Adding a new analysis to Trellis.

1.0 Adding New Analysis Types to the Framework

This section discusses what must be done to add a new type of analysis to Trellis. This includes the definition of the problem information, defintion of the analysis and definition of elements to be used in the analysis. To demonstrate the work that is needed the implementation of classes to perform static and transient heat transfer will be discussed.

1.1 Defining the Analysis Class

For each type of analysis that you wish to perform you must define a class derived from FEAnalysis. In this context, a “type of analysis” corresponds to a statement of the weak form of the equations you are solving (i.e. if you want to solve the same problem using two different weak forms you must define two different analysis classes).

The purpose of the analysis class is to interpret the analysis attributes that exist on the model in a manner consistent with the chosen weak form and create objects (various system contributors such as elements, force contributors and constraints) that correspond to this weak form.

As an example two different, but related, analyses will be defined. The two analyses are for static and transient heat transfer. Since they share alot of common functionality, they will both be derived from a common base class (which is derived from FEAnalysis) called HeatTransferAnalysis.

1.1.1 Definition of Your Analysis class

Your analysis class should be publicly derived from FEAnalysis. It’s constructor should take in a pointer to an attribute case (AttCase) that describes the analysis to be performed. The pointer can be retrieved from the attribute manager as follows:

AttManager * mngr = AttManager::retrieve(atdbFileName);

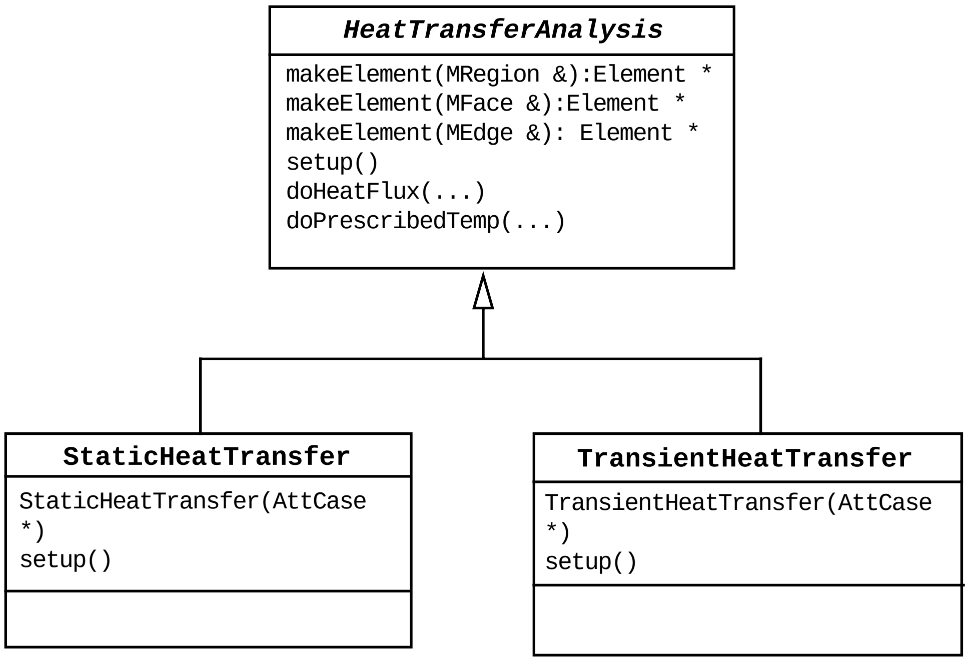
AttCase *theCase = mngr->findCase(CaseName);

where atdbFileName is the name of the attribute file that specifies the case CaseName.

In your class definition you should define a pointer to the correct type (Scalar, Vector) field for each of your primary fields. You will also need to override certain virtual functions in FEAnalysis which are described below.

In the example, most of this work is done in the HeatTransferAnalysis class.

```
class HeatTransferAnalysis : public FEAnalysis {
public:
```



HeatTransferAnalysis
makeElement(MRegion &):Element *
makeElement(MFace &):Element *
makeElement(MEdge &): Element *
setup()
doHeatFlux(...)
doPrescribedTemp(...)
StaticHeatTransfer
StaticHeatTransfer(AttCase *)
setup()
TransientHeatTransfer
TransientHeatTransfer(AttCase *)
setup()

FIGURE 1.

```
 HeatTransferAnalysis(AttCase *theCase);
 virtual ~HeatTransferAnalysis();

protected:
 virtual StiffnessContributor * makeElement(MRegion &meshRegion);
 virtual StiffnessContributor * makeElement(MFace &meshFace);
 virtual StiffnessContributor * makeElement(MEdge &meshEdge);

 virtual void setup();

 Field<DofScalar> *TemperatureField;
};
```

1.1.2 Member Functions of Your Analysis class

Your analysis class will likely want to override the following virtual functions in FEAnalysis:

```
void setup();
void solve();
Element * makeElement(MRegion &meshRegion);
Element * makeElement(MFace &meshFace);
Element * makeElement(MEdge &meshEdge);
```

void setup()

The setup() function is responsible for setting up the analysis. This is where most of the work in the class is done. The things that must be done here are:

1. Make fields corresponding to the primary variables for your analysis

   There are functions (makeScalarField(...) and makeVectorField(...) that will create the appropriate fields for you given the name of the field. The name of the field is used to retrieve further information about the field from the attribute file.

   ```
   TemperatureField = makeScalarField("temperature");
   ```

2. Make a renumbering scheme.
   The renumbering scheme used depends on the number of fields that are created from your analysis. Currently
   Renumberer *nq = new NodeQueue(mesh(), *TemperatureField);
   creates a renumberer for one field, and
   Renumberer *nq = new TwoFieldNodeQueue(mesh(), *firstField, *secondField);
   would create a renumberer for two fields. In the future there will be a renumberer that takes an array of fields as an argument, and will therefore work on any number of primary unknowns.

3. Make the correct form of DiscreteSystem

   The correct derived class of DiscreteSystem depends on the time order of your equations. There are currently two classes you can make: DiscreteSystemZeroOrder and DiscreteSystemFirstOrder.

   ```
   ds = new DiscreteSystemZeroOrder();
   ```

4. Create elements

   To do this simply call:

   ```
   FEAnalysis::setup();
   ```

   This will end up calling your makeElement(...) functions (described below) where elements need to be created. The default behavior is to call makeElement for any mesh entity without a higher order entity adjacent to it (all mesh regions, mesh faces without adjacent mesh regions, mesh edges without adjacent mesh faces). If you need different behavior there are a couple of approaches that can be used, ask.

5. Find the applicable boundary conditions on the model and make the appropriate force contributors and constraints

   This is the messiest part of the analysis class, although still fairly simple. Basically, what you need to do here is:

   a. get the geometric model from the mesh

      ```
      SGModel *gModel = mesh()->model();
      ```

   b. loop over all the entities in the model that might have attributes that you care about and check if they are there

In the example below, each edge in the model is checked to see if there is an attribute of the type “temperature”, if there is the member function doPrescribedTemperature(..) is called, passing the model edge and the attribute.

```
// look for essential boundary conditions
for(ie = 0; ie < gModel->numEdge(); ie++){
    GEdge ge = gModel->edge(ie);
    SSList<Attribute*> atts = ge->attributes("temperature");
    Attribute *att;
    for(SSListCIter<Attribute*> aIter(atts); aIter(att); )
        doPrescribedTemp(ge,(AttributeTensorOr1*)att);
}
```

c. When you’ve found an attribute you care about make the appropriate force contributor or constraint to reflect the attribute:

```
void HeatTransferAnalysis::doPrescribedTemp(GEdge ge, AttributeTensorOr0 *at)
{
    double temp = *at;
    MEdge *me;
    GEntity::MEdgeIter eIter = ge->firstMeshEdge(mesh());
    while(eIter(me)){
        Constraint *bc =
                new TemperatureBC(TemperatureField->getInterpolation(me),at);
        system()->add(bc);
    }
}
```

Here we loop over all the mesh edges on the model edge and make a new TemperatureBC object for each edge (the class for this object is described below). The object is then added to the DiscreteSystem.

StiffnessContributor * makeElement(MRegion &meshRegion);

StiffnessContributor * makeElement(MFace &meshFace);

StiffnessContributor * makeElement(MEdge &meshEdge);

These three routines are called by FEAnalysis::setup() to create the appropriate types of elements for a mesh entity that has been determined to need an element. How an element is defined is discussed below. What needs to be done here is to:

1. Create an interpolation for the element to use
2. Create the element, giving it the interpolation

An example for the heat transfer problem is given below:

```
StiffnessContributor * HeatTransferAnalysis::makeElement(MRegion &meshRegion)
{
  StiffnessContributor *e;
  Interpolation3d<DofScalar> *interp;
  interp = TemperatureField->createInterpolation(&meshRegion);
```

```
    e = new HeatTransferElement3d(interp);
    return e;
  }
```

1.2 Defining the Equation Contributors

1.2.1 StiffnessContributors

StiffnessContributors define coupling terms between degrees of freedom in the global set of equations. The basic functionality of an element is to form a matrix that represents the element contribution to the global system and pass this to an assembler object.

Elements are written in a manner independent of the type of interpolation.

Elements can contribute to various parts of the global system of equations. For an equation written in the form:

$$Mu'' + Cu' + Ku = f \tag{1}$$

we see that there are 3 global coupling matricies, M, C and K. Each element will have one or more member functions (du2,du1 and du0 respectively) that will calculate contributions to the corresponding global matrix. These functions are defined in the base class StiffnessContributor from which your element will be derived (actually, you will derive your element from one of the derived classes of StiffnessContributor: Stiffness1d, Stiffness2d or Stiffness3d). One function that all elements must provide is a function called dofs() that returns the list of degrees of freedom that the element couples together.

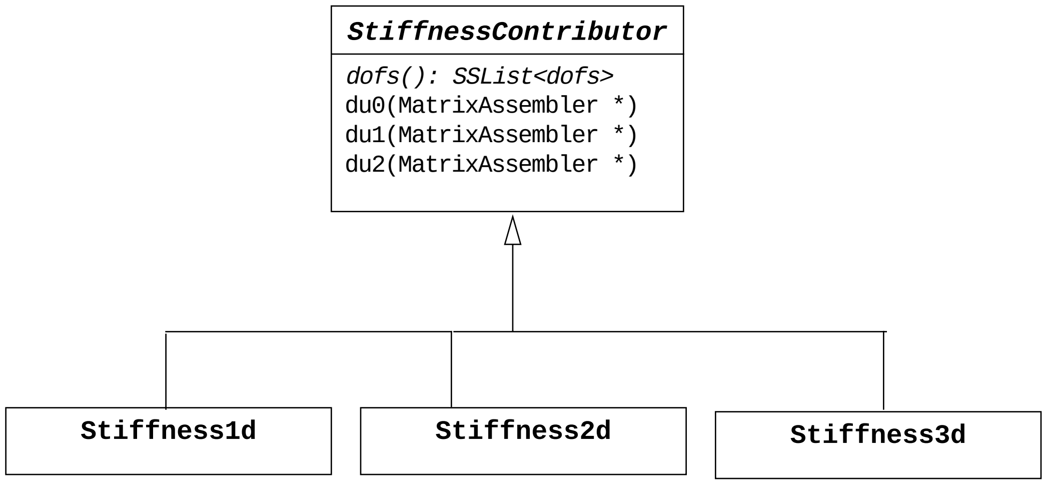


FIGURE 2.

A simple 2d heat transfer element is given below. This element works for both static and transient heat transfer analyses.

```
class HeatTransferElement2d : public Element2d {
public:
  HeatTransferSC2d(Interpolation2d<DofScalar> *interp) ;
  virtual SSList<DofRef> dofs() const;
  void du0(MatrixAssembler *a);
  void du1(MatrixAssembler *a);

  ElementMatrix ki(const SPoint2 &pt,int);
  ElementMatrix ci(const SPoint2 &pt,int);

protected:
  ConductiveMaterial *getMaterial();
  Interpolation2d<DofScalar> * Interp;
};
```

The only data that the element stores is a pointer to the interpolation over which it is active. The private member function getMaterial() retrieves a pointer to a conductive material that was associated with the geometric entity the current mesh entity is classified with. The constructor for this element simply takes in an interpolation and stores it.

```
HeatTransferSC2d::HeatTransferSC2d(Interpolation2d<DofScalar> *interp)
: Interp(interp)
{ }
```

The dofs() member function for this element is simple, it couples all of the degrees of freedom of the interpolation over which it is active. Since the interpolation has a member function to get this information this function is simply:

```
SSList<DofRef> HeatTransferSC2d::dofs() const
{ return Interp->dofs();}
```

The main work of the element is done in the du0() and ki() routines. The du0() routine must calculate the contribution of this element to the global K matrix. Here this is done by numerical integration over the element.

```
void HeatTransferSC2d::du0(MatrixAssembler *a)
{
  GaussQuadratureTri<ElementMatrix> integrator;
  Integrable2dObject<HeatTransferSC2d,ElementMatrix,DofScalar> integrand(this,&ki,Interp);
  int order = 2*(Interp->order()-1)+(Interp->mapOrder()-1);
  a->accept(integrator.integrate(integrand,order));
}
```

The ki() routine calculates the element matrix at a single integration point. In order to do this it must first retrieve the material information that is needed in this calculation. This particular element needs a conductive material formulation, and in this case a linear isotropic conductive material was coded. The interface for a typical material class is explained in detail somewhere else, for now it should be enough to know that the material has to provide a function that computes the material matrix C. To retrieve a pointer to the material the member function getMaterial has to be coded.

For this element formulation the elemental k matrix is:

$$(\nabla N)^T C \nabla N \tag{2}$$

where N are the interpolants of the unknown temperature field. This form is calculated as shown below.

```
ElementMatrix HeatTransferSC2d::ki(const SPoint2 &pt,int)
{
  ConductiveMaterial *mat = getMaterial();
  DofMatrix B = Gradient(*Interp,pt);
  ElementMatrix K = product(B,mat->C(),B);
  return K;
 }
```

1.2.2 Force Contributors

In addition to contributors to the coupling between degrees of freedom, there are also contributors to the right hand side, called force contributors (although in general the term on the right hand side might not really be a force).

For force contributors of the form:

$$\int_{\Gamma} N^T f \, d\Gamma \tag{3}$$

where N are the interpolants of the field, there are special classes that make it simple to write the force contributors. For the case of a 1d force (a force applied to an edge in a 2d problem) you simply must derive a class from SimpleForce1d. This derived class must override the member function attributeName() that returns the name of the attribute that describes the distribution of the force.

For example a class for heat fluxes applied to edges is written as:

```
class EdgeHeatFlux : public SimpleForce1d<DofScalar> {
public:
  EdgeHeatFlux(Interpolation1d<DofScalar> *interp);

protected:
  virtual SString attributeName() const;
};
```

The constructor for this class takes in the interpolation of the temperature field along that edge. That object is just passed up to the constructor of SimpleForce1d.

```
EdgeHeatFlux::EdgeHeatFlux(Interpolation1d<DofScalar> *interp)
: SimpleForce1d<DofScalar>(interp)
{
}
```

The attributeName function just returns a SString object that contains the name of the attribute, in this case “heat flux”. The base classes take care of all the rest

```
SString EdgeHeatFlux::attributeName() const
{ return SString("heat flux"); }
```

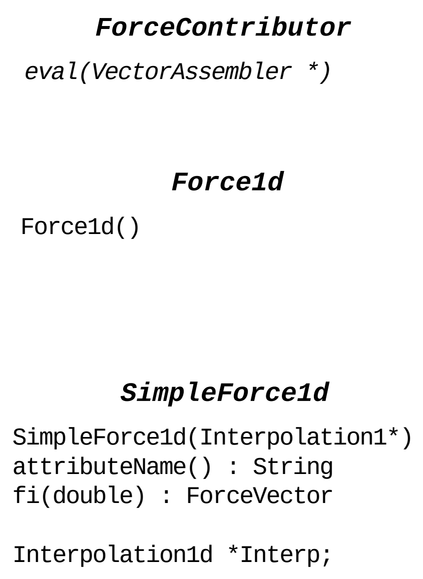


FIGURE 3.

1.2.3 Constraints

Constraints specify the values of degrees of freedom along a certain mesh entity. Their defintion and use is very similar to that of force contributors. There is a similar class hierarchy to that of ForceContributors and a similar class (SimpleEssentialBC) for the simple case of when the field must just be equal to a certain prescribed value along the mesh entity.

There is also an even simpler class for when a constraint is simply zero along the mesh entity. This is the ZeroBC class.

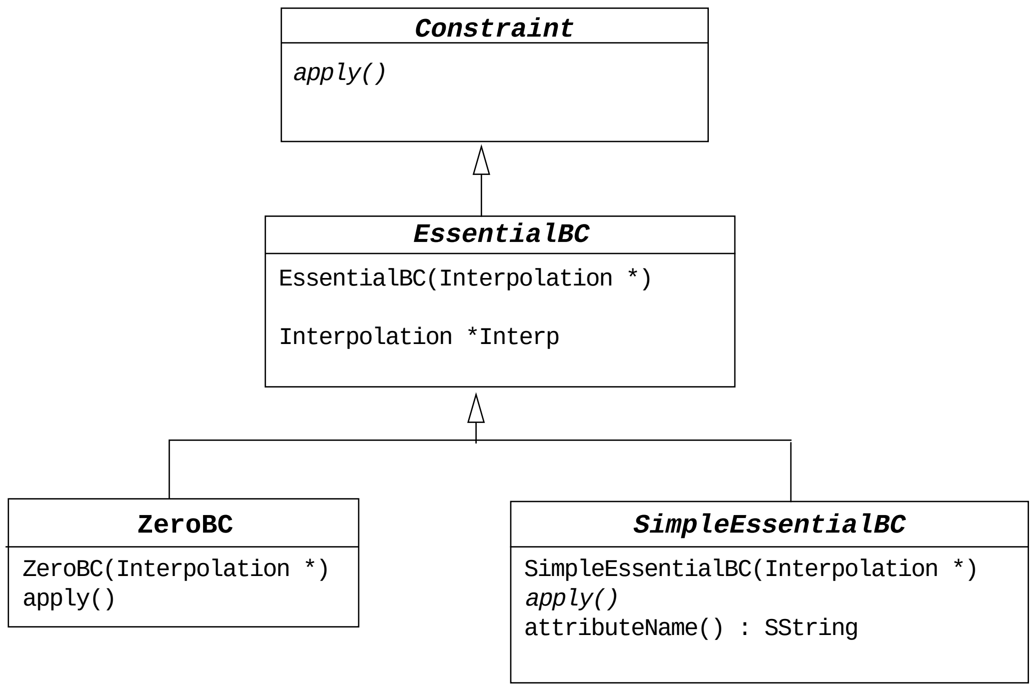

Constraint
apply()
EssentialBC
EssentialBC(Interpolation *)
Interpolation *Interp
ZeroBC
ZeroBC(Interpolation *)
apply()
SimpleEssentialBC
SimpleEssentialBC(Interpolation *)
apply()
attributeName() : SString

FIGURE 4.