Refactoring of StdRegions/LocalRegions for generalised basis handling and coprocessor platforms

September 4, 2015

1 Objective

The current code structure provides limited scope for supporting arbitrary types and forms of elemental expansion bases. Modal bases are considered the default since they are frequently used and support is, in general, good. Currently nodal bases are bolted onto the modal-focused classes in a less efficient manner making it difficult to extend our existing code to other non-modal bases.

More generally moving forward, we would like to improve the robustness of the library classes and support other platforms such as GPUs and Phis.

The aim is to restructure the StdRegions and LocalRegions libraries to generically support both nodal and modal bases efficiently and incorporate scope within the library architecture to include additional bases as they are developed or desired. At the core of this plan is a separation of the data and algorithms within the elemental regions, which are currently lumped together within the same StdRegions and LocalRegions classes.

2 Proposed solution

Perform the following changes:

- Create a hierarchy of expansion bases, which encapsulate a (potentially multi-dimensional) basis definition. As at present, these would couple to a (potentially multi-dimensional) point distribution.
- Within StdRegions, create a library of operators (factory- instantiated) which may be generic across basis types or potentially specialised to a given basis or class of bases.
- Within StdRegions, create a library of platform-aware containers for elemental reference shapes (factory-instantiated), primarily storing the coefficients and physical solution.

- Within LocalRegions, create a library of operators which extend the StdRegions operators to incorporate geometry.
- Within LocalRegions, create a library of containers for physical elements, which extend the StdRegions counterparts by augmenting it with geometry information.
- Within Collections, leverage operator implementations within the StdRegions library to remove duplication of operator implementation and enable support for a wider range of bases (currently nodal is not properly supported).

3 Basis Definition

Basis components would be constructed as a class hierarchy and instantiated through a factory, so as to remove the enum definition currently in place. This will allow the addition of arbitrary bases without needing to modify the library.

Example derived classes, from the BasisComponent base class, might include:

- ModifiedA
- ModifiedB
- ModifiedC
- OrthoA
- OrthoB
- OrthoC
- Bernstein
- BernsteinTri
- BernsteinTet
- Lagrange

Construction parameters include

- a map of (key, double-precision value) parameters
- points distribution + optional index (if basis dim $<$ points dim)
- OR two point distributions
- OR three point distributions

BasisComponents may themselves be 1D, 2D or 3D.

3.1 Main Basis class

The purpose of this class is to:

- Provide scope for storing a richer description of the basis and its capabilities.
- Support tensor-product and multi-dimensional bases and point distributions within a single object entity.
- Remove the varying number of constructor parameters in elements to enable easier use of the factory pattern.

The class would be instantiated from one, two or three BasisComponent classes which are treated in a tensor-product style. For example:

```
Basis(BasisComponentSharedPtr,
      BasisComponentSharedPtr = 0,
      BasisComponentSharedPtr = 0)
```

Note that a single BasisComponent may be multidimensional, so this class must ensure the total product does not exceed three dimensions. In addition, for bases containing partial PointsKeys (i.e. a multi-dimensional points-key + index), this class will verify that the complete PointsKey is defined across the specified BasisComponents.

4 Element container classes

In StdRegions, these would take a suitable Basis as a constructor argument and store the vectors of coefficients and the physical solution.

- They would also include the auxiliary functions such as GetFaceBasisKey, GetEdgeNumPoints, FaceToElementMap, GetEdgePhysVals, etc as these are related to the container storage.
- The functions such as GetNumVerts are geometry/shape-related and so make more sense to be in the corresponding Geometry class in SpatialDomains.
- For the LocalRegions counterparts, they would store in addition a Geometry object and auxiliary data such as normals.
- Containers should be aware of their state (i.e. Coeff, Phys) and consistency between Coeff and Phys representations.
- Containers can be designed to interoperate with different platforms/co-processors (e.g. GPU/Phi) and manage data movement.

GPU and Phi support should be compile-time options. Therefore support for such enabled containers should be confined to separate classes.

5 Operator classes

Only a single operator object is necessary to act on any element container of a given type/basis.

Operators would be registered with a factory. Keys would specify the bases for which the operator is valid. An operator may be registered for multiple bases where the action is appropriately generic (e.g. a BwdTrans). Operators would be synomymous with their matrix equivalents. An operator class should therefore implement:

- a GetMatrix function, which implements (if appropriate) a matrix representation of the operator. An operator class would manage the matrices it generates.
- an Apply function, which implements a matrix-free approach for the operator (or it can operate with the matrix if that is just as efficient).
- Operator implementations can be written for co-processor support.

Again, separate implementations of operators should be written for GPU/Phi support to allow for simplified compile-time selection.

6 Example

```
typedef std::string TBasisParamKey;
typedef NekDouble TBasisParamValue;
typedef std::map<TBasisParamKey, TBasisParamValue> TBasisParamList;

// Basis parameters (order + e.g. Jacobi parameters)
TBasisParamList p;
p["Order"]=4;
//p["alpha"]=0;
//p["beta"]=0;

// Modified Basis Uniform Quad
LibUtilities::Points<NekDouble> pts_gll
    = factory.CreateInstance(GaussLobattoLegendre,5);
LibUtilities::ModifiedA b_modA(p, pts_gll);
LibUtilities::Basis     b0    (b_modA, b_modA); // e.g. Quad

// Uniform Hex
LibUtilities::Basis     b1    (b_modA, b_modA, b_modA); // e.g. Hex

// 2D Fekete points with Lagrange for Tri
LibUtilities::Points<NekDouble> pts_fekete
    = factory.CreateInstance(FeketeTri,15);

LibUtilities::Lagrange2 b_lag (p, pts_fekete);
LibUtilities::Basis     b2    (b_lag);

// Tri modified
LibUtilities::Points<NekDouble> pts_gr
    = factory.CreateInstance(GaussRadau,4);
LibUtilities::ModifiedB b_modB(p, pts_gr);
LibUtilities::Basis     b3    (b_modA, b_modB);

// Prism using Fekete/Lagrange on the tri base and modA
LibUtilities::Basis     b3    (b_lag, b_modA);

// Create some elements
std::vector<LocalRegions::ExpansionSharedPtr> explist;
for (i = 0; i < nel; ++i)
{
    SpatialDomains::GeometrySharedPtr g
            = Factory.CreateInstance(eQuad, ...)
    LocalRegions::ExpansionSharedPtr e
            = Factory.CreateInstance(eQuad, b1, g);
```

```
    explist.push_back(e)
}

// Do a BwdTrans
Operators::OpKey opKey("BwdTrans", b1);
Operators::OpSharedPtr OpBwdTrans = Factory.CreateInstance(opKey);
for (i = 0; i < nel; ++i)
{
    OpBwdTrans->Apply(expList[i], expList[i]);
}
```