in listing 5.

Listing 5: BaseObject class definition.

```
#include <ConfigObject.hh>
#include <Provider.hh>

class BaseObject : public ConfigObject {
public:
  typedef Provider<BaseObject> PROVIDER;
  typedef const string& ARGS; // constructor takes a string (name) as input

  BaseObject(const string& name); // constructor
  virtual ~BaseObject() {}        // destructor

  // ... member virtual some functions

  // get the class name
  static string getClassName() {return "BaseObject";}

protected:
  std::string getParentName() const {return getClassName();}
};
```

On the one hand, `BaseObject` derives from `ConfigObject`, which makes it self-configurable, on the other hand, it defines the types `PROVIDER`, `ARGS` and the class name, which make itself and its children eligible for self-registration.
Let's now consider a prototype derived class `DerivedObject`, implementing a specific kind of problem for

multiple overlapping meshes, as defined in listing 6. In the file DerivedObject.cxx, we can instantiate the corre-

Listing 6: DerivedObject class definition.

```
class DerivedObject : public BaseObject {
public:
  DerivedObject(const string& name); // constructor
  ~DerivedObject();                  // destructor

  // ... some member functions

  // configure all parameters from file
  void configure(OptionMap& cmap, std::string* prefix);

private:
  vector<double> m_inletState;  // state vector defining the inlet
  int m_nspecies;               // number of species
};
```

sponding provider by defining: This provider defines the name ”DerivedObject” which qualifies a specific kind

Listing 7: ObjectProvider for DerivedObject defined in DerivedObject.cxx.

```
ObjectProvider<DerivedObject, BaseObject> oversetProvider("DerivedObject");
```

of `BaseObject` and which can be used as a keyword or as a value in the configuration file to configure the corresponding object (see examples in Sec. 1.1.4).
We can now qualify the private data of `DerivedObject` to be configurable by following the guidelines described in Sec. 2.1 and adding the corresponding options inside the constructor. Herein, two configurable options

Listing 8: DerivedObject constructor.

```
DerivedObject::DerivedObject(const string& name) : BaseObject(name)
{
  m_inletState = vector<double>();  // default value for inlet field
  addOption("inlet",&m_inletState,"State for inlet field", true,
            new GenericValidation< vector<double> >("if(w>0.,1,0)"));

  m_nspecies = 0;                 // default value for number of species
  addOption("species",&m_species,"Number of species", false,
      new CrossCheckValidation<int>
      ("if((w>0&w1=4&w2>0)|(w=0&w1!=4&w2>=0),1,0)","equation,implicit"));
}
```

(key names and corresponding values) are defined: ”inlet” and ”species”, of type `vector<double>` and `int` respectively. The value of ”species” cannot be changed interactively during the simulation (fourth argument of `addOption()` is set to `false`), while the inlet state can be modified interactively, allowing time dependent inlet conditions.
In both cases, we focus our attention on the last parameters which define the option validation conditions.
`GenericValidation` accepts a string specifying one arbitrarily complex analytical function `F` in the form

```
if(F(w),1,0)
```

where ”w” indicates the option value (which will be set in the input file) and ”1” stands for ”true” and 0 for ”false”. In the first simple case (here included only for illustrative purposes), we ask that each individual component of `m_inletState` is positive:

```
if(w>0.,1,0)
```

We remark that the expression can be much more complex than this, involving analytical mathematical functions (abs, sqrt, tan, sin, etc.) and constants (pi, e, etc.).
The second example better shows the power of this approach. `CrossCheckValidation` takes two strings: the first string defines the analytical validation function

```
if((w>0&w1=4&w2>0)|(w=0&w1!=4&w2>=0),1,0)
```

where ”&” and ”—” stands for ”and” and ”or”; the second string specify a comma separated list of other existing configurable options (”equation,implicit” in our example) which have been defined in the class itself or in whatever other configurable object existing in the code. Those options name are mapped to variable names of the type ”wN” where N is a number corresponding to the position in the specified name list. As a result, ”w1” corresponds to ”equation”, ”w2” corresponds to ”implicit” in the validation expression

```
if((w>0&w1=4&w2>0)|(w=0&w1!=4&w2>=0),1,0)
```

allowing developers to cross check validity of one user-input value (the value corresponding to ”species”), in comparison with other related inputs (the values corresponding to ”equation” and ”implicit”).
The input and parsing of analytical functions is made possible by the open source FunctionParser library (version 4.0.5) [5] available under LPGL license, whose full sources and licence files are included in the current ADPDIS3D distribution and automatically pre-compiled into a separate statically linked library.

2.4 User-defined anaytical fields

Let’s now elaborate on the example in 2.3 to show how anaytical functions turn out to be useful also in different contexts inside ADPDIS3D, for instance to input generic space/time varying boundary conditions or initial fields. We substitute the constant initial values defined inside `DerivedObject` with an array of generic initial fields, one per conservative (or other useful) variable (see listing 9). Herein, the variable $m\_iniNames$ defines the names

Listing 9: DerivedObject class definition 2.

```
class DerivedObject : public BaseObject {
  // ... everything as before
  private:
    int m_nspecies;                          // number of species
    vector<string> m_iniNames;               // names of initialization fields
    vector<FunctionField*> m_iniFields; // initialization fields functions
    // other useful private data (check the code ...)
};
```

for fields for which a specific initialization wants to be imposed: the first name (user-defined) **must** correspond to the internal portion of the domain (”Inner”, ”Domain”, or whatever else); the other names must be the boundary IDs corresponding to the boundary conditions specified in the mesh file (e.g. ”100” for Dirichlet, ”8” for periodic boundary etc., as defined in the file bc.F) if the user wishes to specify some specific initial values. Each name is associated to a different `FunctionField`, a wrapper class for the FunctionParser [5] defined in ADPDIS3D enabling the end-user to input multiple anaytical functions, one per flow variable. The implementation details are left to the interested reader. We look at the effect on a prototype configuration file:

```
.DerivedObject.IniFields = Inner

.Inner {                        # initialization field names
                                # specific settings for the field called "Inner"
.InVars = fact
.InDefs = if(R2(x,y)<=0.5,0.,if(R2(x,y)>0.9,1.,(R2(x,y)-0.5)/0.4))
.Funs = 0.01*(fact+7.*(1.-fact)) \      # rho
                  42.7021*FC 0. 0. \    # rhoU rhoV rhoW
                92956.5597205 0. 0. 0.  # rhoE Bx By Bz
}
```

We have declared some parameters for the initialization field, namely `InVars`, `InDefs` and `Funs`:

- `InVars` allows the user to enter the names of some new variables that will be used in the expressions specified by `Funs`. By default (if `InVars` is omitted), `Funs` can be formulated only in terms of spatial coordinates (x,y,z) and t (time).
- `InDefs` are the functions defining the variables declared by `InVars`. Those functions can only be using x,y,z,t as indipendent variables. The number of `InVars` defined, separated by a space, must match the number of `InDefs`.
- `Funs` defines functions of x,y,z,t and the extra variables named in `InVars`.

If we look at the given example, the user has defined an extra variable called `fact` as

```
if(R2(x,y)<=0.5,0.,if(R2(x,y)>0.9,1.,(R2(x,y)-0.5)/0.4))
```

where `R2(x,y)` computes the distance in 2D (`R2(x,y,z)` should be used in a 3D case). The final expressions of the initial field, one per conservative variable (=8 in total), are functions of x,y,z,t and `fact`, as shown above, and they are clearly much simpler than what they would be without the definition of the intermediate `fact` variable.
In our example we have restricted ourselves to an initialization field, but a similar treatment is reserved to boundary condition fields as well.
As far as the FunctionParser is concerned, syntax errors in the analytical functions, if present, will be detected and notified to the user. In particular, **no spaces are allowed inside an analytical function.**
An extensive list of **predefined supported functions** is available directly on the FunctionParser official webpage: *http://warp.povusers.org/FunctionParser/fparser.html#parsertypes*. More functions and mathematical constants can be defined inside the header file `src/CPP/DerivedParser.hh`, provided within this distribution: interested developers can check the file itself for examples on how `R2`, `R3` and some constants (`pi`, `e`, `Rair`) were added.

References

[1] J. Beveridge. Self-Registering Objects in C++. *Dr. Dobb's Journal*, 8/1998.

[2] J. Pace. Another Getopt Library, http://yagol.sourceforge.net, 2003.

[3] A. Lani. *An Object Oriented and high performance platform for aerothermodynamics simulation*. PhD thesis, Von Karman Institute for Fluid Dynamics, 2008.

[4] T. L. Quintino. *A Component Environment for High-Performance Scientific Computing. Design and Implementation*. PhD thesis, Von Karman Institute for Fluid Dynamics, 2008.

[5] J. Nieminen, J. Yliluoma. Function Parser for C++, http://warp.povusers.org/FunctionParser/, 2009.