Chapter 6

Variable Base Interface

6.1 Introduction

Finite element codes has been changed a lot during the evolution of the Finite Element Method, In its early times, finite element applications were developed to solve a certain type of problems in an specific domain. Many applications for solving shells, plates, stokes fluid etc. were created in this period. By the time, engineers began to apply FEM in engineering problems which mostly consist of more than one type of problem. First they tried to solve it by separating the problem and solve each part by one module meanwhile developers began to gather previous small modules related to the same domain in a domain specific application which could handle more complex problems. Following this line various applications for solving structural, fluid and other problems created.

Applying FEM to solve a coupled problem was a natural step forward from single domain problems. Reusing the existing codes and connect them via an interface is the most common and approved way to deal with these problems. In this way complexity reduced and existing modules for solving each domain can be reused to solve coupled problem. But what happen if the interfaces are inconsistent? Writing a file and reading it in another module is a common way while there is more elegant way to do it using libraries which can handle objects in a generic way. There are many successful examples using file interface or libraries like CORBA [101] or omniORB, though using them can cause big overheads in the performance of the code. The latency time of invoking a request on a CORBA object is approximately 500-5000 times higher than doing it by a function call in C++ [58].

In these days still developing and extending finite element applications to solve new and more complex problems is a challenge. Also using FEM in any new area creates a new demand for developing an application to make the method applicable in practice. So flexibility, extensibility and reusability are the key features in the design and development of modern finite element codes.

Though the previous strategy to chain different modules in an application works fine, the level of reusability is respectively low. The reason is that in this manner we can reuse the whole module but not a part of it. For example each module has its own data structure and IO routines which cannot be used by another method or new modules in the same way. This is the motivation to establish a variable base interface which can be used at high and low levels in the same manner.

6.2 The Variable Base Interface Definition

In many connection points between different parts of a finite element program we are asking a value of some variable or mapping some variables and so on. The methodology to design this interface is sending each request with variable or variables it depends on. It sounds simple and natural, meanwhile it is a powerful trick to make generic algorithms and modules using variables.

First let us see what we need in general to create some finite element generic algorithms:

- Type of a variable is an important information while many operation and also storing mechanism are depended to it
- Sometimes we need also the name of variable, (i.e. to print the results).
- While each variable can be identified by its name but using name in checkpoints give a big unnecessary overhead (comparing two strings is respectively slow) so an identifier number is essential, for example in case of fast searching and database working.
- Another useful information is a zero value. Though this information seems unnecessary but it helps a lot in case of initializing zero vectors and matrix with the correct size.
- Also in some modules we may need to work just with some component of a variable. In such a cases we need a mechanism to identify the components owner.
- Another useful feature is to work with raw pointers. This can be useful specially in the case of connecting external modules and extracting value form pointer passed by modules.

Keeping in mind all above requirements now a global design is possible. In this variable base interface:

- A variable encapsulate all information needed by different objects to work in a generic way over different variables. Doing this helps to simplifying modules interfaces. Considering a `PrintNodalResult` module which normally need result name, an index to retrieve nodal results and a zero value if some `Nodes` don't have results. All this information can be passed by one variable in this design.
- Each module must configure itself in term of variable or variables given to it.
- Type-safety reached by statically typing variables.
- Each instance of variable class has the same name as its represented variable name. This is a great added value to code readability.

6.3 Where Can be used?

As mentioned earlier, in a multi-disciplinary application always there is a need to exchange data between different domains. So one important point is to exchange data in a robust and safe way. Developing different modules by the VBI (Variable Base Interface) make data exchanging between them trivial, though each domain has different data stored and sometimes the internal data structure may be different. For example a fluid application may have velocities and pressures stored while a thermal domain just temperature. When using variables the interface provides the means to recover their values as well as to store new values for given variable.

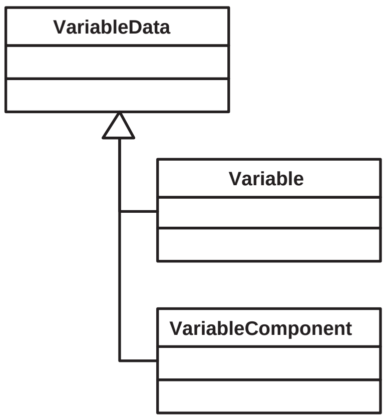

Figure 6.1: Variable classes structure

Now let us go one level down, by developing a set of generic algorithms and data structures in a VBI conforming manner. This can boost up reusability of the code. Also code management becomes easier while any new extension can be done without changing the interface. There are several algorithms which can be equipped with VBI. For example:

Data Structure First and more important place is to store and then retrieve a value in data structure. Here type, id and zero value are needed to create a typesafe data container. All this information is represented by a variable. The interface is `GetVAlue` or `SetValue` and so on for given variable.

Reading input file Each variable has its type and name, so they can be used to validate tokens. In this way the input interpreter can be extended to read new data just by defining a variable presenting this new data.

Printing output Printing output needs name of variable and its value and zero in case of no data. Using a data structure with VBI, and sending a request to print a variable gives all information to print even for new variables.

Mapping or interpolating Again using a VBI conforming data structure we can generalize, or map and interpolate algorithms.

6.4 Kratos variable base interface implementation

In Kratos `VariableData` and its derivatives `Variable` and `VariableComponent` represent interface information.

6.4.1 VariableData

`VariableData` is the base class which contains two basic information about the variable it represents; Its name `mName`, and its key number `mKey`.

`VariableData` has trivial access methods for these two attributes while also has virtual and empty methods for raw pointer manipulation. The reason of not implementing these methods here is the fact that `VariableData` has not any information about the variable type it represents. Lack

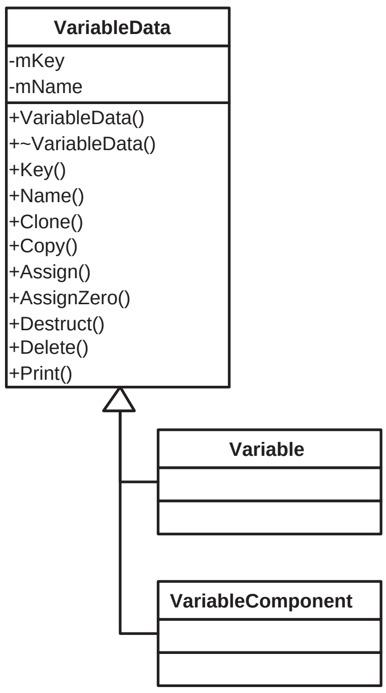

Figure 6.2: VariableData class

of type information in `VariableData` make it unsuitable and less usable to pass it in many parts of the interface but the idea of this class is to provide a lower level of information which is common between all types of variables and their components and use it as a place holder when there is no distinction between variables and their components. Also in this implementation we use a virtual method base to dispatch various operations on raw pointers. This may result a poor performance in some cases while the function call overhead seems to be considerable.

6.4.2 Variable

`Variable` is the most important class in this structure. It has information represented by `VariableData` which is derived from it. Also it has another important information which is the type of variable representing.

Using C++ as implementing language `Variable` has its data type as a template parameter. In this manner interface can be specialize for each type of data and also type-safety can be achieved. Another important advantage of having `variable` in template form with its data type as template parameter is to have a restriction form in interface. If we want to restrict a method to accept just matrices for instance, then by passing a `variable<Matrix>` as its argument we can prevent users to pass another type by mistake. This feature shows to be important especially in some cases which there are different types representing the same variable. (Constitutive matrix and its corresponding vector is a good example of this case). Variable data type `TDataType` must satisfy following requirements:

- Copy constructible.

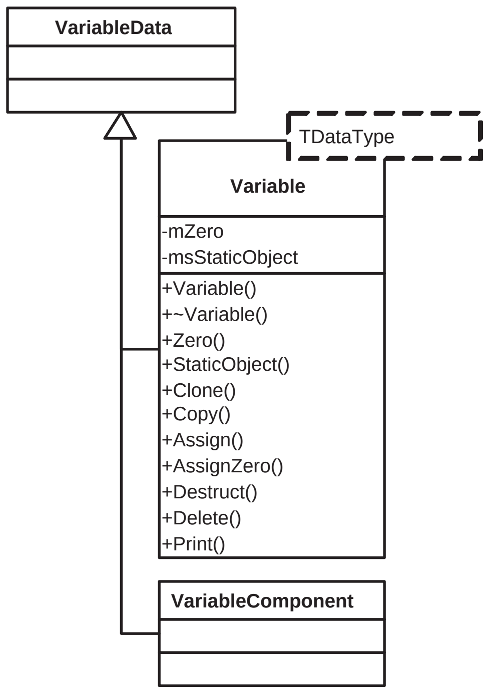

Figure 6.3: Variable class

- Assignable.
- streaming operator defined

In `Variable` by knowing the data type raw pointer manipulating methods are implemented. These methods use raw pointers to perform basic operations over its type:

- `Clone` create a copy of this object using copy constructor of class. It takes a `void*` as its argument and returns also `void*` which is pointing to the memory allocated for cloned object. `Clone` is useful to avoid shallow copying of complex objects and also without actually having information about variable type.
- `Copy` is very similar to `Clone` except that destination pointer also passed as a `void*` argument to it. it is a helpful method specially to create a copy of heterogeneous data arrays .
- `Assign` is very similar to `Copy`. It just differs in using assignment operator beside copy constructor. `Copy` is to create a new object while `Assign` do the assignment for both existing objects.
- `AssignZero` is a special case of `Assign` which variable zero value used as source. This method is useful for initializing arrays or resetting values in memory.
- `Delete` removes an object of variable type from memory. It takes a `void*` as its argument which is the location of object to be removed. `Delete` calls destructor of object to prevent memory leak and frees the memory allocated for this object assuming that object allocated in heap.

- `Destruct` is to destruct an object maintaining the memory it is using. It takes a `void*` as its argument which is the location of object to be destructed. It calls destructor of object but unlike `Delete` it does nothing with memory allocated for it. So it is very useful in case of reallocating a part of memory.

- `Print` is an auxiliary method to produce output of given variable knowing its address. For example writing an heterogenous container in output stream can be down using this method. This method assumes that streaming operator is defined for the variable type.

All this methods are available for low level usage. They are useful because they can be called by a `VariableData` pointer and equally for all type of data arranged in memory but maintaining typesafety using these methods is not straightforward and needs special attention.

Zero value is another attribute of `Variable`, stored in `mZero`. This value is important specially when a generic algorithms needs to initialize a variable without losing generality. For example an algorithm to calculate the norm of a variable for some `Element`s must return a zero if there is no `Element` at all. In case of double values there is no problem to call default constructor of variable type but applying same algorithm to vector or matrix values can cause a problem because default constructor of this types will not have the correct size. But returning a zero value instead of default constructed value keeps generality of algorithms even for vectors and matrices, assuming that variables are defined properly.

There is another method which is `StaticObject`. This method just returns `None` which is an static variable of this type with None as its name and can be used in case of undefined variable (like null for pointers). It is just an auxiliary variable to help managing undefined, no initialized or exceptional cases.

6.4.3 VariableComponent

As mentioned before, there are situations that we want to deal with just component of a variable but not all of it. `VariableComponent` implemented to help in these situations.

`VariableComponent` like `Variable` derived from `VariableData`.

`VariableData` is a template taking an adaptor as its argument. Adaptor is the extending point of component mechanism. For any new type's component a new adaptor needed to be implemented. This adaptor type requirements are:

- `GetSourceVariable` method to retrieve parent variable.

- `GetValue` method to convert extract component value from source variable value.

- `StaticObjec` used to create none component.

Unlike `Variable`, `VariableComponent` has not been implemented to have zero value or raw pointer manipulators. A zero value can be extracted from the source value so there is no need to have it here. Operations over raw pointers are not allowed here by purpose. This interface manages variables entirely and not just some part of them. In fact a part of an object cannot be copied, cloned, deleted or destroyed. So these methods are not implemented to protect objects from unsafe memory operations.

Having adaptor as template parameter helps compiler to optimize the code and eliminating overheads. In this manner adaptor's `GetValue` method will be inlined in `VariableComponent`'s one so there won't be any overhead due to decomposition while extensibility reached.

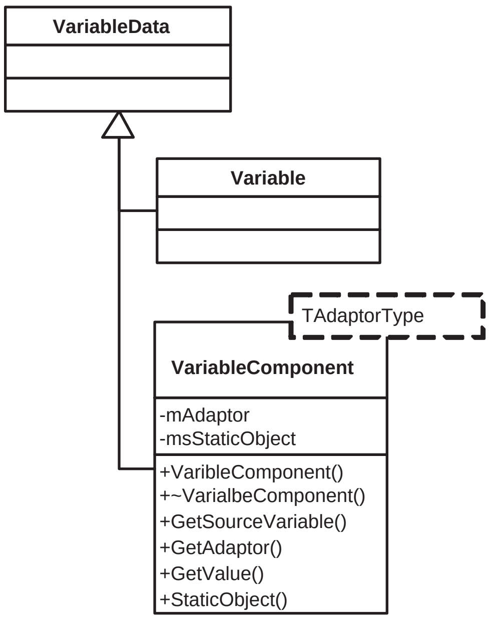

Figure 6.4: VariableComponent class

6.5 How to Implement Interface

Sending any request with variables is the way this interface works. But to make the things working in real world a uniform way to getting and setting variables value in objects must be defined.

6.5.1 Getting Values

In many interfaces there are methods like `GetDisplacement` or `Flux` to get values of for example displacement or thermal flux. This cannot work for a generic interface while the variables to be accessed can be completely different from one domain to another. The idea is to specify the variable not by the method name but by the variable passed to a generic method. In this manner any previously defined algorithm can be reused for new domain and new variables using this generic access method.

Having a `GetValue` method is essential for each class which contains a data to be processed. Note that the name of this method doesn't change with the variable we want to get . For classes with just one type or few types of data it can be written as a normal method using specific variable

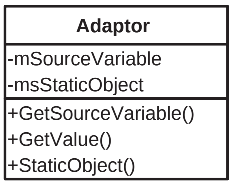

Figure 6.5: Adaptor class

type as argument:

```
DataType GetValue(VariableType const&) const

DataType const& GetValue(VariableType const&) const

DataType& GetValue(VariableType const&)
```

These three versions are different in returning the values which make them usable in different places.

The first version is more suitable for returning some calculated and not stored value and also for small objects like doubles.

The second one is useful for an internal value access method returning just a reference to the internal variable without any cost.

The third version is for left value representation of an internal variable via this interface. The returned reference is not constant so the value can be changed and can be used as a left value.

In the case of classes containing arbitrary type variables like heterogenous containers `GetValue` can be implemented in template form maintaining generality of the code

```
template<TDataType> TDataType GetValue(Variable<TDataType> const&)
const

template<TDataType> TDataType const& GetValue(Variable<TDataType>
const&) const

template<TDataType> TDataType& GetValue(Variable<TDataType>
const&)
```

In this way the interface is open to not only any new variable but also to any new type of variables. Here template member specialization can be used as a very useful tool to define exceptional cases and also to optimize the code using different specialized implementations.

Sometimes it is necessary to take a variable component and not hole variable. Note that `VariableComponent` is not convertible to `Variable` therefore cannot be pass as argument in above patterns. So to make this possibility another set of methods with `VariableComponent` as their argument needed to be implemented. As before, an specific implementation can be done using known adaptor for specific component:

```
DataType GetValue(
  VariableComponentType const&) const

DataType const& GetValue(
  VariableComponentType const&) const

DataType& GetValue(VariableComponentType const&)
```

Also template implementation for more generic interface:

```
template<TAdaptorType> typename TAdaptorType::Type GetValue(
  VariableComponent<TAdaptorType> const&
  ) const

template<TAdaptorType> typename TAdaptorType::Type const&
GetValue(
  VariableComponent<TAdaptorType> const&
  ) const

template<TAdaptorType> typename TAdaptorType::Type& GetValue(
  VariableComponent<TAdaptorType> const&)
```

In above patterns having the type of component defined in adaptor used to specify the type of return value. In this manner adaptor is responsible to specify the return value type and extensibility guaranteed.

Finally methods for getting variables and those for getting components can be combined in one set of methods creating a uniform interface:

```
template<TVariableType> typename TVariableType::Type GetValue(
  TVariableType const&) const

template<TVariableType> typename TVariableType::Type const&
GetValue(
  TVariableType const&) const

template<TVariableType> typename TVariableType::Type& GetValue(
  TVariableType const&)
```

6.5.2 Setting Values

This part of the interface has the same characteristic of getting values. Though the third version of getting values can be used to setting it as a left value but in some cases a separate set of methods needed to ensure generality of the interface.

To set a value of a variable it is necessary to pass both the variable and its value as `SetValue` argument. Again implementation depends on the variety of variables that have to be accessed via this interface. For few variable types accessing `SetValue` method can be implemented by knowing data type and its variable type:

```
void SetValue(VariableType const&,
         DataType      const&)
```

Not that here we have just one version, due to the fact that `SetValue` method cannot be constant. Again to implement a more general version templates are useful:

```
template<TDataType> void SetValue(VariableType const&,
         TDataType     const&)
```

In the case of setting a component the two above versions must be modified in order to accept a `VariableComponent` instead of `Variable`.

```
void SetValue(VariableComponentType const&,
         DataType              const&)
```

or template version using adaptors:

```
template<TAdaptorType> void SetValue(VariableComponentType
const&,
     typename TAdaptorType::Type const&)
```

Finally the uniform version combining variables and their components in the same method:

```
template<TVariableType> void SetValue(TVariableType
const&,
     typename TVariableType::Type const&)
```

6.5.3 Extended Setting and Getting Values

In some cases there are additional information needed to access a variable inside a class. For simple cases an additional index is enough to indicate solution step or `Node`'s number which are passed as extra argument.

In more complex interfaces an information object with a VBI interface also is passed as argument to extend its generality. In this way any new information just passed through existing interface and can be retrieved by the interface.

```
template<TVariableType> typename TVariableType::Type
GetValue(TVariableType const&,
         InfoType const&) const

template<TVariableType> typename TVariableType::Type const&
GetValue(TVariableType const&,
         InfoType const&) const

template<TVariableType> void SetValue(TVariableType
const&,
     typename TVariableType::Type const&,
     InfoType                      const&)
```

Using the variable base interface to extending itself is a very powerful trick to create extremely flexible interfaces without sacrificing the clarity of the code. In this way not only the type and the variable to be passed is extendible but also the number and type of arguments to be passed can be arbitrary extended.

6.5.4 Inquiries and Information

Not all the time we need to transfer data by the interface and there are some situations where an inquiry is needed or some information about an object respect to a certain variable is required. Assuming that in this operations the type and components of a variable are not important, the base class `VariableData` can be used in interface to deal with all kinds of variables and their components in the same way.

6.5.5 Generic Algorithms

Now it is time to implement generic algorithms using this interface. In general algorithm interface can use any forms described above for getting and setting specially the extended ones. In fact each algorithm is very different in the arguments it accepts but what is important here is the variable or variables needed which are passed through the interface. The point is to use the the interface of objects inside algorithms using variables passed to it. In this way the algorithm is transparent to the working variable and can be applied generally to any new variable in new domains.

Many algorithms can be generalized in this way, reading inputs, print results, mapping and interpolating values, calculating norms and errors, etc. All these algorithms have a common property: they operate over the value of some variable. A carefully implementation of these algorithms can free them from knowing really which variable they are working on. They just work on the variable which the user passes to them and use that to extract values it is needed. In other word, algorithms are implemented between two layer of interfaces.

The idea looks simple, but in practice results in reusable algorithms which can be used in any new area of work with a new set of variables.

6.6 Examples

To see how above idea works in practice and also to make some details more clear, some examples from different levels of interface are presented next.

6.6.1 Nodal interface

A very first example is to access nodal values in a finite element program. We will consider that any new extension to the program may introduce new set of variables to access. Using variable base interface can be a solution for extensibility needed in these cases. A basic get and set value interface implemented to give basic accessibility is as follow:

```
// Getting a reference
template<class TVariableType> typename TVariableType::Type&
Node::GetValue(TVariableType const&) {
// Accessing to database and
// returning value ...
}

// Getting a constant reference
template<class TVariableType> typename TVariableType::Type const&
Node::GetValue(TVariableType const&) const {
// Accessing to database and
// returning value  ...
}

template<class TVariableType> void Node::SetValue(TVariableType
const&, typename TVariableType::Type const&) {
// Accessing to database and
// setting value  ...
}
```

It can be seen that the uniform version of the interface adapts here perfectly and prevent us from having different versions of `GetValue` and `SetValue` for variables and variables components. And finally overwriting the `[]` operators make the syntax easier to use:

```
template<class TVariableType> typename TVariableType::Type&
Node::operator[](const TVariableType&) {
  return GetValue(rThisVariable);
}
```

Also some inquiries can be implemented to see if the variable exists:

```
template<class TDataType> bool Node::Has(Variable<TDataType>
const&) const {
// Check if the variable
// stored before ...
}
```

```
template<class TAdaptorType> bool
Node::Has(VariableComponent<TAdaptorType> const&) const {
// Check if the variable component
// stored before ...
}
```

Here, the separate variable and components methods used to implement this part of interface. As mentioned before, This can be done just by using the `VariableData` in the interface:

```
bool Node::Has(VariableData const&) const {
// Check if the variable
// stored before ...
}
```

Note that, to use the `VariableData` version a uniform search by key id is required from the internal containers.

Now it is easy to use this `Node` in the code and access any variable through interface:

```
// Getting pressure of the center node

double pressure = center_node[PRESSURE];

// Setting velocity of node 1

Nodes[1][VELOCITY] = calculated_velocity;

// Printing temperature of the
// nodes to output

for(IteratorType i_node = mNodes.begin() ;
    i_node != mNodes.end() ; i_node++)
{
    std::cout << "Temperature of node #"
        << i_node->Id()
        << " = "
        << i_node->GetValue(TEMPERATURE)
        << std::endl;
}

// Setting displacement of nodes to zero

Vector zero = ZeroVector(3); for(IteratorType i_node =
mNodes.begin() ;
    i_node != mNodes.end() ; i_node++)
{
  i_node->SetValue(DISPLACEMENT, zero);
}
```

Accessing to history of nodal variable is an example of a simple extended interface. Keeping previous arguments for `GetValue` and `SetValue` an additional parameter can be passed to indicate for example time step needed.

```
template<class TVariableType> typename TVariableType::Type&
Node::GetValue(const TVariableType&,
  IndexType SolutionStepIndex)
{
// Accessing to database and
// returning value ...
}

template<class TVariableType> typename TVariableType::Type const&
Node::GetValue(const TVariableType&,
  IndexType SolutionStepIndex) const
{
// Accessing to database and
// returning value ...
}
```

Also () operator can be overridden to provide an easy interface. (Note: [] operator cannot accept more than one argument and cannot be used here)

```
template<class TVariableType> typename TVariableType::Type&
Node::operator()(const TVariableType&,
  IndexType SolutionStepIndex)
{
  return GetValue(rThisVariable,
                  SolutionStepIndex);
}
```

In practice this interface can be more complicated due to the fact that the history must not be stored for all variables and it is useful to have separate containers to store historical and not historical variables. For example in Kratos there are two sets of access methods to give the possibility of storing only needed history and not all of them.

6.6.2 Elemental Interface

In `Elements`, access methods can be implemented like `Nodes`. it is better to keep the interface as unchanged as possible to avoid extra effort due to the inconsistence interfaces. So let's leave the access methods as before and take some other methods to make examples.

Methods to calculate local matrices and vectors also can be implemented using VBI. But what is the advantage of using VBI here? There are several additional parameters needed for calculating local contributions of each `Element`, like time step, current time, delta time and so on. These arguments can be completely different from one formulation to the other and passing all of them as individual arguments is somehow impossible. An approach is to create a helper class to encapsulate these arguments and pass all of them through this helper class. Again the VBI can be used here to make a uniform interface also at this level of working. In Kratos there is a helper class named `ProcessInfo` which is passed to the methods which calculating the local systems as follows:

```
virtual void SomeElement::CalculateLocalSystem(
    MatrixType& rLeftHandSideMatrix,
    VectorType& rRightHandSideVector,
    ProcessInfo& rCurrentProcessInfo)
{
  // Getting process information
  double time = rCurrentProcessInfo[TIME];
  // Calculating local matrix and
  // vector ...
}
```

6.6.3 Input-Output

Writing a generic input-output with enough flexibility is a complex task. Making extensions causes problems in many codes while the IO is not flexible enough to process new variables.

In modern applications a parser used to read the input file and understand the input grammar [54, 77]. Normally this input file parser is a complex part of the code and modifying it for any new variable added by extensions is expensive and not free of bugs. A good solution is to make the parser work with a list of variables and change the list each time reading a new variable is needed. Doing this adding new extensions to the input parser can be much easier while the parser itself won't be changed. The input file parser is too big to add here as an example and interested readers can find a working version of it in Kratos `DatafileIO` class.

Writing output files like reading inputs can be generalized using variables. In this way any new extension to the library can write its results using existing output procedures. Here is an example of using variables to write a generic output procedure for GiD [84, 83]:

```
void GidIO::WriteNodalResults(Variable<double> const& rVariable,
                              NodesContainerType& rNodes,
                              double SolutionTag,
                              std::size_t SolutionStepNumber)
{
  GiD_BeginResult( (char*)(rVariable.Name().c_str()),
    "Kratos",
    SolutionTag,
    GiD_Scalar,
    GiD_OnNodes, NULL, NULL, 0, NULL );

  for(NodesContainerType::iterator i_node = rNodes.begin();
      i_node != rNodes.end() ; ++i_node)
      GiD_WriteScalar( i_node->Id(),
          i_node->GetSolutionStepValue(rVariable,
              SolutionStepNumber));

  GiD_EndResult();
}

void GidIO::WriteNodalResults(Variable<Vector> const& rVariable,
                              NodesContainerType& rNodes,
                              double SolutionTag,
                              std::size_t SolutionStepNumber)
{
  GiD_BeginResult( (char*)(rVariable.Name().c_str()),
    "Kratos",
    SolutionTag,
    GiD_Vector,
    GiD_OnNodes, NULL, NULL, 0, NULL );

  for(NodesContainerType::iterator i_node = rNodes.begin();
      i_node != rNodes.end() ; ++i_node)
  {
    array_1d<double, 3>& temp =
      i_node->GetSolutionStepValue(rVariable, SolutionStepNumber);
    GiD_WriteVector( i_node->Id(), temp[0], temp[1], temp[2] );
  }

  GiD_EndResult();
}
```

In above examples the GiD interface has different rules for scalar and vectorial variables. Knowing the type of variable helps to implement customized versions of `WriteNodalResults` for each type of variable. This is an important feature of this interface which can handle exceptional cases for certain types and handle them with a uniform syntax for users:

```
// Writing temperature of all the nodes
gid_io.WriteNodalResults(TEMPERATURE, mesh.Nodes(), time, 0);
// Writing velocity of all the nodes
gid_io.WriteNodalResults(VELOCITY, mesh.Nodes(), time, 0);
```

Each variable as mentioned before has its name which can be accessed via `Name` method. This gives us necessary information to print the variable in output. In this way there is no need to pass the name of the variable as an argument by itself. Though This wouldn't be difficult, it keeps the simplicity of the interface.

6.6.4 Error Estimator

Writing an error estimator is another example of making a generic and reusable code using VBI. Here is an example of a simple recovery error estimator [104] implemented in a generic way:

```
virtual void EstimateError(const VariableType& ThisVariable,
                           ModelPart& rModelPart)
{
  mpRecovery->Recover(ThisVariable, rModelPart);

  double e_sum = double();

  typedef ModelPart::ElementIterator iterator_type;

  for(iterator_type element_iterator = rModelPart.ElementsBegin();
      element_iterator != rModelPart.ElementsEnd();
      ++element_iterator)
  {
    double error = CalculateError(ThisVariable,*element_iterator);
    element_iterator->GetValue(ERROR) = error;
    e_sum += error;
  }

  SetGlobalError(e_sum);

}

double CalculateError(const VariableType& ThisVariable,
                      Element& rThisElement)
{
  Element::NodesArrayType& element_nodes = rThisElement.Nodes();

  if(element_nodes.empty())
    return double();

  double result = 0.00;

  typedef Element::NodesArrayType::iterator iterator_type;

  for(iterator_type node_iterator = element_nodes.begin() ;
      node_iterator != element_nodes.end() ; ++node_iterator)
  {
    TDataType error = CalculateError(ThisVariable,
                                     rThisElement,
                                     *node_iterator);

    result += sqrt(error * error);
  }

  result *= rThisElement.GetGeometry()->Area();
  return result / element_nodes.size();
}

TDataType CalculateError(const VariableType& ThisVariable,
                         Element& rThisElement,
                         Node& rThisNode)
{
  TDataType result = rThisNode[ThisVariable];

  result -= rThisElement.Calculate(ThisVariable, rThisNode);
  return result;
}
```

In this manner the error estimator is not depended to the domain and can work in the same way with thermal flow or pressure gradient.

6.7 Problems and Difficulties

As usual nothing is perfect and the VBI is not excluded from that. There are some difficulties and problems still arising using this interface. Though they are not so important in some cases but in some other ones they need more attentions.

A problem is to store a variable which can be replaced with a component. Where can this happen? Any process which works over variables or their components in the same way and wants to store the variable or component causes some difficulties. A typical example is a process over some doubles, like calculating the norm, and applied to a double variable, like temperature, or a component of a vector, velocity_x, which also wants to keep a given variable to work on it after. In many cases this can be easily solved just by replacing the data type template parameter with a variable type parameter. The pattern can be same to the uniform template version of `GetValue` and `SetValue` methods.

But still there are some cases which above method cannot solve so easily. Storing the variable of a dof is a complex task. Assuming that each dof wants to store its variable and then search database to find its value for example to update the results:

```
// Updating
typedef EquationSystemType::DofsArrayType::iterator iterator_type;
for(iterator_type i_dof = equation_system.DofsBegin() ;
    i_dof != equation_system.DofsEnd() ; ++i_dof)
{
  if(i_dof->IsFree())
    i_dof->GetSolutionStepValue() =
      equation_system.GetResults()[i_dof->EquationId()];
}
```

In this case giving different template parameters to Dofs prevent us to use them in a normal array and using virtual functions is not allowed due inner loop usage of these methods in finite element code. So what to do? In Kratos an indexing mechanism is used to decide if the variable is a component or not and the some *traits* [99] are used to dispatch and select the proper procedure. This solution is not so clean and applying it is not so encapsulated yet. Further work to improve these tasks remains to be done in the future.

```
TDataType& GetReference(VariableData const& ThisVariable,
                        FixDataValueContainer& rData,
                        int ThisId)
{
  switch(ThisId)
  {
    KRATOS_DOF_TRAITS
  }
  KRATOS_ERROR(std::invalid_argument, "Not supported type for Dof" , "");
}
```