Nektar++: Spectral/hp Element Framework

Version

Developer Guide

January 17, 2018

Department of Aeronautics, Imperial College London, UK
Scientific Computing and Imaging Institute, University of Utah, USA

Contents

Introduction

Nektar++ [**?**] is a tensor product based finite element package designed to allow one to construct efficient classical low polynomial order h-type solvers (where h is the size of the finite element) as well as higher p-order piecewise polynomial order solvers. The framework currently has the following capabilities:

- Representation of one, two and three-dimensional fields as a collection of piecewise continuous or discontinuous polynomial domains.
- Segment, plane and volume domains are permissible, as well as domains representing curves and surfaces (dimensionally-embedded domains).
- Hybrid shaped elements, i.e triangles and quadrilaterals or tetrahedra, prisms and hexahedra.
- Both hierarchical and nodal expansion bases.
- Continuous or discontinuous Galerkin operators.
- Cross platform support for Linux, Mac OS X and Windows.

The framework comes with a number of solvers and also allows one to construct a variety of new solvers.

Our current goals are to develop:

- Automatic auto-tuning of optimal operator implementations based upon not only h and p but also hardware considerations and mesh connectivity.
- Temporal and spatial adaption.
- Features enabling evaluation of high-order meshing techniques.

This document provides implementation details for the design of the libraries, Nektar++-specific data structures and algorithms and other development information.

Warning
This document is still under development and may be incomplete in parts.

For further information and to download the software, visit the Nektar++ website at `http://www.nektar.info`.

Chapter 1

Core Concepts

This section describes some of the key concepts which are useful when developing code within the Nektar++ framework.

1.1 Factory method pattern

The factory method pattern is used extensively throughout Nektar++ as a mechanism to instantiate objects. It provides the following benefits:

- Encourages modularisation of code such that conceptually related algorithms are grouped together
- Structuring of code such that different implementations of the same concept are encapsulated and share a common interface
- Users of a factory-instantiated modules need only be concerned with the interface and not the details of underlying implementations
- Simplifies debugging since code relating to a specific implementation resides in a single class
- The code is naturally decoupled to reduce header-file dependencies and improves compile times
- Enables implementations (e.g. relating to third-party libraries) to be disabled through the build process (CMake) by not compiling a specific implementation, rather than scattering preprocessing statements throughout the code

For conceptual details see the Wikipedia page.

1.1.1 Using NekFactory

The templated NekFactory class implements the factory pattern in Nektar++. There are two distinct aspects to creating a factory-instantiated collection of classes: defining the public interface, and registering specific implementations. Both of these involve adding standard boilerplate code. It is assumed that we are writing a code which implements a particular concept or functionality within the code, for which there are multiple implementations. The reasons for multiple implementations may be very low level such as alternative algorithms for solving a linear system, or high level, such as selecting from a range of PDEs to solve.

1.1.1.1 Creating an interface (base class)

A base class must be defined which prescribes an implementation-independent interface. In Nektar++, the template method pattern is used, requiring public interface functions to be defined which call private virtual implementation methods. The latter will be overridden in the specific implementation classes. In the base class these virtual methods should be defined as pure virtual, since there is no implementation and we will not be instantiating this base class explicitly.

As an example we will create a factory for instantiating different implementations of some concept `MyConcept`, defined in `MyConcept.h` and `MyConcept.cpp`. First in `MyConcept.h`, we need to include the NekFactory header

```
#include <LibUtilities/BasicUtils/NekFactory.hpp>
```

The following code should then be included just before the base class declaration (in the same namespace as the class):

```
class MyConcept

// Datatype for the MyConcept factory
typedef LibUtilities::NekFactory< std::string, MyConcept,
            ParamType1,
            ParamType2 > MyConceptFactory;
MyConceptFactory& GetMyConceptFactory();
```

The template parameters define the datatype of the key used to retrieve a particular implementation (usually a string, enum or custom class such as `MyConceptKey`, the base class (in our case `MyConcept` and a list of zero or more parameters which are taken by the constructors of all implementations of the type `MyConcept` (in our case we have two). Note that all implementations must take the same parameter list in their constructors.

The normal definition of our base class then follows:

```
class MyConcept
{
    public:
        MyConcept(ParamType1 p1, ParamType2 p2);
        ...
```

```
};
```

We must also define a shared pointer for our base class for use later

```
typedef boost::shared_ptr<MyConcept> MyConceptShPtr;
```

1.1.1.2 Creating a specific implementation (derived class)

A class is defined for each specific implementation of a concept. It is these specific implementations which are instantiated by the factory.

In our example we will have an implementations called `MyConceptImpl1` defined in `MyConceptImpl1.h` and `MyConceptImpl1.cpp`. In the header file we include the base class header file

```
#include <Subdir/MyConcept.h>
```

We then define the derived class as normal:

```
class MyConceptImpl1 : public MyConcept
{
...
};
```

In order for the factory to work, it must know

- that MyConceptImpl1 exists, and
- how to create it.

To allow the factory to create instances of our class we define a function in our class:

```
/// Creates an instance of this class
static MyConceptSharedPtr create(
                ParamType1 p1,
                ParamType2 p2)
{
    return MemoryManager<MyConceptImpl1>::AllocateSharedPtr(p1, p2);
}
```

This function simply creates an instance of `MyConceptImpl1` using the supplied parameters. It must be `static` because we are not operating on an existing instance and it should return a base class shared pointer (rather than a `MyConceptImpl1` shared pointer), since the point of the factory is that the calling code does not know about specific implementations.

The last task is to register our implementation with the factory. This is done using the `RegisterCreatorFunction` member function of the factory. However, we wish this to

happen as early on as possible (so we can use the factory straight away) and without needing to explicitly call the function for every implementation at the beginning of our program (since this would again defeat the point of a factory)! The solution is to use the function to initialise a static variable: it will be executed prior to the start of the `main()` routine, and can be located within the very class it is registering, satisfying our code decoupling requirements.

In `MyConceptImpl1.h` we define a static variable with the same datatype as the key used in our factory (in our case `std::string`)

```
1 static std::string className;
```

The above variable can be `private` since it is typically never actually used within the code. We then initialise it in `MyConceptImpl1.cpp`

```
1 string MyConceptImpl1::className
2         = GetMyConceptFactory().RegisterCreatorFunction(
3                                     "Impl1",
4                                     MyConceptImpl1::create,
5                                     "First implementation of my concept.");
```

The first parameter specifies the value of the key which should be used to select this implementation. The second parameter is a function pointer to our static function used to instantiate our class. The third parameter provides a description which can be printed when listing the available MyConcept implementations.

1.1.2 Instantiating classes

To create instances of MyConcept implementations elsewhere in the code, we must first include the ”base class” header file

```
1 #include <Subdir/MyConcept.h>
```

Note we do not include the header files for the specific MyConcept implementations anywhere in the code (apart from `MyConceptImpl1.cpp`). If we modify the implementation, only the implementation itself requires recompiling and the executable relinking.

We create an instance by retrieving the `MyConceptFactory` and call the `CreateInstance` member function of the factory:

```
1 ParamType p1 = ...;
2 ParamType p2 = ...;
3 MyConceptShPtr p = GetMyConceptFactory().CreateInstance( "Impl1", p1, p2 );
```

Note that the class is used through the pointer `p`, which is of type `MyConceptShPtr`, allowing the use of any of the public interface functions in the base class (and therefore the specific implementations behind them) to be called, but not directly any functions declared solely in a specific implementation.

1.2 NekArray

An Array is a thin wrapper around native arrays. Arrays provide all the functionality of native arrays, with the additional benefits of automatic use of the Nektar++ memory pool, automatic memory allocation and deallocation, bounds checking in debug mode, and easier to use multi-dimensional arrays.

Arrays are templated to allow compile-time customization of its dimensionality and data type.

Parameters:

- `Dim` Must be a type with a static unsigned integer called `Value` that specifies the array's dimensionality. For example

```
struct TenD {
    static unsigned int Value = 10;
};
```

- `DataType` The type of data to store in the array.

It is often useful to create a class member Array that is shared with users of the object without letting the users modify the array. To allow this behavior, Array<Dim, !DataType> inherits from Array<Dim, const !DataType>. The following example shows what is possible using this approach:

```
class Sample {
    public:
        Array<OneD, const double>& getData() const { return m_data; }
        void getData(Array<OneD, const double>& out) const { out = m_data; }

    private:
        Array<OneD, double> m_data;
};
```

In this example, each instance of Sample contains an array. The getData method gives the user access to the array values, but does not allow modification of those values.

1.2.1 Efficiency Considerations

Tracking memory so it is deallocated only when no more Arrays reference it does introduce overhead when copying and assigning Arrays. In most cases this loss of efficiency is not noticeable. There are some cases, however, where it can cause a significant performance penalty (such as in tight inner loops). If needed, Arrays allow access to the C-style array through the `Array::data` member function.

1.3 Threading

Note

Threading is not currently included in the main code distribution. However, this hybrid MPI/pthread functionality should be available within the next few months.

We investigated adding threaded parallelism to the already MPI parallel Nektar++. MPI parallelism has multiple processes that exchange data using network or network-like communications. Each process retains its own memory space and cannot affect any other process's memory space except through the MPI API. A thread, on the other hand, is a separately scheduled set of instructions that still resides within a single process's memory space. Therefore threads can communicate with one another simply by directly altering the process's memory space. The project's goal was to attempt to utilise this difference to speed up communications in parallel code.

A design decision was made to add threading in an implementation independent fashion. This was achieved by using the standard factory methods which instantiate an abstract thread manager, which is then implemented by a concrete class. For the reference implementation it was decided to use the Boost library rather than native p-threads because Nektar++ already depends on the Boost libraries, and Boost implements threading in terms of p-threads anyway.

It was decided that the best approach would be to use a thread pool. This resulted in the abstract classes ThreadManager and ThreadJob. ThreadManager is a singleton class and provides an interface for the Nektar++ programmer to start, control, and interact with threads. ThreadJob has only one method, the virtual method run(). Subclasses of ThreadJob must override run() and provide a suitable constructor. Instances of these subclasses are then handed to the ThreadManager which dispatches them to the running threads. Many thousands of ThreadJobs may be queued up with the ThreadManager and strategies may be selected by which the running threads take jobs from the queue. Synchronisation methods are also provided within the ThreadManager such as wait(), which waits for the thread queue to become empty, and hold(), which pauses a thread that calls it until all the threads have called hold(). The API was thoroughly documented in Nektar++'s existing Javadoc style.

Classes were then written for a concrete implementation of ThreadManager using the Boost library. Boost has the advantage of being available on all Nektar++'s supported platforms. It would not be difficult, however, to implement ThreadManager using some other functionality, such as native p-threads.

Two approaches to utilising these thread classes were then investigated. The bottom-up approach identifies likely regions of the code for parallelisation, usually loops around a simple and independent operation. The top-down approach seeks to run as much of the

code as is possible within a threaded environment.

The former approach was investigated first due to its ease of implementation. The operation chosen was the multiplication of a very large sparse block diagonal matrix with a vector, where the matrix is stored as its many smaller sub matrices. The original algorithm iterated over the sub matrices multiplying each by the vector and accumulating the result. The new parallel algorithm sends ThreadJobs consisting of batches of sub matrices to the thread pool. The worker threads pick up the ThreadJobs and iterate over the sub matrices in the job accumulating the result in a thread specific result vector. This latter detail helps to avoid the problem of cache ping-pong which is where multiple threads try to write to the same memory location, repeatedly invalidating one another's caches.

Clearly this approach will work best when the sub matrices are large and there are many of them . However, even for test cases that would be considered large it became clear that the code was still spending too much time in its scalar regions.

This led to the investigation of the top-down approach. Here the intent is to run as much of the code as possible in multiple threads. This is a much more complicated approach as it requires that the overall problem can be partitioned suitably, that a mechanism be available to exchange data between the threads, and that any code using shared resources be thread safe. As Nektar++ already has MPI parallelism the first two requirements (data partitioning and exchange) are already largely met. However since MPI parallelism is implemented by having multiple independent processes that do not share memory space, global data in the Nektar++ code, such as class static members or singleton instances, are now vulnerable to change by all the threads running in a process.

To Nektar++'s communication class, Comm, was added a new class, ThreadedComm. This class encapsulates a Comm object and provides extra functionality without altering the API of Comm (this is the Decorator pattern). To the rest of the Nektar++ library this Comm object behaves the same whether it is a purely MPI Comm object or a hybrid threading plus MPI object. The existing data partitioning code can be used with very little modification and the parts of the Nektar++ library that exchange data are unchanged. When a call is made to exchange data with other workers ThreadedComm first has the master thread on each process (i.e. the first thread) use the encapsulated Comm object (typically an MPI object) to exchange the necessary data between the other processes, and then exchanges data with the local threads using direct memory to memory copies.

As an example: take the situation where there are two processes A and B, possibly running on different computers, each with two threads 1 and 2. A typical data exchange in Nektar++ uses the Comm method AllToAll(...) in which each worker sends data to each of the other workers. Thread A1 will send data from itself and thread A2 via the embedded MPI Comm to thread B1, receiving in turn data from threads B1 and B2. Each thread will then pick up the data it needs from the master thread on its process

using direct memory to memory copies. Compared to the situation where there are four MPI processes the number of communications that actually pass over the network is reduced. Even MPI implementations that are clever enough to recognise when processes are on the same host must make a system call to transfer data between processes.

The code was then audited for situations where threads would be attempting to modify global data. Where possible such situations were refactored so that each thread has a copy of the global data. Where the original design of Nektar++ did not permit this access to global data was mediated through locking and synchronisation. This latter approach is not favoured except for global data that is used infrequently because locking reduces concurrency.

The code has been tested and Imperial College cluster cx1 and has shown good scaling. However it is not yet clear that the threading approach outperforms the MPI approach; it is possible that the speedups gained through avoiding network operations are lost due to locking and synchronisation issues. These losses could be mitigated through more in-depth refactoring of Nektar++.

Chapter 2

Library Design

A major challenge which arises when one aims to develop a software package that implements the spectral/hp element method is to implement the mathematical structure of the method in a digestible and coherent matter. Obviously, there are many ways to encapsulate the fundamental concepts related to the spectral/*hp* element method, depending on e.g. the intended goal of the developer or the chosen programming language. We will (without going in too much detail) give a an overview of how we have chosen to abstract and implement spectral/hp elements in the *Nektar++* library. However, we want to emphasise that this is not the only possible choice.

Five different sublibraries, employing this characteristic pattern, are provided in the full *Nektar++* library:

- the supporting utilities sublibrary (LibUtilities library)
- the standard elemental region sublibrary (StdRegions library)
- the parametric mapping sublibrary (SpatialDomains library)
- the local elemental region sublibrary (LocalRegions library)
- the global region sublibrary (MultiRegions library)
- the solver support sublibrary (SolverUtils library)

This structure can also be related to the formulation of a global spectral/hp element expansion, i.e.

$$u(\boldsymbol{x}) = \overbrace{\sum_{e\in\mathcal{E}} \underbrace{\sum_{n\in\mathcal{N}} \phi_n^e(\boldsymbol{x})\hat{u}_n^e}_{\text{LocalRegions library}}}^{\text{MultiRegions library}} = \underbrace{\sum_{e\in\mathcal{E}} \sum_{n\in\mathcal{N}} \phi_n^{std} \overbrace{\left([\chi^e]^{-1}(\boldsymbol{x})\right)}^{\text{SpatialDomains library}} \hat{u}_n^e}_{\text{StdRegions library}}$$

A more detailed overview of the *Nektar++* structure is given in Figure 2.1. A diagram showing the most important classes in the core sub-libraries, is depicted in the Figure 2.2.

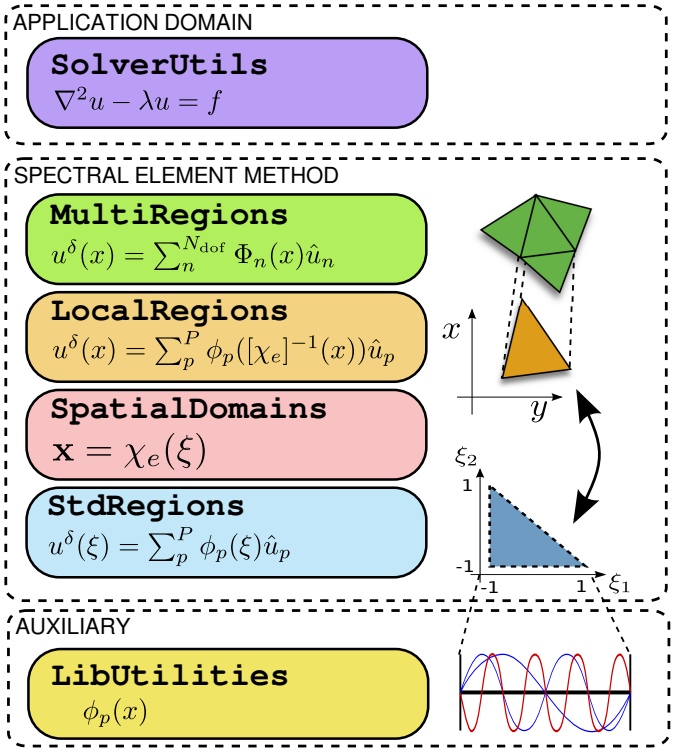


Figure 2.1 Structural overview of the Nektar++ libraries

2.1 LibUtilities

This contains the underlying building blocks for constructing a spectral element formulation including linear algebra, polynomial routines and memory management [**?**, **?**, **?**, **?**, **?**]. This includes:

- Basic Constants
- Basic Utilities (Nektar++ Arrays)
- Expression Templates
- Foundations
- Interpreter
- Kernel
- Linear Algebra
- Memory Management
- Nodal Data

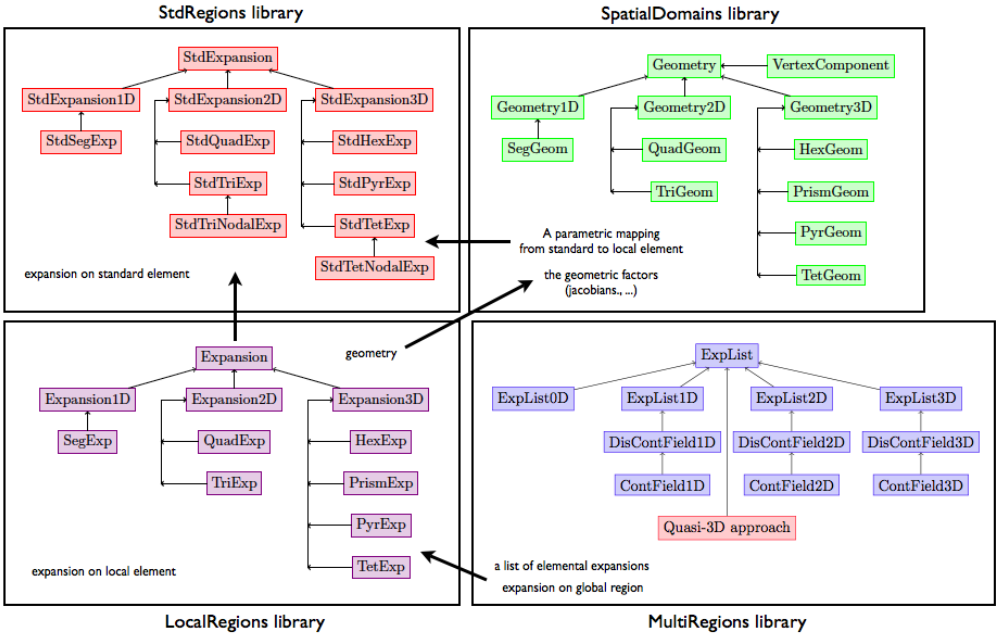


Figure 2.2 Diagram of the important classes in each library.

- Polynomial Subroutines

- Time Integration

2.1.1 The Polylib library

These are routines for orthogonal polynomial calculus and interpolation based on codes by Einar Ronquist and Ron Henderson.

2.1.1.1 Abbreviations

Character(s)	Description
z	Set of collocation/quadrature points
w	Set of quadrature weights
D	Derivative matrix
h	Lagrange Interpolant
I	Interpolation matrix
g	Gauss
k	Kronrod
gr	Gauss-Radau
gl	Gauss-Lobatto
j	Jacobi
m	point at minus 1 in Radau rules
p	point at plus 1 in Radau rules

2.1.1.2 Main routines

Points and Weights	
Routine	Description
zwgj	Compute Gauss-Jacobi points and weights
zwgrjm	Compute Gauss-Radau-Jacobi points and weights (z=-1)
zwgrjp	Compute Gauss-Radau-Jacobi points and weights (z= 1)
zwglj	Compute Gauss-Lobatto-Jacobi points and weights
zwgk	Compute Gauss-Kronrod-Jacobi points and weights
zwrk	Compute Radau-Kronrod points and weights
zwlk	Compute Lobatto-Kronrod points and weights
JacZeros	Compute Gauss-Jacobi points and weights

Derivative Matrices	
Routine	Description
Dgj	Compute Gauss-Jacobi derivative matrix
Dgrjm	Compute Gauss-Radau-Jacobi derivative matrix (z=-1)
Dgrjp	Compute Gauss-Radau-Jacobi derivative matrix (z= 1)
Dglj	Compute Gauss-Lobatto-Jacobi derivative matrix

Lagrange Interpolants	
Routine	Description
hgj	Compute Gauss-Jacobi Lagrange interpolants
hgrjm	Compute Gauss-Radau-Jacobi Lagrange interpolants (z=-1)
hgrjp	Compute Gauss-Radau-Jacobi Lagrange interpolants (z= 1)
hglj	Compute Gauss-Lobatto-Jacobi Lagrange interpolants

Interpolation Operators	
Routine	Description
Imgj	Compute interpolation operator gj->m
Imgrjm	Compute interpolation operator grj->m (z=-1)
Imgrjp	Compute interpolation operator grj->m (z= 1)
Imglj	Compute interpolation operator glj->m

Polynomial Evaluation	
Routine	Description
jacobfd	Returns value and derivative of Jacobi poly. at point z
jacobd	Returns derivative of Jacobi poly. at point z (valid at z=-1,1)

2.1.1.3 Local routines

Routine	Description
jacobz	Returns Jacobi polynomial zeros
gammaf	Gamma function for integer values and halves
RecCoeff	Calculates the recurrence coefficients for orthogonal poly
TriQL	QL algorithm for symmetrix tridiagonal matrix
JKMatrix	Generates the Jacobi-Kronrod matrix

2.1.1.4 Notes

- Legendre polynomial $\alpha = \beta = 0$
- Chebychev polynomial $\alpha = \beta = -0.5$
- All routines are double precision.
- All array subscripts start from zero, i.e. vector $0..N-1$

2.2 StdRegions

The StdRegions library, a summary of which is shown in Figure 2.3, bundles all classes that mimic a spectral/*hp* element expansion on a standard region. Such an expansion, i.e.

$$u(\boldsymbol{\xi}_i) = \sum_{n \in \mathcal{N}} \phi_n(\boldsymbol{\xi}_i)\hat{u}_n,$$

can be encapsulated in a class that essentially should only contain three data structures, respectively representing:

- the coefficient vector $\hat{\boldsymbol{u}}$,

- the discrete basis matrix $\boldsymbol{B}$, and
- the vector $\boldsymbol{u}$ which represents the value of the expansion at the quadrature points $\boldsymbol{\xi}_i$.

All standard expansions, independent of the dimensionality or shape of the standard region, can be abstracted in a similar way. Therefore, it is possible to define these data structures in an *abstract* base class, i.e. the class !StdExpansion. This base class can also contain the implementation of methods that are identical across all shapes. Derived from this base class is another level of abstraction, i.e. the abstract classes StdExpansion1D, StdExpansion2D and StdExpansion3D. All other shape-specific classes (such as e.g. StdSegExp or StdQuadExp) are inherited from these abstract base classes. These shape-specific classes are the classes from which objects will be instantiated. They also contain the shape-specific implementation for operations such as integration or differentiation.

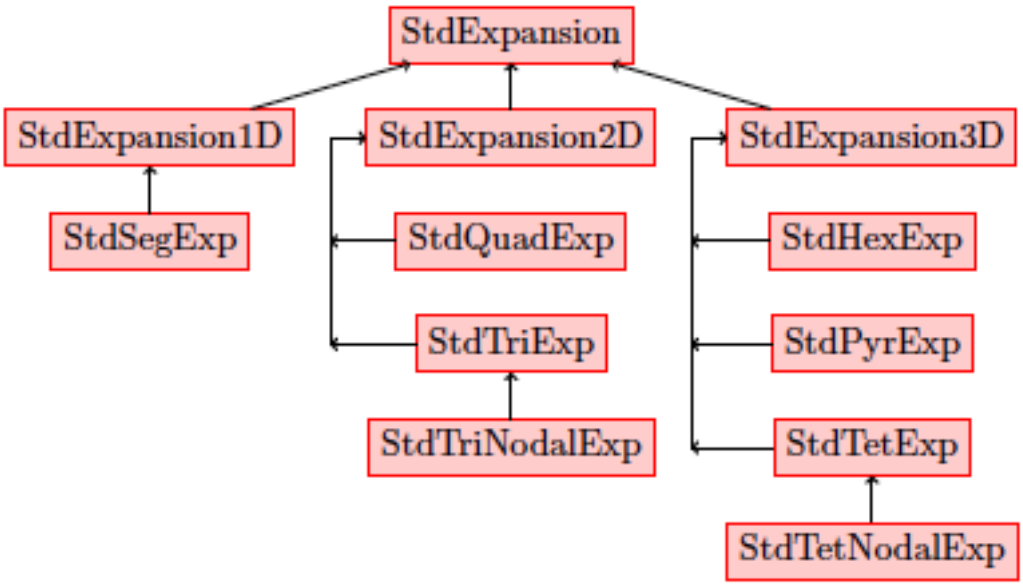


Figure 2.3 Main classes in the StdRegions library.

2.3 SpatialDomains

The most important family of classes in the SpatialDomains library is the Geometry family, as can also be seen in Figure 2.4. These classes are the representation of a (geometric) element in *physical space*. It has been indicated before that every local

element can be considered as an image of the standard element where the corresponding one-to-one mapping can be represented as an elemental standard spectral/hp expansion. As such, a proper encapsulation should at least contain data structures that represent such an expansion in order to completely define the geometry of the element. Therefore, we have equipped the classes in the Geometry family with the following data structures:

- an object of !StdExpansion class, and
- a data structure that contains the metric terms (Jacobian, derivative metrics) of the transformation.

Note that although the latter data structure is not necessary to define the geometry, it contains information inherent to the iso-parametric representation of the element that can later be used in e.g. the LocalRegions library. Again, the StdExpansion object can be defined in the abstract base class Geometry. However, for every shape-specific geometry class, it needs to be initialised according to the corresponding StdRegions class (e.g. for the QuadGeom class, it needs to be initialised as an StdQuadExp object).

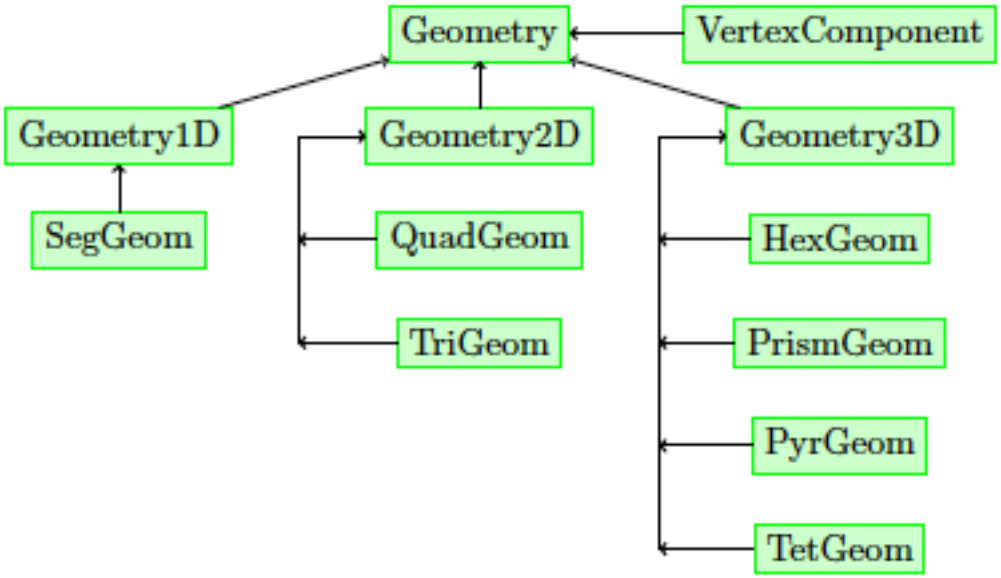


Figure 2.4 Main classes in the SpatialDomains library.

2.4 LocalRegions

The LocalRegions library is designed to encompass all classes that encapsulate the elemental spectral/hp expansions in physical space, see also the figure below. It can be

appreciated that such a local expansion essentially is a standard expansion that has a (in C++ parlance) additional coordinate transformation that maps the standard element to the local element. In an object-oriented context, these is-a and has-a relationships can be applied as follows: the classes in the !LocalRegions library are derived from the !StdExpansion class tree but they are supplied with an additional data member representing the geometry of the local element. Depending on the shape-specific class in the !LocalRegions library, this additional data member is an object of the corresponding class in the Geometry class structure. This inheritance between the !LocalRegions and !StdRegions library also allows for a localised implementation that prevents code duplication. In order to e.g. evaluate the integral over a local element, the integrand can be multiplied by the Jacobian of the coordinate transformation, where after the evaluation is redirected to the !StdRegions implementation.

This provides extensions of the spectral element formulation into the world. It provides spatially local forms of the reference space expansions through a one-to-one linear mapping from a standard straight-sided region to the physical space, based on the vertices.

2.4.1 Local Mappings

2.4.1.1 Linear Mappings

In one dimension this has the form

$$x = \chi(\xi) = \frac{1-\xi}{2}x_{e-1} + \frac{1+\xi}{2}x_e \quad \xi\Omega^e$$

In two dimensions, for a quadrilateral, each coordinate is given by

$$\begin{aligned} x_i = \chi(\xi_1,\xi_2) = x_i^A \frac{1-\xi_1}{2}\frac{1-\xi_2}{2} + x_i^B \frac{1+\xi_1}{2}\frac{1-\xi_2}{2} \\ + x_i^D \frac{1-\xi_1}{2}\frac{1+\xi_2}{2} + x_i^C \frac{1+\xi_1}{2}\frac{1+\xi_2}{2}, \quad i = 1,2 \end{aligned}$$

2.4.1.2 Curvilinear mappings

The mapping can be extended to curved-sided regions through the use of an iso-parametric representation. In contrast to the linear mapping, where only information about the vertices of the element were required, a curvilinear mapping requires information about the shape of each side. This is provided by shape functions, $f^A(\xi_1), f^B(\xi_2), f^C(\xi_1)$ and $f^D(\xi_2)$, in the local coordinate system. For example, the linear blending function is given by

$$\begin{aligned} x_i = \chi_i(\xi_1,\xi_2) = f^A(\xi_1)\frac{1-\xi_2}{2} + f^C(\xi_1)\frac{1+\xi_2}{2} + f^B(\xi_2)\frac{1-\xi_1}{2} + f^D(\xi_2)\frac{1+\xi_1}{2} \\ - \frac{1-\xi_1}{2}\frac{1-\xi_2}{2}f^A(-1) - \frac{1+\xi_1}{2}\frac{1-\xi_2}{2}f^A(1) \\ - \frac{1-\xi_1}{2}\frac{1+\xi_2}{2}f^C(-1) - \frac{1+\xi_1}{2}\frac{1+\xi_2}{2}f^C(1) \end{aligned}$$

2.4.2 Classes

All local expansions are derived from the top level Expansion base class. Three classes, Expansion1D, Expansion2D and Expansion3D, are derived from this and provided base classes for expansions in one-, two- and three- dimensions, respectively. The various local expansions are derived from these. The class hierarchy is shown in Figure 2.5.

One dimension:

- SegExp - Line expansion, local version of StdRegions::StdSegExp.

Two dimensions:

- TriExp - Triangular expansion.
- QuadExp - Quadrilateral expansion.

Three dimensions:

- TetExp - Tetrehedral expansion. (All triangular faces)
- HexExp - Hexahedral expansion. (All rectangular faces)
- PrismExp - Prism expansion. (Two triangular, three rectangular faces)
- PyrExp - Pyramid expansion. (One rectangular, four triangular faces)

Other classes:

- PointExp
- LinSys
- MatrixKey

2.5 Collections

The Collections library contains optimised approaches to performing finite element operations on multiple elements. Typically, the geometric information is handled separately to the core reference operator, allowing the reference operator to be applied to elements stored in contiguous blocks of memory. While this does not necessarily reduce the operation count, data transfer from memory to CPU is often substantially reduced and the contiguous storage structures enable more efficient access, thereby reducing runtime.

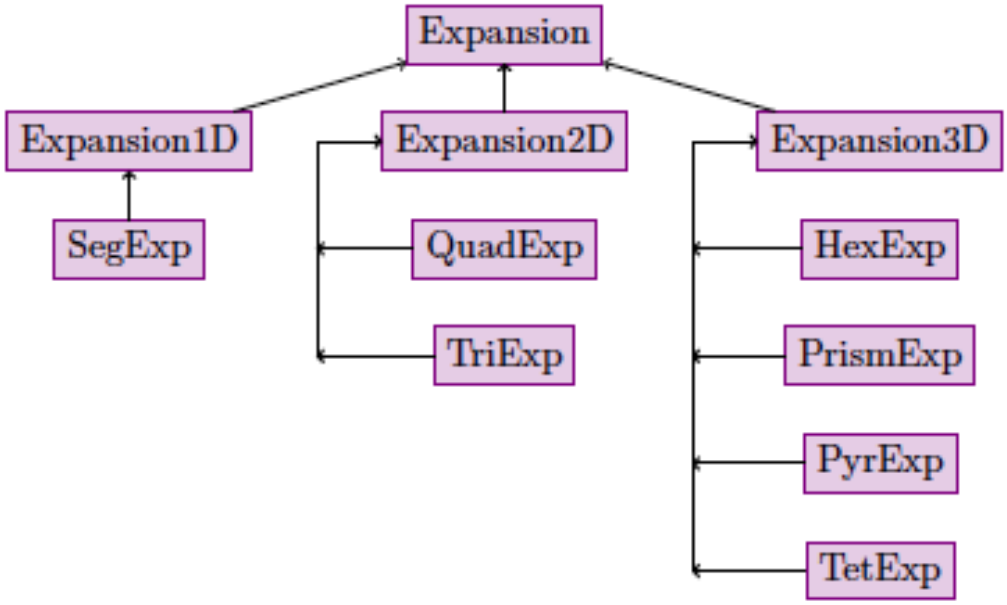


Figure 2.5 Main classes in the LocalRegions library.

2.5.1 Structure

The top-level container is the `Collection` class, which manages one or more LocalRegions objects of the same type (shape and basis). The `Operator` class generically describes an operation of these elements. Derived classes from `Operator` are created for each specific operation (e.g. BwdTrans, IProductWRTBase) on each element type. A factory pattern is used to instantiate the correct class using the key triple (shape, operator, impl). All classes relating to a particular operator are collated in a single .cpp file.

An example template for a specific `Operator` class is as follows:

```
1  class [[NAME]] : public Operator
2  {
3      public:
4          OPERATOR_CREATE([[NAME]])
5
6          virtual ~[[NAME]]()
7          {
8          }
9
10         virtual void operator()(
11                 const Array<OneD, const NekDouble> &input,
12                       Array<OneD, NekDouble> &output,
13                       Array<OneD, NekDouble> &output1,
14                       Array<OneD, NekDouble> &output2,
15                       Array<OneD, NekDouble> &wsp)
```

```
16          {
17              [[IMPLEMENTATION]]
18          }
19
20      protected:
21          [[MEMBERS]]
22
23      private:
24          [[NAME]](
25                  vector<StdRegions::StdExpansionSharedPtr> pCollExp,
26                  CoalescedGeomDataSharedPtr pGeomData)
27              : Operator(pCollExp, pGeomData)
28          {
29              [[INITIALISATION]]
30          }
31  };
```

The placeholders in double square brackets should be replaced as follows:

- `[[NAME]]`: The name of the class in the form

  $$\texttt{<operation>\_<impl>\{\_<shape>\}}$$

  where the shape need only be included if the operator is shape-specific.

- `[[MEMBERS]]`: Any member variables necessary to store precomputed quantities and ensure computational efficiency.

- `[[IMPLEMENTATION]]`: The code which actually computes the action of the operator on the elements in the collection.

- `[[INITIALIZATION]]`: Code to initialize member variables and precomputed quantities.

2.5.2 Instantiation

Operators are instantiated through the OperatorFactory. Therefore the operator classes must be registered with the factory during start-up. This is achieved using static initialisation with either the `m_type` or `m_typeArr`. The latter is shown in the following example:

```
1 OperatorKey BwdTrans_StdMat::m_typeArr[] = {
2     GetOperatorFactory().RegisterCreatorFunction(
3         OperatorKey(eSegment, eBwdTrans, eStdMat,false),
4         BwdTrans_StdMat::create, "BwdTrans_StdMat_Seg"),
5     GetOperatorFactory().RegisterCreatorFunction(
6         OperatorKey(eTriangle, eBwdTrans, eStdMat,false),
7         BwdTrans_StdMat::create, "BwdTrans_StdMat_Tri"),
8     ...
9 };
```

This instructs the factory to use the `BwdTrans_StdMat` class for all shapes when performing a backward transform using the `StdMat` approach. In contrast, if the class is shape specific, the non-array member variable would be initialised, for example:

```
1 OperatorKey BwdTrans_SumFac_Seg::m_type = GetOperatorFactory().
2     RegisterCreatorFunction(
3         OperatorKey(eSegment, eBwdTrans, eSumFac, false),
4         BwdTrans_SumFac_Seg::create, "BwdTrans_SumFac_Seg");
```

2.6 MultiRegions

In the MultiRegions library, all classes and routines are related to the process of assembling a global spectral/hp expansion out of local elemental contributions are bundled together. The most important entities of this library are the base class ExpList and its daughter classes. These classes all are the abstraction of a multi-elemental spectral/hp element expansion. Three different types of multi-elemental expansions can be distinguished:

2.6.1 A collection of local expansions

This collection is just a list of local expansions, without any coupling between the expansions on the different elements, and can be formulated as:

$$u^{\delta}(\boldsymbol{x}) = \sum_{e=1}^{N_{\text{el}}} \sum_{n=0}^{N_m^e - 1} \hat{u}_n^e \phi_n^e(\boldsymbol{x})$$

where

- N_{el} is the number of elements,
- N_m^e is the number of local expansion modes within the element e,
- $\phi_n^e(\boldsymbol{x})$ is the n^{th} local expansion mode within the element e,
- $\hat{u}_n^e$ is the n^{th} local expansion coefficient
- within the element e.

These types of expansion are represented by the classes ExpList0D, ExpList1D, ExpList2D and ExpList3D, depending on the dimension of the problem (ExpList0D is used just to deal with boundary conditions for 1D expansions).

2.6.2 A multi-elemental discontinuous global expansion

The expansions are represented by the classes DisContField1D, DisContField2D and DisContField3D. Objects of these classes should be used when solving partial differential equations using a discontinuous Galerkin approach. These classes enforce a coupling between elements and augment the domain with boundary conditions.

All local elemental expansions are now connected to form a global spectral/hp representation. This type of global expansion can be defined as:

$$u^{\delta}(\boldsymbol{x}) = \sum_{n=0}^{N_{\text{dof}}-1} \hat{u}_n \Phi_n(\boldsymbol{x}) = \sum_{e=1}^{N_{\text{el}}} \sum_{n=0}^{N_m^e-1} \hat{u}_n^e \phi_n^e(\boldsymbol{x})$$

where

- N_{dof} refers to the number of global modes,
- $\Phi_n(\boldsymbol{x})$ is the n^{th} global expansion mode,
- $\hat{u}_n$ is the n^{th} global expansion coefficient.

Typically, a mapping array to relate the global degrees of freedom $\hat{u}_n$ and local degrees of freedom $\hat{u}_n^e$ is required to assemble the global expansion out of the local contributions.

In order to solve (second-order) partial differential equations, information about the boundary conditions should be incorporated in the expansion. In case of a standard Galerkin implementation, the Dirichlet boundary conditions can be enforced by lifting a known solution satisfying these conditions, leaving a homogeneous Dirichlet problem to be solved. If we denote the unknown solution by $u^{\mathcal{H}}(\boldsymbol{x})$ and the known Dirichlet boundary conditions by $u^{\mathcal{D}}(\boldsymbol{x})$ then we can decompose the solution $u^{\delta}(\boldsymbol{x})$ into the form

$$u^{\delta}(\boldsymbol{x}_i) = u^{\mathcal{D}}(\boldsymbol{x}_i) + u^{\mathcal{H}}(\boldsymbol{x}_i) = \sum_{n=0}^{N^{\mathcal{D}}-1} \hat{u}_n^{\mathcal{D}} \Phi_n(\boldsymbol{x}_i) + \sum_{n=N^{\mathcal{D}}}^{N_{\text{dof}}-1} \hat{u}_n^{\mathcal{H}} \Phi_n(\boldsymbol{x}_i).$$

Implementation-wise, the known solution can be lifted by ordering the known degrees of freedom $\hat{u}_n^{\mathcal{H}}$ first in the global solution array $\hat{\boldsymbol{u}}$.

2.6.3 A multi-elemental continuous global expansion

The discontinuous case is supplimented with a global continuity condition. In this case a C^0 continuity condition is imposed across the element interfaces and the expansion is therefore globally continuous.

This type of global continuous expansion which incorporates the boundary conditions are represented by the classes ContField1D, ContField2D and ContField3D. Objects of these classes should be used when solving partial differential equations using a standard Galerkin approach.

2.6.4 Additional classes

Furthermore, we have two more sets of classes:

- The class LocalToGlobalBaseMap and its daughter classes: LocalToGlobalC0ContMap and LocalToGlobalDGMap.

  These classes are an abstraction of the mapping from local to global degrees of freedom and contain one or both of the following mapping arrays:

  – map $[e][n]$

    This array contains the index of the global degree of freedom corresponding to the n^{th} local expansion mode within the e^{th} element.

  – bmap $[e][n]$

    This array contains the index of the global degree of freedom corresponding to the n^{th} local boundary mode within the e^{th} element.

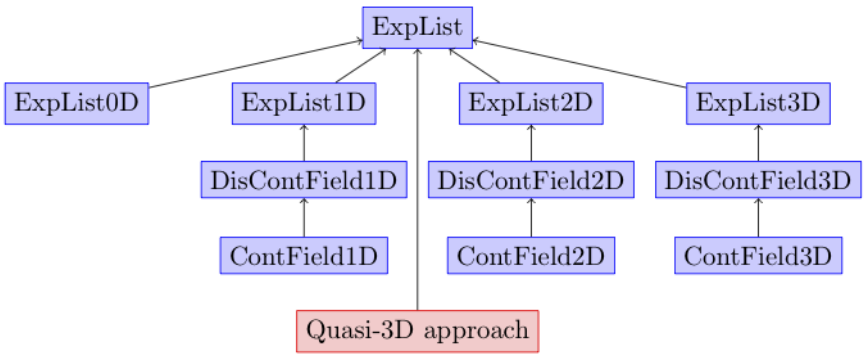
  Next to the mapping array, these classes also contain routines to assemble the global system from the local contributions, and other routines to transform between local and global level.

- The classes GlobalLinSys and GlobalLinSysKey.

  The class GlobalLinSys is an abstraction of the global system matrix resulting from the global assembly procedure. Depending of the choice to statically condense the global matrix or not, the relevant blocks are stored as a member of this class. Given a proper right hand side vector, this class also contains a routine to solve the resulting matrix system.

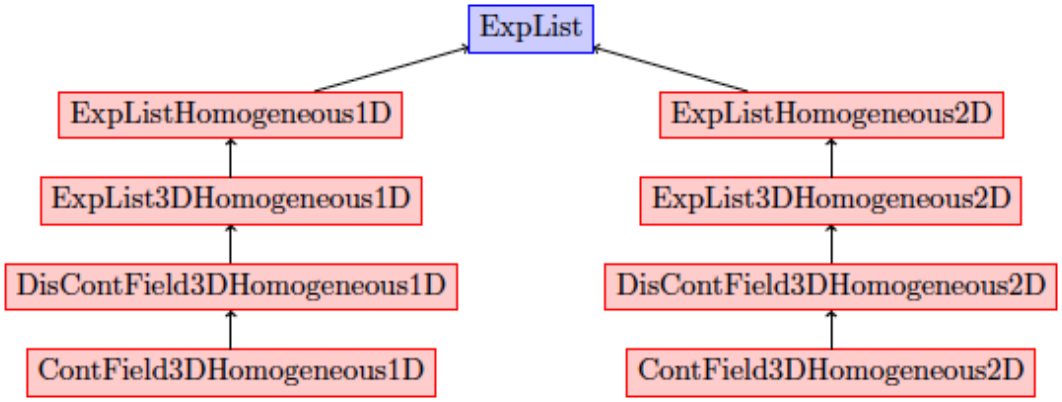
  The class GlobalLinSysKey represents a key which uniquely defines a global matrix. This key can be used to construct or retrieve the global matrix associated to a certain key.

More information about the implementation of connectivity between elements in Nektar++ can be found [wiki:Connectivity here].



2.6.5 Quasi-3D approach

The Quasi-3D approach is an extension of the 1D and the 2D spectral/hp element method. This technique permits to study 3D problems combining the spectral/hp element method with a spectral method. In the Quasi-3D approach with 1 homogenous direction, the third dimension (z-axis) is expandend with an harmonic expansion (a Fourier series). In each quadrature point of the Fourier discretisation we can find a 2D plane discretised with a 2D spectral/hp elements expasions. In the case with 2 homogeneous directions a plane is discretised with a 2D Fourier expansion (y-z palne). In each one of the quadrature point of this harmonic expansion there is a 1D spectral/hp element discretisation. The homogenous classes derive directly form ExpList, and they are ExpListHomogeneous1D and ExpListHomogeneous2D. This classes are used to represent the collections of 2D (or 1D) spectral/hp element problems which are located in the Fourier expansions quatradure points to create a 3D problem. As describer above, we can find the find the continuos or discontinuos case, depending on the spectral/hp element approach. ExpList2DHomogeneous1D and ExpList1DHomogeneous2D are used to manage boundary conditions. A description of the Quasi-3D approach usage can be found in Chapter 3 in the User Guide.



2.7 SolverUtils

2.7.1 Drivers

Drivers govern the high-level execution of a solver.

2.7.1.1 Implementing a new Driver

To take advantage of the Nektar++ architecture and implement an algorithm which will wrap around an existing solver, new drivers can be created. This can be done in a few steps by using DriverStandard.cpp (and .h) as a template:

- Create the new files called DriverMyAlgorithm.cpp (and .h)
- Implement constructor and destructor
- Provide implementation for `v_InitObject` and `v_Execute` as necessary.
- Register the new driver with the driver factory.

```
1 string DriverMyAlgorithm::className =
2     GetDriverFactory().RegisterCreatorFunction("MyAlgorithm",
3                                                DriverMyAlgorithm::create);
4 string DriverMyAlgorithm::driverLookupId =
5     LibUtilities::SessionReader::RegisterEnumValue("Driver","MyAlgorithm",0);
```

- Add the new driver to the library. In `CMakeLists.txt`, DriverMyAlgorithm.cpp must be added in the `SOLVER_UTILS_SOURCES` section and DriverMyAlgorithm.h in the `SOLVER_UTILS_HEADERS` section.

Chapter 3

Data Structures and Algorithms

3.1 Connectivity

The typical elemental decomposition of the spectral/hp element method requires a global assembly process when considering multi-elemental problems. This global assembly will ensure some level of connectivity between adjacent elements sucht that there is some form of continuity across element boundaries in the global solution. In this section, we will merely focus on the classical Galerkin method, where global continuity is typically imposed by making the approximation C^0 continuous.

3.1.1 Connectivity in two dimensions

As explained in [**?**], the global assembly process involves the transformation from local degrees of freedom to global degrees of freedom (DOF). This transformation is typically done by a mapping array which relates the numbering of the local (= elemental) DOF's to the numbering of the global DOF's. To understand how this transformation is set up in Nektar++ one should understand the following:

- **Starting point**

  The starting point is the initial numbering of the elemental expansion modes. This corresponds to the order in which the different local expansion modes are listed in the coefficient array `m_coeffs` of the elemental (local or standard) expansion. The specific order in which the different elemental expansion modes appear is motivated by the compatability with the sum-factorisation technique. This also implies that this ordering is fixed and should not be changed by the user. Hence, this unchangeable initial local numbering will serve as starting input for the connectivity.

- **end point**

  Obviously, we are working towards the numbering of the global DOF's. This global ordering should:

  - reflect the chosen continuity approach (standard C^0 Galerkin in our case)

– (optionally) have some optimal ordering (where optimality can be defined in different ways, e.g. minimal bandwith)

All intermittent steps from starting point to end point can basically be chosen freely but they should allow for an efficient construction of the global numbering system starting from the elemental ordering of the local degrees of freedom. Currently, Nektar++ provides a number of tools and routines in the different sublibraries which can be employed to set up the mapping from local to global DOF's. These tools will be listed below, but first the connectivity strategies for both modal and nodal expansions will be explained. Note that all explanations below are focussed on quadrilateral elements. However, the general idea equally holds for triangles.

3.1.2 Connectivity strategies

For a better understanding of the described strategies, one should first understand how Nektar++ deals with the basic geometric concepts such as edges and (2D) elements.

In Nektar++, a (2D) element is typically defined by a set of edges. In the input .xml files, this should be done as (for a quadrilateral element):

```
1 <Q ID="i"> e0 e1 e2 e3</Q>
```

where `e0` to `e3` correspond to the mesh ID's of the different edges. It is important to know that in Nektar++, the convention is that these edges should be ordered counterclokwise (Note that this order also corresponds to the order in which the edges are passed to the constructors of the 2D geometries). In addition, note that we will refer to edge `e0` as the edge with local (elemental) edge ID equal to 0, to edge `e1` as local edge with ID 1, to edge `e2` as local edge with ID equal 2 and to edge `e3` as local edge with ID 3. Furthermore, one should note that the local coordinate system is orientated such that the first coordinate axis is aligned with edge 0 and 2 (local edge ID), and the second coordinate axis is aligned with edge 1 and 3. The direction of these coordinate axis is such that it points in counterclockwise direction for edges 0 and 1, and in clockwise direction for edge 2 and 3.

Another important feature in the connectivity strategy is the concept of edge orientation. For a better understanding, consider the input format of an edge as used in the input .xml files which contain the information about the mesh. An edge is defined as:

```
1 <E ID="i"> v0 v1 </E>
```

where `v0` and `v1` are the ID's of the two vertices that define the edge (Note that these vertices are passed in the same order to the constructor of the edge). Now, the orientation of an edge of a two-dimensional element (i.e. quadrilateral or triangle) is defined as:

- Forward if the vertex with ID `v0` comes before the vertex with ID `v1` when considering the vertices the vertices of the element in a counterclockwise direction

- Backward otherwise.

This has the following implications:

- The common edge of two adjacent elements has always a forward orientation for one of the elements it belongs to and a backward orientation for the other.
- The orientation of an edge is only relevant when considering two-dimensional elements. It is a property which is not only inherent to the edge itself, but depends on the element it belongs to. (This also means that a segment does not have an orientation)

3.1.2.1 Modal expansions

We will follow the basic principles of the connectivity strategy as explained in Section 4.2.1.1 of [**?**] (such as the hierarchic ordering of the edge modes). However, we do not follow the strategy described to negate the odd modes of an intersecting edge of two adjacent elements if the local coordinate systems have an opposite direction. The explained strategy involves checking the direction of the local coordinate systems of the neighbouring elements. However, for a simpler automatic procedure to identify which edges need to have odd mode negated, we would like to have an approach which can be applied to the elements individually, without information being coupled between neighbouring elements. This can be accomplished in the following way. Note that this approach is based on the earlier observation that the intersecting edge of two elements always has opposite orientation. Proper connectivity can now be guaranteed if:

- forward oriented edges always have a counterclockwise local coordinate axis
- backward oriented edges always have a clockwise local coordinate axis.

Both the local coordinate axis along an intersecting edge will then point in the same direction. Obviously, these conditions will not be fulfilled by default. But in order to do so, the direction of the local coordinate axis should be reversed in following situations:

```
1  if ((LocalEdgeId == 0)||(LocalEdgeId == 1)) {
2      if( EdgeOrientation == Backward ) {
3          change orientation of local coordinate axis
4      }
5  }
6
7  if ((LocalEdgeId == 2)||(LocalEdgeId == 3)) {
8      if( EdgeOrientation == Forward ) {
9          change orientation of local coordinate axis
10     }
11 }
```

This algorithm above is based on the earlier observation that the local coordinate axis automatically point in counterclockwise direction for edges 0 and 1 and in clockwise direction for the other edges. As explained in [**?**] the change in local coordinate axis can actually be done by reversing the sigqn of the odd modes. This is implemented by means of an additional sign vector.

3.1.2.2 Nodal expansions

For the nodal expansions, we will use the connectivity strategy as explained in Section 4.2.1.1 of [**?**]. However, we will clarify this strategy from a Nektar++ point of view. As pointed out in [**?**], the nodal edge modes can be identified with the physical location of the nodal points. In order to ensure proper connectivity between elements the egde modes with the same nodal location should be matched. This will be accomplished if both the sets of local edge modes along the intersection edge of two elements are numbered in the same direction. And as the intersecting edge of two elements always has opposite direction, this can be guaranteed if:

- the local numbering of the edge modes is counterclockwise for forward oriented edges
- the local numbering of the edge modes is clockwise for backward oriented edges.

This will ensure that the numbering of the global DOF's on an edge is in the same direction as the tow subsets of local DOF's on the intersecting edge.

3.1.3 Implementation

The main entity for the transformation from local DOF's to global DOF's (in 2D) is the LocalToGlobalMap2D class. This class basically is the abstraction of the mapping array map[e][i] as introduced in section 4.2.1 of [**?**]. This mapping array is contained in the class' main data member, LocalToGlobalMap2D::m_locToContMap. Let us recall what this mapping array

```
map[e][i] = globalID
```

actually represents:

- e corresponds to the e th element
- i corresponds to the i th expansion mode within element e. This index i in this map array corresponds to the index of the coefficient array m_coeffs.
- globalID represents the ID of the corresponding global degree of freedom.

However, rather than this two-dimensional structure of the mapping array, `LocalToGlobalMap2D::m_locToContMap` stores the mapping array as a one-dimensional array which is the concatenation of the different elemental mapping arrays map[e]. This mapping array can then be used to assemble the global system out of the local entries, or to do any other transformation between local and global degrees of freedom (Note that other mapping arrays such as the boundary mapping bmap[e][i] or the sign vector sign[e][i] which might be required are also contained in the LocalToGlobalMap2D class).

For a better appreciation of the implementation of the connectivity in Nektar++, it might be useful to consider how this mapping array is actually being constructed (or filled). To understand this, first consider the following:

- The initial local elemental numbering (referred to as starting point in the beginning of this document) is not suitable to set up the mapping. In no way does it correspond to the local numbering required for a proper connectivity as elaborated in Section 4.2.1 of [**?**]. Hence, this initial ordering asuch cannot be used to implement the connectivity strategies explained above. As a result, additional routines (see here), which account for some kind of reordering of the local numbering will be required in order to construct the mapping array properly.

- Although the different edge modes can be thought of as to include both the vertex mode, we will make a clear distinction between them in the implementation. In other words, the vertex modes will be treated separately from the other modes on an edge as they are conceptually different from an connectivity point of view. We will refer to these remaining modes as interior edge modes.

The fill-in of the mapping array can than be summarised by the following part of (simplified) code:

```
1  for(e = 0; e < Number_Of_2D_Elements; e++) {
2      for(i = 0; i < Number_Of_Vertices_Of_Element_e; i++) {
3          offsetValue = ...
4          map[e][GetVertexMap(i)] = offsetValue;
5      }
6
7      for(i = 0; i < Number_Of_Edges_Of_Element_e; i++) {
8          localNumbering = GetEdgeInteriorMap(i); offsetValue = ...
9          for(j = 0; j < Number_Of_InteriorEdgeModes_Of_Edge_i; j++) {
10             map[e][localNumbering(j)] = offsetValue + j;
11         }
12     }
13 }
```

In this document, we will not cover how the calculate the offsetValue which:

- for the vertices, corresponds to the global ID of the specific vertex

- for the edges, corresponds to the starting value of the global numbering on the concerned edge

However, we would like to focus on the routines GetVertexMap() and GetEdgeInteriorMap(), 2 functions which somehow reorder the initial local numbering in order to be compatible with the connectivity strategy.

3.1.3.1 GetVertexMap()

Given the local vertex id (i.e. 0,1,2 or 3 for a quadrilateral element), it returns the position of the corresponding vertex mode in the elemental coefficient array StdRegions::StdExpansion::m_coeffs. By using this function as in the code above, it is ensured that the global ID of the vertex is entered in the correct position in the mapping array.

3.1.3.2 GetEdgeInteriorMap()

Like the previous routine, this function is also defined for all two dimensional expanions in the StdRegions::StdExpansion class tree. This is actually the most important function to ensure proper connectivity between neigbouring elements. It is a function which reorders the numbering of local DOF's according to the connectivity strategy. As input this function takes:

- the local edge ID of the edge to be considered
- the orientation of this edge.

As output, it returns the local ordering of the requested (interior) edge modes. This is contained in an array of size N-2, where N is the number of expansion modes in the relevant direction. The entries in this array represent the position of the corresponding interior edge mode in the elemental coefficient array StdRegions::StdExpansion::m_coeffs.

Rather than the actual values of the local numbering, it is the ordering of local edge modes which is of importance for the connectivity. That is why it is important how the different interior edge modes are sorted in the returned array. This should be such that for both the elements which are connected by the intersecting edge, the local (interior) edge modes are iterated in the same order. This will guarantee a correct global numbering scheme when employing the algorithm shown above. This proper connectivity can be ensured if the function GetEdgeInteriorMap:

- for modal expansions: returns the edge interior modes in hierarchical order (i.e. the lowest polynomial order mode first),
- for nodal expansions: returns the edge interior modes in:
  - counterclockwise order for forward oriented edges
  - clockwise order for backward oriented edges.

3.2 Time integration

This page discusses the implementation of time-integration in Nektar++.

3.2.1 General Linear Methods

General linear methods (GLMs) can be considered as the generalization of a broad range of different numerical methods for ordinary differential equations. They were introduced by Butcher and they provide a unified formulation for traditional methods such as the Runge-Kutta methods and the linear multi-step methods. From an implementation point of view, this means that all these numerical methods can be abstracted in a similar way. As this allows a high level of generality, it is chosen in Nektar++ to cast all time integration schemes in the framework of general linear methods.

For background information about general linear methods, please consult [**?**]

3.2.2 Introduction

The standard initial value problem can written in the form

$$\frac{d\boldsymbol{y}}{dt} = \boldsymbol{f}(t, \boldsymbol{y}), \quad \boldsymbol{y}(t_0) = \boldsymbol{y}_0$$

where $\boldsymbol{y}$ is a vector containing the variable (or an array of array contianing the variables).

In the formulation of general linear methods, it is more convenient to consider the ODE in autonomous form, i.e.

$$\frac{d\hat{\boldsymbol{y}}}{dt} = \hat{\boldsymbol{f}}(\hat{\boldsymbol{y}}), \quad \hat{\boldsymbol{y}}(t_0) = \hat{\boldsymbol{y}}_0.$$

3.2.3 Formulation

Suppose the governing differential equation is given in autonomous form, the n^{th} step of the GLM comprising

- r steps (as in a multi-step method)
- s stages (as in a Runge-Kutta method)

is formulated as:

$$\boldsymbol{Y}_i = \Delta t \sum_{j=0}^{s-1} a_{ij} \boldsymbol{F}_j + \sum_{j=0}^{r-1} u_{ij} \hat{\boldsymbol{y}}_j^{[n-1]}, \qquad i = 0, 1, \ldots, s-1$$

$$\hat{\boldsymbol{y}}_i^{[n]} = \Delta t \sum_{j=0}^{s-1} b_{ij} \boldsymbol{F}_j + \sum_{j=0}^{r-1} v_{ij} \hat{\boldsymbol{y}}_j^{[n-1]}, \qquad i = 0, 1, \ldots, r-1$$

where $\boldsymbol{Y}_i$ are referred to as the stage values and $\boldsymbol{F}_j$ as the stage derivatives. Both quantities are related by the differential equation:

$$\boldsymbol{F}_i = \hat{\boldsymbol{f}}(\boldsymbol{Y}_i).$$

The matrices $A = [a_{ij}]$, $U = [u_{ij}]$, $B = [b_{ij}]$, $V = [v_{ij}]$ are characteristic of a specific method. Each scheme can then be uniquely defined by a partioned $(s + r) \times (s + r)$ matrix

$$\begin{bmatrix} A & U \\ B & V \end{bmatrix}$$

3.2.4 Matrix notation

Adopting the notation:

$$\hat{\boldsymbol{y}}^{[n-1]} = \begin{bmatrix} \hat{\boldsymbol{y}}_0^{[n-1]} \\ \hat{\boldsymbol{y}}_1^{[n-1]} \\ \vdots \\ \hat{\boldsymbol{y}}_{r-1}^{[n-1]} \end{bmatrix}, \quad \hat{\boldsymbol{y}}^{[n]} = \begin{bmatrix} \hat{\boldsymbol{y}}_0^{[n]} \\ \hat{\boldsymbol{y}}_1^{[n]} \\ \vdots \\ \hat{\boldsymbol{y}}_{r-1}^{[n]} \end{bmatrix}, \quad \boldsymbol{Y} = \begin{bmatrix} \boldsymbol{Y}_0 \\ \boldsymbol{Y}_1 \\ \vdots \\ \boldsymbol{Y}_{s-1} \end{bmatrix}, \quad \boldsymbol{F} = \begin{bmatrix} \boldsymbol{F}_0 \\ \boldsymbol{F}_1 \\ \vdots \\ \boldsymbol{F}_{s-1} \end{bmatrix}$$

the general linear method can be written more compactly in the following form:

$$\begin{bmatrix} \boldsymbol{Y} \\ \hat{\boldsymbol{y}}^{[n]} \end{bmatrix} = \begin{bmatrix} A \otimes I_N & U \otimes I_N \\ B \otimes I_N & V \otimes I_N \end{bmatrix} \begin{bmatrix} \Delta t \boldsymbol{F} \\ \hat{\boldsymbol{y}}^{[n-1]} \end{bmatrix}$$

where I_N is the identity matrix of dimension $N \times N$.

3.2.5 General Linear Methods in Nektar++

Although the GLM is essentially presented for ODE's in its autonomous form, in Nektar++ it will be used to solve ODE's formulated in non-autonomous form. Given the ODE,

$$\frac{d\boldsymbol{y}}{dt} = \boldsymbol{f}(t, \boldsymbol{y}), \quad \boldsymbol{y}(t_0) = \boldsymbol{y}_0$$

a single step of GLM can then be evaluated in the following way:

- **input**

  $\boldsymbol{y}^{[n-1]}$, *i.e.* the r subvectors comprising $\boldsymbol{y}_i^{[n-1]}$ - $t^{[n-1]}$, *i.e.* the equivalent of $\boldsymbol{y}^{[n-1]}$ for the time variable t.

- **step 1**: The stage values $\boldsymbol{Y}_i$, $\boldsymbol{T}_i$ and the stage derivatives $\boldsymbol{F}_i$ are calculated through the relations:

  1. $\boldsymbol{Y}_i = \Delta t \sum_{j=0}^{s-1} a_{ij} \boldsymbol{F}_j + \sum_{j=0}^{r-1} u_{ij} \boldsymbol{y}_j^{[n-1]}, \qquad i = 0, 1, \ldots, s-1$

2. $T_i \;=\; \Delta t \sum_{j=0}^{s-1} a_{ij} + \sum_{j=0}^{r-1} u_{ij} t_j^{[n-1]}, \qquad i = 0, 1, \ldots, s-1$
3. $\boldsymbol{F}_i \;=\; f(T_i, \boldsymbol{Y}_i), \qquad i = 0, 1, \ldots, s-1$

- **step 2**: The approximation at the new time level $\boldsymbol{y}^{[n]}$ is calculated as a linear combination of the stage derivatives $\boldsymbol{F}_i$ and the input vector $\boldsymbol{y}^{[n-1]}$. In addition, the time vector $t^{[n]}$ is also updated

1. $\boldsymbol{y}_i^{[n]} = \Delta t \sum_{j=0}^{s-1} b_{ij} \boldsymbol{F}_j + \sum_{j=0}^{r-1} v_{ij} \boldsymbol{y}_j^{[n-1]}, \qquad i = 0, 1, \ldots, r-1$
2. $t_i^{[n]} \;=\; \Delta t \sum_{j=0}^{s-1} b_{ij} + \sum_{j=0}^{r-1} v_{ij} t_j^{[n-1]}, \qquad i = 0, 1, \ldots, r-1$

- **output**

1. $\boldsymbol{y}^{[n]}$, i.e. the r subvectors comprising $\boldsymbol{y}_i^{[n]}$. $\boldsymbol{y}_0^{[n]}$ corresponds to the actual approximation at the new time level.
2. $t^{[n]}$ where $t_0^{[n]}$ is equal to the new time level $t + \Delta t$.

For a detailed describtion of the formulation and a deeper insight of the numerical method see [**?**].

3.2.6 Types of time Integration Schemes

Nektar++ contains various classes and methods which implement the concept of GLMs. This toolbox is capable of numerically solving the generalised ODE using a broad range of different time-stepping methods. We distinguish the following types of general linear methods:

- **Formally Explicit Methods**: These types of methods are considered explicit from an ODE point of view. They are characterised by a lower triangular coefficient matrix A, "i.e." $a_{ij} = 0$ for $j \geq i$. To avoid confusion, we make a further distinction:
  - **direct explicit method**: Only forward operators are required.
  - **indirect explicit method**: The inverse operator is required.
- **Diagonally Implicit Methods**: Compared to explicit methods, the coefficient matrix A has now non-zero entries on the diagonal. This means that each stage value depend on the stage derivative at the same stage, requiring an implicit step. However, the calculation of the different stage values is still uncoupled. Best known are the DIRK schemes.
- **IMEX schemes**: These schemes support the concept of being able to split right hand forcing term into an explicit and implicit component. This is useful in advection diffusion type problems where the advection is handled explicity and the diffusion is handled implicit.

- **Fully Implicit Methods Methods**: The coefficient matrix has a non-zero upper triangular part. The calculation of all stages values is fully coupled.

The aim in Nektar++ is to fully support the first three types of GLMs. Fully implicit methods are currently not implemented.

3.2.7 Usage

The goal of abstracting the concept of general linear methods is to provide users with a single interface for time-stepping, independent of the chosen method. The classes tree allows the user to numerically integrate ODE's using high-order complex schemes, as if it was done using the Forward-Euler method. Switching between time-stepping schemes is as easy as changing a parameter in an input file.

In the already implemented solvers the time-integration schemes have been set up according to the nature of the equations. For example the incompressible Navier-Stokes equations solver allows the use of three different Implicit-Explicit time-schemes if solving the equations with a splitting-scheme. This is because this kind of scheme has an explicit and an implicit operator that combined solve the ODE's system.

Once aware of the problem's nature and implementation, the user can easily switch between some (depending on the problem) of the following time-integration schemes:

AdamsBashforthOrder1	Adams-Bashforth Forward multi-step scheme of order 1
AdamsBashforthOrder2	Adams-Bashforth Forward multi-step scheme of order 2
AdamsBashforthOrder3	Adams-Bashforth Forward multi-step scheme of order 3
AdamsMoultonOrder1	Adams-Moulton Forward multi-step scheme of order 1
AdamsMoultonOrder2	Adams-Moulton Forward multi-step scheme of order 2
BDFImplicitOrder2	BDF multi-step scheme of order 2 (implicit)
ClassicalRungeKutta4	Runge-Kutta multi-stage scheme 4th order explicit
RungeKutta2_ModifiedEuler	Runge-Kutta multi-stage scheme 2nd order explicit
RungeKutta2_ImprovedEuler	Runge-Kutta multi-stage scheme 2nd order explicit
ForwardEuler	Forward-Euler scheme
BackwardEuler	Backward Euler scheme
IMEXOrder1	IMEX 1st order scheme using Euler Backwards Euler Forwards
IMEXOrder2	IMEX 2nd order scheme using Backward Different Formula & Extrapolation
IMEXOrder3	IMEX 3rd order scheme using Backward Different Formula & Extrapolation
Midpoint	Midpoint method
DIRKOrder2	Diagonally Implicit Runge-Kutta scheme of order 2
DIRKOrder3	Diagonally Implicit Runge-Kutta scheme of order 3
CNAB	Crank-Nicolson-Adams-Bashforth Order 2 (CNAB)
IMEXGear	IMEX Gear Order 2
MCNAB	Modified Crank-Nicolson-Adams-Bashforth Order 2 (MCNAB)
IMEXdirk_1_1_1	Forward-Backward Euler IMEX DIRK(1,1,1)
IMEXdirk_1_2_1	Forward-Backward Euler IMEX DIRK(1,2,1)
IMEXdirk_1_2_2	Implicit-Explicit Midpoint IMEX DIRK(1,2,2)
IMEXdirk_2_2_2	L-stable, two stage, second order IMEX DIRK(2,2,2)
IMEXdirk_2_3_2	L-stable, three stage, third order IMEX DIRK(3,4,3)
IMEXdirk_2_3_3	L-stable, two stage, third order IMEX DIRK(2,3,3)
IMEXdirk_3_4_3	L-stable, three stage, third order IMEX DIRK(3,4,3)
IMEXdirk_4_4_3	L-stable, four stage, third order IMEX DIRK(4,4,3)

Nektar++ input file for your problem will ask you just the string corresponding the time-stepping scheme you want to use (between quotation marks in the previous list), and few parameters to define your integration in time (time-step and number of steps or final time). For example:

```
<SOLVERINFO>
  <I PROPERTY="EQTYPE" VALUE="UnsteadyNavierStokes" />
  <I PROPERTY="SolverType" VALUE="VelocityCorrectionScheme" />
  <I PROPERTY="AdvectionForm" VALUE="Convective" />
  <I PROPERTY="Projection" VALUE="Galerkin" />
  <I PROPERTY="TimeIntegrationMethod" VALUE="IMEXOrder1" />
</SOLVERINFO>

<PARAMETERS>
  <P> TimeStep = 0.001 </P>
  <P> NumSteps = 1000 </P>
  <P> Kinvis = 1 </P>
</PARAMETERS>
```

3.2.8 Implementation of a time-dependent problem

In order to implement a new solver which takes advantage of the time-integration class in Nektar++, two main ingredients are required:

- A main files in which the time-integration of you ODE's system is initialized and performed.
- A class representing the spatial discretization of your problem, which reduces your system of PDE's to a system of ODE's.

Your pseudo-main file, where you actually loop over the time steps, will look like

```
1 NekDouble timestep = 0.1;
2 NekDouble time = 0.0;
3 int NumSteps = 1000;
4
5 YourClass solver(Input);
6
7 LibUtilities::TimeIntegrationMethod TIME_SCHEME;
8 LibUtilities::TimeIntegrationSchemeOperators ODE;
9
10 ODE.DefineOdeRhs(&YourClass::YourExplicitOperatorFunction,solver);
11 ODE.DefineProjection(&YourClass::YourProjectionFunction,solver);
12 ODE.DefineImplicitSolve(&YourClass::YourImplicitOperatorFunction,solver);
13
14 Array<OneD, LibUtilities::TimeIntegrationSchemeSharedPtr> IntScheme;
15
16 LibUtilities::TimeIntegrationSolutionSharedPtr ode_solution;
17
18 Array<OneD, Array<OneD, NekDouble> > U;
19
20 TIME_SCHEME = LibUtilities::eForwardEuler;
21 int numMultiSteps=1;
22 IntScheme = Array<OneD,LibUtilities::TimeIntegrationSchemeSharedPtr>(numMultiSteps);
23 LibUtilities::TimeIntegrationSchemeKey IntKey(TIME_SCHEME);
24 IntScheme[0] = LibUtilities::TimeIntegrationSchemeManager()[IntKey];
25 ode_solution = IntScheme[0]->InitializeScheme(timestep,U,time,ODE);
26
27 for(int n = 0; n < NumSteps; ++n) {
28     U = IntScheme[0]->TimeIntegrate(timestep,ode_solution,ODE);
29 }
```

We can distinguish three different sections in the code above

3.2.8.1 Definitions

```
1 NekDouble timestep = 0.1;
2 NekDouble time = 0.0;
3 int NumSteps = 1000;
4
5 YourClass equation(Input);
```

```
6
7 LibUtilities::TimeIntegrationMethod TIME_SCHEME;
8 LibUtilities::TimeIntegrationSchemeOperators ODE;
9
10 ODE.DefineOdeRhs(&YourClass::YourExplicitOperatorFunction,equation);
11 ODE.DefineProjection(&YourClass::YourProjectionFunction, equation);
12 ODE.DefineImplicitSolve(&YourClass::YourImplicitOperatorFunction, equation);
```

In this section you define the basic parameters (like time-step, initial time, etc.) and the time-integration objects. The operators are not all required, it depends on the nature of your problem and on the type of time integration schemes you want to use. In this case, the problem has been set up to work just with Forward-Euler, then for sure you will not need the implicit operator. An object named `equation` has been initialized, is an object of type `YourClass`, where your spatial discretization and the functions which actually represent your operators are implemented. An example of this class will be shown later in this page.

3.2.8.2 Initialisations

```
1 Array<OneD,LibUtilities::TimeIntegrationSchemeSharedPtr> IntScheme;
2
3 LibUtilities::TimeIntegrationSolutionSharedPtr ode_solution;
4
5 Array<OneD, Array<OneD, NekDouble> > U;
6
7 TIME_SCHEME = LibUtilities::eForwardEuler;
8 int numMultiSteps=1;
9 IntScheme = Array<OneD,LibUtilities::TimeIntegrationSchemeSharedPtr>(numMultiSteps);
10 LibUtilities::TimeIntegrationSchemeKey IntKey(TIME_SCHEME);
11 IntScheme[0] = LibUtilities::TimeIntegrationSchemeManager()[IntKey];
12 ode_solution = IntScheme[0]->InitializeScheme(timestep,U,time,ODE);
```

The second part consists in the scheme initialization. In this example we set up just Forward-Euler, but we can set up more then one time-integration scheme and quickly switch between them from the input file. Forward-Euler does not require any other scheme for the start-up procedure. High order multi-step schemes may need lower-order schemes for the start up.

3.2.8.3 Integration

```
1 for(int n = 0; n < NumSteps; ++n) {
2     U = IntScheme[0]->TimeIntegrate(timestep,ode_solution,ODE);
3 }
```

The last step is the typical time-loop, where you iterate in time to get your new solution at each time-level. The solution at time t^{n+1} is stored into vector `U` (you need to properly initialize this vector). `U` is an Array of Arrays, where the first dimension corresponds to the number of variables (eg. u,v,w) and the second dimension corresponds to the variables size (e.g. the number of modes or the number of physical points).

The variable `ODE` is an object which contains the methods. A class representing a PDE equation (or a system of equations) must have a series of functions representing the implicit/explicit part of the method, which represents the reduction of the PDE's to a system of ODE's. The spatial discretization and the definition of this method should be implemented in `YourClass`. `&YourClass::YourExplicitOperatorFunction` is a functor, i.e. a pointer to a function where the method is implemented. `equation` is a pointer to the object, i.e. the class, where the function/method is implemented. Here a pseudo-example of the .h file of your hypothetical class representing the set of equations. The implementation of the functions is meant to be in the related .cpp file.

```
1  class YourCalss
2  {
3  public:
4      YourClass(INPUT);
5
6      ~YourClass(void);
7
8      void YourExplicitOperatorFunction(
9          const Array<OneD, Array<OneD, NekDouble> > & inarray,
10                 Array<OneD, Array<OneD, NekDouble> > & outarray,
11         const NekDouble time);
12
13     void YourProjectionFunction(
14         const Array<OneD, Array<OneD, NekDouble> > & inarray,
15                 Array<OneD, Array<OneD, NekDouble> > & outarray, const
16                 NekDouble time);
17
18     void YourImplicitOperatorFunction(
19         const Array<OneD, Array<OneD, NekDouble> > & inarray,
20                 Array<OneD, Array<OneD, NekDouble> > & outarray, const
21                 NekDouble time,
22         const NekDouble lambda);
23
24     void InternalMethod1(...);
25
26     NekDouble internalvariale1;
27
28 protected:
29     ...
30
31 private:
32     ...
33 };
```

3.2.9 Strongly imposed essential boundary conditions

Dirichlet boundary conditions can be strongly imposed by lifting the known Dirichlet solution. This is equivalent to decompose the approximate solution y into an known part, $y^{\mathcal{D}}$, which satisfies the Dirichlet boundary conditions, and an unknown part, $y^{\mathcal{H}}$, which is zero on the Dirichlet boundaries, i.e.

$$y = y^{\mathcal{D}} + y^{\mathcal{H}}$$

In a Finite Element discretisation, this corresponds to splitting the solution vector of coefficients $\boldsymbol{y}$ into the known Dirichlet degrees of freedom $\boldsymbol{y}^{\mathcal{D}}$ and the unknown homogeneous degrees of freedom $\boldsymbol{y}^{\mathcal{H}}$. Ordering the known coefficients first, this corresponds to:

$$\boldsymbol{y} = \begin{bmatrix} \boldsymbol{y}^{\mathcal{D}} \\ \boldsymbol{y}^{\mathcal{H}} \end{bmatrix}$$

The generalised formulation of the general linear method (i.e. the introduction of a left hand side operator) allows for an easier treatment of these types of boundary conditions. To better appreciate this, consider the equation for the stage values for an explicit general linear method where both the left and right hand side operator are linear operators, i.e. they can be represented by a matrix.

$$\boldsymbol{M}\boldsymbol{Y}_i = \Delta t \sum_{j=0}^{i-1} a_{ij} \boldsymbol{L}\boldsymbol{Y}_j + \sum_{j=0}^{r-1} u_{ij} \boldsymbol{y}_j^{[n-1]}, \qquad i = 0, 1, \ldots, s-1$$

In case of a lifted known solution, this can be written as:

$$\begin{bmatrix} \boldsymbol{M}^{\mathcal{D}\mathcal{D}} & \boldsymbol{M}^{\mathcal{D}\mathcal{H}} \\ \boldsymbol{M}^{\mathcal{H}\mathcal{D}} & \boldsymbol{M}^{\mathcal{H}\mathcal{H}} \end{bmatrix} \begin{bmatrix} \boldsymbol{Y}_i^{\mathcal{D}} \\ \boldsymbol{Y}_i^{\mathcal{H}} \end{bmatrix} = \Delta t \sum_{j=0}^{i-1} a_{ij} \begin{bmatrix} \boldsymbol{L}^{\mathcal{D}\mathcal{D}} & \boldsymbol{L}^{\mathcal{D}\mathcal{H}} \\ \boldsymbol{L}^{\mathcal{H}\mathcal{D}} & \boldsymbol{L}^{\mathcal{H}\mathcal{H}} \end{bmatrix} \begin{bmatrix} \boldsymbol{Y}_j^{\mathcal{D}} \\ \boldsymbol{Y}_j^{\mathcal{H}} \end{bmatrix} + \sum_{j=0}^{r-1} u_{ij} \begin{bmatrix} \boldsymbol{y}_j^{\mathcal{D}[n-1]} \\ \boldsymbol{y}_j^{\mathcal{H}[n-1]} \end{bmatrix},$$
$$i = 0, 1, \ldots, s-1$$

In order to calculate the stage values correctly, the explicit operator should now be implemented to do the following:

$$\begin{bmatrix} \boldsymbol{b}^{\mathcal{D}} \\ \boldsymbol{b}^{\mathcal{H}} \end{bmatrix} = \begin{bmatrix} \boldsymbol{L}^{\mathcal{D}\mathcal{D}} & \boldsymbol{L}^{\mathcal{D}\mathcal{H}} \\ \boldsymbol{L}^{\mathcal{H}\mathcal{D}} & \boldsymbol{L}^{\mathcal{H}\mathcal{H}} \end{bmatrix} \begin{bmatrix} \boldsymbol{y}^{\mathcal{D}} \\ \boldsymbol{y}^{\mathcal{H}} \end{bmatrix}$$

Note that only the homogeneous part $\boldsymbol{b}^{\mathcal{H}}$ will be used to calculate the stage values. This means essentially that only the bottom part of the operation above, i.e. $\boldsymbol{L}^{\mathcal{H}\mathcal{D}}\boldsymbol{y}^{\mathcal{D}} + \boldsymbol{L}^{\mathcal{H}\mathcal{H}}\boldsymbol{y}^{\mathcal{H}}$ is required. However, sometimes it might be more convenient to use/implement routines for the explicit operator that also calculate $\boldsymbol{b}^{\mathcal{D}}$.

An implicit method should solve the system:

$$\left(\begin{bmatrix} \boldsymbol{M}^{\mathcal{D}\mathcal{D}} & \boldsymbol{M}^{\mathcal{D}\mathcal{H}} \\ \boldsymbol{M}^{\mathcal{H}\mathcal{D}} & \boldsymbol{M}^{\mathcal{H}\mathcal{H}} \end{bmatrix} - \lambda \begin{bmatrix} \boldsymbol{L}^{\mathcal{D}\mathcal{D}} & \boldsymbol{L}^{\mathcal{D}\mathcal{H}} \\ \boldsymbol{L}^{\mathcal{H}\mathcal{D}} & \boldsymbol{L}^{\mathcal{H}\mathcal{H}} \end{bmatrix} \right) \begin{bmatrix} \boldsymbol{y}^{\mathcal{D}} \\ \boldsymbol{y}^{\mathcal{H}} \end{bmatrix} = \begin{bmatrix} \boldsymbol{H}^{\mathcal{D}\mathcal{D}} & \boldsymbol{H}^{\mathcal{D}\mathcal{H}} \\ \boldsymbol{H}^{\mathcal{H}\mathcal{D}} & \boldsymbol{H}^{\mathcal{H}\mathcal{H}} \end{bmatrix} \begin{bmatrix} \boldsymbol{y}^{\mathcal{D}} \\ \boldsymbol{y}^{\mathcal{H}} \end{bmatrix} = \begin{bmatrix} \boldsymbol{b}^{\mathcal{D}} \\ \boldsymbol{b}^{\mathcal{H}} \end{bmatrix}$$

for the unknown vector $\boldsymbol{y}$. This can be done in three steps:

- Set the known solution $\boldsymbol{y}^{\mathcal{D}}$
- Calculate the modified right hand side term $\boldsymbol{b}^{\mathcal{H}} - \boldsymbol{H}^{\mathcal{HD}}\boldsymbol{y}^{\mathcal{D}}$
- Solve the system below for the unknown $\boldsymbol{y}^{\mathcal{H}}$, i.e. $\boldsymbol{H}^{\mathcal{HH}}\boldsymbol{y}^{\mathcal{H}} = \boldsymbol{b}^{\mathcal{H}} - \boldsymbol{H}^{\mathcal{HD}}\boldsymbol{y}^{\mathcal{D}}$

3.2.10 How to add a new time-stepping method

To add a new time integration scheme, follow the steps below:

- Choose a name for the method and add it to the `TimeIntegrationMethod` enum list.
- Populate the switch statement in the `TimeIntegrationScheme` constructor with the coefficients of the new method.
- Use (or modify) the function `InitializeScheme` to select (or implement) a proper initialisation strategy for the method.
- Add documentation for the method (especially indicating what the auxiliary parameters of the input and output vectors of the multi-step method represent)

3.2.11 Examples of already implemented time stepping schemes

Here we show some examples time-stepping shemes implemented in Nektar++, to give an idea of what is required to add one of them.

3.2.11.1 Forward Euler

$$\left[\begin{array}{c|c} A & U \\ \hline B & V \end{array}\right] = \left[\begin{array}{c|c} 0 & 1 \\ \hline 1 & 1 \end{array}\right], \qquad \boldsymbol{y}^{[n]} = \left[\begin{array}{c} \boldsymbol{y}_0^{[n]} \end{array}\right] = \left[\begin{array}{c} \boldsymbol{m}\left(t^n, \boldsymbol{y}^n\right) \end{array}\right]$$

3.2.11.2 Backward Euler

$$\left[\begin{array}{c|c} A & U \\ \hline B & V \end{array}\right] = \left[\begin{array}{c|c} 1 & 1 \\ \hline 1 & 1 \end{array}\right], \qquad \boldsymbol{y}^{[n]} = \left[\begin{array}{c} \boldsymbol{y}_0^{[n]} \end{array}\right] = \left[\begin{array}{c} \boldsymbol{m}\left(t^n, \boldsymbol{y}^n\right) \end{array}\right]$$

3.2.11.3 2nd order Adams Bashforth

$$\left[\begin{array}{c|cc} A & U \\ \hline B & V \end{array}\right] = \left[\begin{array}{c|cc} 0 & 1 & 0 \\ \hline \frac{3}{2} & 1 & \frac{-1}{2} \\ 1 & 0 & 0 \end{array}\right], \qquad \boldsymbol{y}^{[n]} = \left[\begin{array}{c} \boldsymbol{y}_0^{[n]} \\ \boldsymbol{y}_1^{[n]} \end{array}\right] = \left[\begin{array}{c} \boldsymbol{m}\left(t^n, \boldsymbol{y}^n\right) \\ \Delta t \boldsymbol{l}(t^{n-1}, \boldsymbol{y}^{n-1}) \end{array}\right]$$

3.2.11.4 1st order IMEX Euler backwards/ Euler Forwards

$$\left[\begin{array}{cc|c} A^{\mathrm{IM}} & A^{\mathrm{EM}} & U \\ \hline B^{\mathrm{IM}} & B^{\mathrm{EM}} & V \end{array}\right] = \left[\begin{array}{c|c|cc} 1 & 0 & 1 & 1 \\ \hline 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \end{array}\right] \quad \text{with} \quad \boldsymbol{y}^{[n]} = \begin{bmatrix} \boldsymbol{y}_0^{[n]} \\ \boldsymbol{y}_1^{[n]} \end{bmatrix} = \begin{bmatrix} \boldsymbol{m}\left(t^n, \boldsymbol{y}^n\right) \\ \Delta t \boldsymbol{l}\left(t^n, \boldsymbol{y}^n\right) \end{bmatrix}$$

3.2.11.5 2nd order IMEX Backward Different Formula & Extrapolation

$$\left[\begin{array}{cc|c} A^{\mathrm{IM}} & A^{\mathrm{EM}} & U \\ \hline B^{\mathrm{IM}} & B^{\mathrm{EM}} & V \end{array}\right] = \left[\begin{array}{c|c|cccc} \frac{2}{3} & 0 & \frac{4}{3} & -\frac{1}{3} & \frac{4}{3} & -\frac{2}{3} \\ \hline \frac{2}{3} & 0 & \frac{4}{3} & -\frac{1}{3} & \frac{4}{3} & -\frac{2}{3} \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{array}\right]$$

with

$$\boldsymbol{y}^{[n]} = \begin{bmatrix} \boldsymbol{y}_0^{[n]} \\ \boldsymbol{y}_1^{[n]} \\ \boldsymbol{y}_2^{[n]} \\ \boldsymbol{y}_3^{[n]} \end{bmatrix} = \begin{bmatrix} \boldsymbol{m}\left(t^n, \boldsymbol{y}^n\right) \\ \boldsymbol{m}\left(t^{n-1}, \boldsymbol{y}^{n-1}\right) \\ \Delta t \boldsymbol{l}\left(t^n, \boldsymbol{y}^n\right) \\ \Delta t \boldsymbol{l}\left(t^{n-1}, \boldsymbol{y}^{n-1}\right) \end{bmatrix}$$

Note

The first two rows are normalised so the coefficient on $\boldsymbol{y}_n^{[n+1]}$ is one. In the standard formulation it is 3/2.

3.2.11.6 3rdorder IMEX Backward Different Formula & Extrapolation

$$\left[\begin{array}{cc|c} A^{\mathrm{IM}} & A^{\mathrm{EM}} & U \\ \hline B^{\mathrm{IM}} & B^{\mathrm{EM}} & V \end{array}\right] = \left[\begin{array}{c|c|cccccc} \frac{6}{11} & 0 & \frac{18}{11} & -\frac{9}{11} & \frac{2}{11} & \frac{18}{11} & -\frac{18}{11} & \frac{6}{11} \\ \hline \frac{6}{11} & 0 & \frac{18}{11} & -\frac{9}{11} & \frac{2}{11} & \frac{18}{11} & -\frac{18}{11} & \frac{6}{11} \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{array}\right]$$

with

$$\boldsymbol{y}^{[n]} = \begin{bmatrix} \boldsymbol{y}_0^{[n]} \\ \boldsymbol{y}_1^{[n]} \\ \boldsymbol{y}_2^{[n]} \\ \boldsymbol{y}_3^{[n]} \\ \boldsymbol{y}_4^{[n]} \\ \boldsymbol{y}_5^{[n]} \end{bmatrix} = \begin{bmatrix} \boldsymbol{m}\left(t^n, \boldsymbol{y}^n\right) \\ \boldsymbol{m}\left(t^{n-1}, \boldsymbol{y}^{n-1}\right) \\ \boldsymbol{m}\left(t^{n-2}, \boldsymbol{y}^{n-2}\right) \\ \Delta t \boldsymbol{l}\left(t^n, \boldsymbol{y}^n\right) \\ \Delta t \boldsymbol{l}\left(t^{n-1}, \boldsymbol{y}^{n-1}\right) \\ \Delta t \boldsymbol{l}\left(t^{n-2}, \boldsymbol{y}^{n-2}\right) \end{bmatrix}$$

Note

The first two rows are normalised so the coefficient on $\boldsymbol{y}_n^{[n+1]}$ is one. In the standard formulation it is 11/6.

3.2.11.7 2nd order Adams Moulton

$$\left[\begin{array}{c|c} A & U \\ \hline B & V \end{array}\right] = \left[\begin{array}{cc|c} \frac{1}{2} & 1 & \frac{1}{2} \\ \hline \frac{1}{2} & 1 & \frac{1}{2} \\ 1 & 0 & 0 \end{array}\right], \qquad \boldsymbol{y}^{[n]} = \left[\begin{array}{c} \boldsymbol{y}_0^{[n]} \\ \boldsymbol{y}_1^{[n]} \end{array}\right] = \left[\begin{array}{c} \boldsymbol{m}\left(t^n, \boldsymbol{y}^n\right) \\ \Delta t \boldsymbol{l}(t^n, \boldsymbol{y}^n) \end{array}\right]$$

3.2.11.8 Midpoint method

$$\left[\begin{array}{c|c} A & U \\ \hline B & V \end{array}\right] = \left[\begin{array}{cc|c} 0 & 0 & 1 \\ \frac{1}{2} & 0 & 1 \\ \hline 0 & 1 & 1 \end{array}\right], \qquad \boldsymbol{y}^{[n]} = \left[\begin{array}{c} \boldsymbol{y}_0^{[n]} \end{array}\right] = \left[\begin{array}{c} \boldsymbol{m}\left(t^n, \boldsymbol{y}^n\right) \end{array}\right]$$

3.2.11.9 RK4: the standard fourth order Runge-Kutta scheme

$$\left[\begin{array}{c|c} A & U \\ \hline B & V \end{array}\right] = \left[\begin{array}{cccc|c} 0 & 0 & 0 & 0 & 1 \\ \frac{1}{2} & 0 & 0 & 0 & 1 \\ 0 & \frac{1}{2} & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ \hline \frac{1}{6} & \frac{1}{3} & \frac{1}{3} & \frac{1}{6} & 1 \end{array}\right], \qquad \boldsymbol{y}^{[n]} = \left[\begin{array}{c} \boldsymbol{y}_0^{[n]} \end{array}\right] = \left[\begin{array}{c} \boldsymbol{m}\left(t^n, \boldsymbol{y}^n\right) \end{array}\right]$$

3.2.11.10 2nd order Diagonally Implicit Runge Kutta (DIRK)

$$\left[\begin{array}{c|c} A & U \\ \hline B & V \end{array}\right] = \left[\begin{array}{cc|c} \lambda & 0 & 1 \\ (1-\lambda) & \lambda & 1 \\ \hline (1-\lambda) & \lambda & 1 \end{array}\right] \quad \text{with} \quad \lambda = \frac{2-\sqrt{2}}{2}, \qquad \boldsymbol{y}^{[n]} = \left[\begin{array}{c} \boldsymbol{y}_0^{[n]} \end{array}\right] = \left[\begin{array}{c} \boldsymbol{m}\left(t^n, \boldsymbol{y}^n\right) \end{array}\right]$$

3.2.11.11 3rd order Diagonally Implicit Runge Kutta (DIRK)

$$\left[\begin{array}{c|c} A & U \\ \hline B & V \end{array}\right] = \left[\begin{array}{ccc|c} \lambda & 0 & 0 & 1 \\ \frac{1}{2}(1-\lambda) & \lambda & 0 & 1 \\ \frac{1}{4}\left(-6\lambda^2+16\lambda-1\right) & \frac{1}{4}\left(6\lambda^2-20\lambda+5\right) & \lambda & 1 \\ \hline \frac{1}{4}\left(-6\lambda^2+16\lambda-1\right) & \frac{1}{4}\left(6\lambda^2-20\lambda+5\right) & \lambda & 1 \end{array}\right]$$

with

$$\lambda = 0.4358665215, \qquad \boldsymbol{y}^{[n]} = \left[\begin{array}{c} \boldsymbol{y}_0^{[n]} \end{array}\right] = \left[\begin{array}{c} \boldsymbol{m}\left(t^n, \boldsymbol{y}^n\right) \end{array}\right]$$

3.2.11.12 3rd order L-stable, three stage IMEX DIRK(3,4,3)

$$\left[\begin{array}{cc|c} A^{\mathrm{IM}} & A^{\mathrm{EM}} & U \\ \hline B^{\mathrm{IM}} & B^{\mathrm{EM}} & V \end{array}\right] =$$

$$\left[\begin{array}{cccc|cccc|c} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & \lambda & 0 & 0 & \lambda & 0 & 0 & 0 & 1 \\ 0 & \frac{1}{2}(1-\lambda) & \lambda & 0 & 0.3212788860 & 0.3966543747 & 0 & 0 & 1 \\ 0 & \frac{1}{4}\left(-6\lambda^2+16\lambda-1\right) & \frac{1}{4}\left(6\lambda^2-20\lambda+5\right) & \lambda & -0.105858296 & 0.5529291479 & 0.5529291479 & 0 & 1 \\ \hline 0 & \frac{1}{4}\left(-6\lambda^2+16\lambda-1\right) & \frac{1}{4}\left(6\lambda^2-20\lambda+5\right) & \lambda & 0 & \frac{1}{4}\left(-6\lambda^2+16\lambda-1\right) & \frac{1}{4}\left(6\lambda^2-20\lambda+5\right) & \lambda & 1 \end{array}\right]$$

with

$$\lambda = 0.4358665215, \qquad \boldsymbol{y}^{[n]} = \left[\begin{array}{c} \boldsymbol{y}_0^{[n]} \end{array}\right] = \left[\begin{array}{c} \boldsymbol{m}\left(t^n, \boldsymbol{y}^n\right) \end{array}\right]$$

3.3 Preconditioners

Most of the solvers in *Nektar++*, including the incompressible Navier-Stokes equations, rely on the solution of a Helmholtz equation,

$$\nabla^2 u(\mathbf{x}) + \lambda u(\mathbf{x}) = f(\mathbf{x}), \tag{3.1}$$

an elliptic boundary value problem, at every time-step, where u is defined on a domain Ω of N_{el} non-overlapping elements. In this section, we outline the preconditioners which are implemented in *Nektar++*. Whilst some of the preconditioners are generic, many are especially designed for the *modified* basis only.

3.3.1 Mathematical formulation

The standard spectral/*hp* approach to discretise (3.1) starts with an expansion in terms of the elemental modes:

$$u^{\delta}(\mathbf{x}) = \sum_{n=0}^{N_{\mathrm{dof}}-1} \hat{u}_n \Phi_n(\mathbf{x}) = \sum_{e=1}^{N_{\mathrm{el}}} \sum_{n=0}^{N_m^e-1} \hat{u}_n^e \phi_n^e(\mathbf{x}) \tag{3.2}$$

where N_{el} is the number of elements, N_m^e is the number of local expansion modes within the element Ω^e, $\phi_n^e(\mathbf{x})$ is the n^{th} local expansion mode within the element Ω^e, $\hat{u}_n^e$ is the n^{th} local expansion coefficient within the element Ω^e. Approximating our solution by (3.2), we adopt a Galerkin discretisation of equation (3.1) where for an appropriate test space V^{δ} we find an approximate solution $\mathbf{u}^{\delta} \in V^{\delta}$ such that

$$\mathcal{L}(v,u) = \int_{\Omega} \nabla v^{\delta} \cdot \nabla u^{\delta} + \lambda v^{\delta} u^{\delta} d\mathbf{x} = \int_{\Omega} v^{\delta} f d\mathbf{x} \quad \forall v^{\delta} \in V^{\delta}$$

This can be formulated in matrix terms as

$$\mathbf{H}\hat{\mathbf{u}} = \mathbf{f}$$

where $\mathbf{H}$ represents the Helmholtz matrix, $\hat{\mathbf{u}}$ are the unknown global coefficients and $\mathbf{f}$ the inner product the expansion basis with the forcing function.

3.3.1.1 C^0 formulation

We first consider the C^0 (i.e. continuous Galerkin) formulation. The spectral/hp expansion basis is obtained by considering interior modes, which have support in the interior of the element, separately from boundary modes which are non-zero on the boundary of the element. We align the boundary modes across the interface of the elements to obtain a continuous global solution. The boundary modes can be further decomposed into vertex, edge and face modes, defined as follows:

- vertex modes have support on a single vertex and the three adjacent edges and faces as well as the interior of the element;
- edge modes have support on a single edge and two adjacent faces as well as the interior of the element;
- face modes have support on a single face and the interior of the element.

When the discretisation is continuous, this strong coupling between vertices, edges and faces leads to a matrix of high condition number κ. Our aim is to reduce this condition number by applying specialised preconditioners. Utilising the above mentioned decomposition, we can write the matrix equation as:

$$\begin{bmatrix} \mathbf{H}_{bb} & \mathbf{H}_{bi} \\ \mathbf{H}_{ib} & \mathbf{H}_{ii} \end{bmatrix} \begin{bmatrix} \hat{\mathbf{u}}_b \\ \hat{\mathbf{u}}_i \end{bmatrix} = \begin{bmatrix} \hat{\mathbf{f}}_b \\ \hat{\mathbf{f}}_i \end{bmatrix}$$

where the subscripts b and i denote the boundary and interior degrees of freedom respectively. This system then can be statically condensed allowing us to solve for the boundary and interior degrees of freedom in a decoupled manor. The statically condensed matrix is given by

$$\begin{bmatrix} \mathbf{H}_{bb} - \mathbf{H}_{bi}\mathbf{H}_{ii}^{-1}\mathbf{H}_{ib} & 0 \\ \mathbf{H}_{ib} & \mathbf{H}_{ii} \end{bmatrix} \begin{bmatrix} \hat{\mathbf{u}}_b \\ \hat{\mathbf{u}}_i \end{bmatrix} = \begin{bmatrix} \hat{\mathbf{f}}_b - \mathbf{H}_{bi}\mathbf{H}_{ii}^{-1}\hat{\mathbf{f}}_i \\ \hat{\mathbf{f}}_i \end{bmatrix}$$

This is highly advantageous since by definition of our interior expansion this vanishes on the boundary, and so $\mathbf{H}_{ii}$ is block diagonal and thus can be easily inverted. The above sub-structuring has reduced our problem to solving the boundary problem:

$$\mathbf{S}_1\hat{\mathbf{u}} = \hat{\mathbf{f}}_1$$

where $\mathbf{S_1} = \mathbf{H}_{bb} - \mathbf{H}_{bi}\mathbf{H}_{ii}^{-1}\mathbf{H}_{ib}$ and $\hat{\mathbf{f}}_1 = \hat{\mathbf{f}}_b - \mathbf{H}_{bi}\mathbf{H}_{ii}^{-1}\hat{\mathbf{f}}_i$. Although this new system typically has better convergence properties (i.e lower κ), the system is still ill-conditioned, leading to a convergence rate of the conjugate gradient (CG) routine that is prohibitively slow. For this reason we need to precondition $\mathbf{S}_1$. To do this we solve an equivalent system of the form:

$$\mathbf{M}^{-1}\left(\mathbf{S}_1\hat{\mathbf{u}} - \hat{\mathbf{f}}_1\right) = 0$$

where the preconditioning matrix $\mathbf{M}$ is such that $\kappa\left(\mathbf{M}^{-1}\mathbf{S}_1\right)$ is less than $\kappa\left(\mathbf{S}_1\right)$ and speeds up the convergence rate. Within the conjugate gradient routine the same preconditioner $\mathbf{M}$ is applied to the residual vector $\hat{\mathbf{r}}_{k+1}$ of the CG routine every iteration:

$$\hat{\mathbf{z}}_{k+1} = \mathbf{M}^{-1}\hat{\mathbf{r}}_{k+1}.$$

3.3.1.2 HDG formulation

When utilising a hybridizable discontinuous Galerkin formulation, we perform a static condensation approach but in a discontinuous framework, which for brevity we omit here. However, we still obtain a matrix equation of the form

$$\mathbf{\Lambda}\hat{\mathbf{u}} = \hat{\mathbf{f}}.$$

where $\mathbf{\Lambda}$ represents an operator which projects the solution of each face back onto the three-dimensional element or edge onto the two-dimensional element. In this setting then, $\hat{\mathbf{f}}$ consists of degrees of freedom for each egde (in 2D) or face (in 3D). The overall system does not, therefore, results in a weaker coupling between degrees of freedom, but at the expense of a larger matrix system.

3.3.2 Preconditioners

Within the *Nektar++* framework a number of preconditioners are available to speed up the convergence rate of the conjugate gradient routine. The table below summarises each method, the dimensions of elements which are supported, and also the discretisation type support which can either be continuous (CG) or discontinuous (hybridizable DG).

Name	Dimensions	Discretisations
`Null`	All	All
`Diagonal`	All	All
`FullLinearSpace`	2/3D	CG
`LowEnergyBlock`	3D	CG
`Block`	2/3D	All
`FullLinearSpaceWithDiagonal`	All	CG
`FullLinearSpaceWithLowEnergyBlock`	2/3D	CG
`FullLinearSpaceWithBlock`	2/3D	CG

The default is the `Diagonal` preconditioner. The above preconditioners are specified through the `Preconditioner` option of the `SOLVERINFO` section in the session file. For example, to enable `FullLinearSpace` one can use:

```
1 <I PROPERTY="Preconditioner" VALUE="FullLinearSpace" />
```

Alternatively one can have more control over different preconditioners for each solution field by using the `GlobalSysSoln` section. For more details, consult the user guide. The following sections specify the details for each method.

3.3.2.1 Diagonal

Diagonal (or Jacobi) preconditioning is amongst the simplest preconditioning strategies. In this scheme one takes the global matrix $\mathbf{H} = (h_{ij})$ and computes the diagonal terms h_{ii}. The preconditioner is then formed as a diagonal matrix $\mathbf{M}^{-1} = (h_{ii}^{-1})$.

3.3.2.2 Linear space

The linear space (or coarse space) of the matrix system is that containing degrees of freedom corresponding only to the vertex modes in the high-order system. Preconditioning of this space is achieved by forming the matrix corresponding to the coarse space and inverting it, so that

$$\mathbf{M}^{-1} = (\mathbf{S}_1^{-1})_{vv}$$

Since the mesh associated with higher order methods is relatively coarse compared with traditional finite element discretisations, the linear space can usually be directly inverted without memory issues. However such a methodology can be prohibitive on large parallel systems, due to a bottleneck in communication.

In *Nektar++* the inversion of the linear space present is handled using the XX^T library. XX^T is a parallel direct solver for problems of the form $\mathbf{A}\hat{\mathbf{x}} = \hat{\mathbf{b}}$ based around a sparse factorisation of the inverse of $\mathbf{A}$. To precondition utilising this methodology the linear sub-space is gathered from the expansion and the preconditioned residual within the CG routine is determined by solving

$$(\mathbf{S}_1)_{vv}\hat{\mathbf{z}} = \hat{\mathbf{r}}$$

The preconditioned residual $\hat{\mathbf{z}}$ is then scattered back to the respective location in the global degrees of freedom.

3.3.2.3 Block

Block preconditioning of the C^0 continuous system is defined by the following:

$$\mathbf{M}^{-1} = \begin{bmatrix} (\mathbf{S}_1^{-1})_{vv} & 0 & 0 \\ 0 & (\mathbf{S}_1^{-1})_{eb} & 0 \\ 0 & 0 & (\mathbf{S}_1^{-1})_{ef} \end{bmatrix}$$

where $\mathrm{diag}[(\mathbf{S}_1)_{vv}]$ is the diagonal of the vertex modes, $(\mathbf{S}_1)_{eb}$ and $(\mathbf{S}_1)_{fb}$ are block diagonal matrices corresponding to coupling of an edge (or face) with itself i.e ignoring the coupling to other edges and faces. This preconditioner is best suited for two dimensional problems.

In the HDG system, we take the block corresponding to each face and invert it. Each of these inverse blocks then forms one of the diagonal components of the block matrix $\mathbf{M}^{-1}$.

3.3.3 Low energy

Low energy basis preconditioning follows the methodology proposed by Sherwin & Casarin. In this method a new basis is numerically constructed from the original basis which allows the Schur complement matrix to be preconditioned using a block preconditioner. The method is outlined briefly in the following.

Elementally the local approximation $\mathbf{u}^\delta$ can be expressed as different expansions lying in the same discrete space V^δ

$$\mathbf{u}^\delta(\mathbf{x}) = \sum_i^{\dim(V^\delta)} \hat{u}_{1i}\phi_{1i}(x) = \sum_i^{\dim(V^\delta)} \hat{u}_{2i}\phi_{2j}(x)$$

Since both expansions lie in the same space it's possible to express one basis in terms of the other via a transformation, i.e.

$$\phi_2 = \mathbf{C}\phi_1 \implies \hat{\mathbf{u}}_1 = C^T\hat{\mathbf{u}}_2$$

Applying this to the Helmholtz operator it is possible to show that,

$$\mathbf{H}_2 = \mathbf{C}\mathbf{H}_1\mathbf{C}^T$$

For sub-structured matrices ($\mathbf{S}$) the transformation matrix ($\mathbf{C}$) becomes:

$$\mathbf{C} = \begin{bmatrix} \mathbf{R} & 0 \\ 0 & \mathbf{I} \end{bmatrix}$$

Hence the transformation in terms of the Schur complement matrices is:

$$\mathbf{S}_2 = \mathbf{R}\mathbf{S}_1\mathbf{R}^T$$

Typically the choice of expansion basis ϕ_1 can lead to a Helmholtz matrix that has undesirable properties i.e poor condition number. By choosing a suitable transformation matrix $\mathbf{C}$ it is possible to construct a new basis, numerically, that is amenable to block diagonal preconditioning.

$$\mathbf{S}_1 = \begin{bmatrix} \mathbf{S}_{vv} & \mathbf{S}_{ve} & \mathbf{S}_{vf} \\ \mathbf{S}_{ve}^T & \mathbf{S}_{ee} & \mathbf{S}_{ef} \\ \mathbf{S}_{vf}^T & \mathbf{S}_{ef}^T & \mathbf{S}_{ff} \end{bmatrix} = \begin{bmatrix} \mathbf{S}_{vv} & \mathbf{S}_{v,ef} \\ \mathbf{S}_{v,ef}^T & \mathbf{S}_{ef,ef} \end{bmatrix}$$

Applying the transformation $\mathbf{S}_2 = \mathbf{R}\mathbf{S}_1\mathbf{R}^T$ leads to the following matrix

$$\mathbf{S}_2 = \begin{bmatrix} \mathbf{S}_{vv} + \mathbf{R}_v\mathbf{S}_{v,ef}^T + \mathbf{S}_{v,ef}\mathbf{R}_v^T + \mathbf{R}_v\mathbf{S}_{ef,ef}\mathbf{R}_v^T & [\mathbf{S}_{v,ef} + \mathbf{R}_v\mathbf{S}_{ef,ef}]\mathbf{A}^T \\ \mathbf{A}[\mathbf{S}_{v,ef}^T + \mathbf{S}_{ef,ef}\mathbf{R}_v^T] & \mathbf{A}\mathbf{S}_{ef,ef}\mathbf{A}^T \end{bmatrix}$$

where $\mathbf{A}\mathbf{S}_{ef,ef}\mathbf{A}^T$ is given by

$$\mathbf{A}\mathbf{S}_{ef,ef}\mathbf{A}^T = \begin{bmatrix} \mathbf{S}_{ee} + \mathbf{R}_{ef}\mathbf{S}_{ef}^T + \mathbf{S}_{ef}\mathbf{R}_{ef}^T + \mathbf{R}_{ef}\mathbf{S}_{ff}\mathbf{R}_{ef}^T & \mathbf{S}_{ef} + \mathbf{R}_{ef}\mathbf{S}_{ff} \\ \mathbf{S}_{ef}^T + \mathbf{S}_{ff}\mathbf{R}_{ef}^T & \mathbf{S}_{ff} \end{bmatrix}$$

To orthogonalise the vertex-edge and vertex-face modes, it can be seen from the above that

$$\mathbf{R}_{ef}^T = -\mathbf{S}_{ff}^{-1}\mathbf{S}_{ef}^T$$

and for the edge-face modes:

$$\mathbf{R}_{v}^T = -\mathbf{S}_{ef,ef}^{-1}\mathbf{S}_{v,ef}^T$$

Here it is important to consider the form of the expansion basis since the presence of $\mathbf{S}_{ff}^{-1}$ will lead to a new basis which has support on all other faces; this is problematic when creating a C^0 continuous global basis. To circumvent this problem when forming the new basis, the decoupling is only performed between a specific edge and the two adjacent faces in a symmetric standard region. Since the decoupling is performed in a rotationally symmetric standard region the basis does not take into account the Jacobian mapping between the local element and global coordinates, hence the final expansion will not be completely orthogonal.

The low energy basis creates a Schur complement matrix that although it is not completely orthogonal can be spectrally approximated by its block diagonal contribution. The final form of the preconditioner is:

$$\mathbf{M}^{-1} = \begin{bmatrix} \text{diag}[(\mathbf{S}_2)_{vv}] & 0 & 0 \\ 0 & (\mathbf{S}_2)_{eb} & 0 \\ 0 & 0 & (\mathbf{S}_2)_{fb} \end{bmatrix}^{-1}$$

where $\text{diag}[(\mathbf{S}_2)_{vv}]$ is the diagonal of the vertex modes, $(\mathbf{S}_2)_{eb}$ and $(\mathbf{S}_2)_{fb}$ are block diagonal matrices corresponding to coupling of an edge (or face) with itself i.e ignoring the coupling to other edges and faces.

Chapter 4

Coding Standard

The purpose of this page is to detail the coding standards of the project which all contributers are requested to follow.

This page describes the coding style standard for C++. A coding style standard defines the visual layout of source code. Presenting source code in a uniform fashion facilitates the use of code by different developers. In addition, following a standard prevents certain types of coding errors.

All of the items below, unless otherwise noted, are guidelines. They are recommendations about how to lay out a given block of code. Use common sense and provide comments to describe any deviation from the standard. Sometimes, violating a guideline may actually improve readability.

If you are working with code that does not follow the standard, bring the code up-to-date or follow the existing style. Don't mix styles.

4.1 Code Layout

The aim here is to maximise readability on all platforms and editors.

- Code width of 80 characters maximum - hard-wrap longer lines.
- Use sensible wrapping for long statements in a way which maximises readability.
- Do not put multiple statements on the same line.
- Do not declare multiple variables on the same line.
- Provide a default value on all variable declarations.
- Enclose every program block (if, else, for, while, etc) in braces, even if empty or just a single line.

- Opening braces ({) should be on their own line.
- Braces at same indentation as preceeding statement.
- One class per .cpp and .h file only, unless nested.
- Define member functions in the .cpp file in the same order as defined in the .h file.
- Templated classes defined and implemented in a single .hpp file.
- Do not put inline functions in the header file unless the function is trivial (e.g. accessor, empty destructor), or profiling explicitly suggests to.
- Inline functions should be declared within the class declaration but defined outside the class declaration at the bottom of the header file.

Note

Virtual and inline are mutually exclusive. Virtual functions should therefore be implemented in the .cpp file.

4.2 Space

Adding an appropriate amount of white space enhances readability. Too much white space, on the other hand, detracts from that readability.

- Indent using a four-space tab. Consistent tab spacing is necessary to maintain formatting. Note that this means when a tab is pressed, four physical spaces are inserted into the source instead.
- Put a blank line at the end of a public/protected/private block.
- Put a blank line at the end of every file.
- Put a space after every keyword (if, while, for, etc.).
- Put a space after every comma, unless the comma is at the end of the line.
- Do not put a space before the opening parenthesis of an argument list to a function.
- Declare pointers and references with the * or & symbol next to the declarator, not the type; e.g., Object *object. Do not put multiple variables in the same declaration.
- Place a space on both sides of a binary operator.
- Do not use a space to separate a unary operator from its operand.

- Place open and close braces on their own line. No executable statements should appear on the line with the brace, but comments are allowed. Indent opening braces at the same level as the statement above and indent the closing brace at the same level as the corresponding opening brace.

- Indent all statements following an open brace by one tab. Developer Studio puts any specifier terminated with a colon at the same indentation level as the enclosing brace. Examples of such specifiers include case statements, access specifiers (public, private, protected), and goto labels. This is not acceptable and should be manually corrected so that all statements appearing within a block and delineated by braces are indented.

- Break a line into multiple lines when it becomes too long to read. Use at least two tabs to start the new line, so it does not look like the start of a block.

- Follow C++ style comments with one space. It is also preferable to consider any text that follows C++ style comments as a sentence and to begin this text with a capital letter. This helps to distinguish the line from a continuation of a previous line; i.e., `// This is my comment.`

- As a general rule, don't keep commented out source code in the final baselined product. Such code leads the reader to believe there was uncertainty in the code as it currently exists.

- Place the # of a preprocessor directive at column one. An exception is the use of nested ifdefs where the bodies only contain other preprocessor directives. Add tabs to enhance readability:

```
1 void foo() {
2     for(int i = 0; i < 10; ++i)
3     {
4 #ifdef BAR
5         do_something();
6 #endif
7         for_loop_code();
8     }
9 }
```

- Use tabular white space if it enhances readability.

- Use only one return statement. Structure the code so that only one return statement is necessary.

4.3 Naming Conventions

Keep variable and function names meaningful but concise.

- Begin variable names with lower-case letter.

- Begin function names and class names with upper-case letter.
- All function, class and variable names should be written in CamelCase, e.g. `MyClass, DoFunction() or myVariableName`.
- All preprocessor definitions written in UPPER_CASE with words separated by underscores, e.g. USE_SPECIFIC_FEATURE.
- All member variables prefixed with m_.
- All constants prefixed with a k.
- All function parameters prefixed with a p.
- All enumerations prefixed with an e.
- Do not use leading underscores.

4.4 Namespaces

The top-level namespace is "Nektar". All code should reside in this namespace or a sub-space of this.

- Namespaces correspond to code structure.
- Namespaces should be kept to a minimum to simplify the interface to their contents.

4.5 Documentation

- Briefs for classes, functions and types in header files using `///` notation.
- Full documentation with implementation using `/** ... *` notation.
- Use @ symbol for @class, @param, @returns, etc for ease of identification.
- Any separate documentation pages not directly associated with a portion of the code should be in a separate file in /docs/html/doxygen.

Bibliography

[1] Milton Abramowitz and Irene A Stegun. *Handbook of mathematical functions*. Dover, 1972.

[2] J. C. Butcher. General linear methods. *Acta Numerica*, 15:157–256, 2006.

[3] CD Cantwell, D Moxey, A Comerford, A Bolis, G Rocco, G Mengaldo, D De Grazia, S Yakovlev, J-E Lombard, D Ekelschot, et al. Nektar++: An open-source spectral/hp element framework. *Computer Physics Communications*, 192:205–219, 2015.

[4] Claudio Canuto, M Yousuff Hussaini, Alfio Quarteroni, and Thomas A Zang. Spectral methods in fluid dynamics. Technical report, Springer, 1988.

[5] Alessandro Ghizzetti and Aldo Ossicini. Quadrature formulae. 1970.

[6] G. E. Karniadakis and S. J. Sherwin. *Spectral/hp Element Methods for Computational Fluid Dynamics*. Oxford Science Publications, 2005.

[7] Gabor Szegö. *Orthogonal polynomials*, volume 23. American Mathematical Soc., 1939.

[8] P. E. J. Vos, C. Eskilsson, A. Bolis, S. Chun, R. M. Kirby, and S. J. Sherwin. A generic framework for time-stepping partial differential equations (pdes): general linear methods, object-oriented implementation and application to fluid problems. *International Journal of Computational Fluid Dynamics*, 25:107–125, 2011.