

A Programmer's Guide to Nektar++

Developer's Guide

Editors: Robert M. (Mike) Kirby, Spencer J. Sherwin, Chris D. Cantwell and David Moxey

A distributed working draft for further contribution by the Nektar++ community

January 20, 2021

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

Contents

Contents **2**

1 Preface **7**

2 Introduction **11**
2.1 The *Ethos* of Nektar++ . . . 12
2.2 The Structure of Nektar++ . . . 14
2.3 Assumed Proficiencies . . . 18
2.4 Other Software Implementations and Frameworks . . . 19
2.5 How to Use This Document . . . 20

3 Preliminaries **22**
3.1 Summary of Development Tools and Best Practices . . . 22
3.1.1 General Resources . . . 22
3.1.2 Version Control (git) . . . 23
3.1.3 Building (CMake) . . . 26
3.1.4 Testing (CTest) . . . 26
3.1.5 Merge Requests (Gitlab) . . . 26
3.2 Documentation and Tutorials . . . 26
3.2.1 Dependencies . . . 26
3.2.2 Compiling the User Guide . . . 27
3.2.3 Developers Guide . . . 27
3.2.4 Compiling the code documentation . . . 28
3.3 Compiling Tutorials . . . 28
3.4 Core Nektar++ Programming Concepts . . . 28
3.4.1 Namespaces . . . 29
3.4.2 C++ Standard Template Library (STL) . . . 29
3.4.3 typedefs . . . 30
3.4.4 Forward Declarations . . . 30
3.4.5 Templated Classes and Specialization . . . 31
3.4.6 Multiple Inheritance and the virtual Keyword . . . 32

3.4.7 Virtual Functions and Inheritance . . . 32
3.4.8 Const keyword . . . 32
3.4.9 Function pointers and bind . . . 32
3.4.10 Memory Pools and NekArray . . . 33
3.5 Design Patterns . . . 34
3.5.1 Template pattern . . . 34
3.5.2 Abstract Factory Pattern . . . 34
3.6 Software Testing Approaches . . . 38
3.6.1 Unit Tests . . . 38
3.6.2 Integration, System and Regression Tests . . . 39
3.6.3 Continuous Integration . . . 39

I Building-Blocks of Our Framework (Inside the Library) 40

4 Inside the Library: LibUtilities 41
4.1 BasicConst . . . 41
4.2 BasicUtils . . . 42
4.3 Communication . . . 42
4.4 FFT . . . 43
4.5 Foundations . . . 43
4.5.1 Points . . . 44
4.5.2 Basis . . . 48
4.6 Interpreter . . . 49
4.7 Kernel . . . 49
4.8 Linear Algebra . . . 50
4.9 Memory . . . 51
4.10 Polylib . . . 51
4.11 Time Integration . . . 53
4.11.1 General Linear Methods . . . 54
4.11.2 Introduction . . . 54
4.11.3 Formulation . . . 54
4.11.4 Matrix notation . . . 55
4.11.5 General Linear Methods in Nektar++ . . . 55
4.11.6 Types of time Integration Schemes . . . 56
4.11.7 Usage . . . 57
4.11.8 Implementation of a time-dependent problem . . . 59
4.11.9 Strongly imposed essential boundary conditions . . . 62
4.11.10 How to add a new time-stepping method . . . 63
4.11.11 Examples of already implemented time stepping schemes . . . 63

5 Inside the Library: StdRegions 67
5.1 The Fundamentals Behind StdRegions . . . 67
5.1.1 Reference Element Transformations That Facilitate Separability . . . 69
5.1.2 Reference Elements On Primitive Geometric Types . . . 70

5.2 The Fundamental Data Structures within StdRegions . . . 72
5.2.1 Variables at the Level of StdExpansion . . . 73
5.2.2 Variables at the Level of StdExpansion$D for various Dimensions . 74
5.2.3 Variables at the Level of Shape-Specific StdExpansions . . . 75
5.2.4 General Layout of the Basis Functions in Memory . . . 75
5.2.5 General Layout . . . 75
5.2.6 2D Geometries . . . 76
5.2.7 3D Geometries . . . 77
5.3 The Fundamental Algorithms within StdRegions . . . 79

6 Inside the Library: SpatialDomains 80
6.1 The Fundamentals Behind SpatialDomains . . . 81
6.1.1 Vertex and Curvature Mapping Information . . . 82
6.1.2 Regular Mappings: Geometric Factors . . . 85
6.1.3 Deformed Mappings: Geometric Factors . . . 86
6.2 The Fundamental Data Structures within SpatialDomains . . . 87
6.2.1 Variables at the Level of Geometry . . . 88
6.2.2 Variables at the Level of Geometry$D for various Dimensions . . . 88
6.2.3 Variables at the Level of Shape-Specific Geometry Information . . 88
6.2.4 Reference to World-Space Mapping . . . 88
6.2.5 MeshGraph . . . 88
6.3 The Fundamental Algorithms within SpatialDomains . . . 88

7 Inside the Library: LocalRegions 90
7.1 The Fundamentals Behind LocalRegions . . . 90
7.2 The Fundamental Data Structures within LocalRegions . . . 90
7.2.1 Variables at the Level of Expansion . . . 91
7.2.2 Variables at the Level of Expansion$D for various Dimensions . . . 92
7.2.3 Variables at the Level of Shape-Specific Expansions . . . 92
7.3 The Fundamental Algorithms within LocalRegions . . . 92

8 Inside the Library: Collections 94
8.1 The Fundamentals Behind Collections . . . 94
8.2 The Fundamental Data Structures within Collections . . . 95
8.3 The Fundamental Algorithms within Collections . . . 95

9 Inside the Library: MultiRegions 97
9.1 The Fundamentals Behind MultiRegions . . . 97
9.2 The Fundamental Data Structures within MultiRegions . . . 99
9.2.1 Variables at the Level of ExpList . . . 100
9.2.2 Variables at the Level of ExpList$D for various Dimensions . . . 100
9.2.3 Variables at the Level of Discontinuous Field Expansions . . . 101
9.2.4 Variables at the Level of Continuous Field Expansions . . . 101
9.3 The Fundamental Algorithms within MultiRegions . . . 101
9.4 Preconditioners . . . 101

9.4.1 Mathematical formulation . . . 102
9.4.2 Preconditioners . . . 103
9.4.3 Low energy . . . 105

10 Inside the Library: GlobalMapping 107
10.1 The Fundamentals Behind GlobalMapping . . . 107
10.1.1 Divergence operator . . . 108
10.1.2 Laplacian operator . . . 109
10.1.3 Anisotropic diffusion . . . 110
10.1.4 Anisotropic Laplacian operator . . . 110
10.2 The Fundamental Data Structures within GlobalMapping . . . 111
10.3 The Fundamental Algorithms within GlobalMapping . . . 111

11 Inside the Library: FieldUtils 112
11.1 The Fundamentals Behind FieldUtils . . . 112
11.2 The Fundamental Data Structures within FieldUtils . . . 112
11.3 The Fundamental Algorithms within FieldUtils . . . 112

12 Inside the Library: SolverUtils 113
12.1 The Fundamentals Behind SolverUtils . . . 113
12.2 The Fundamental Data Structures within SolverUtils . . . 113
12.3 The Fundamental Algorithms within SolverUtils . . . 113

II Solvers 114

13 ADRSolver: Solving the Advection-Reaction-Diffusion Equation 116

14 IncNavierStokesSolver: Solving the Incompressible Navier-Stokes Equations 117

15 CompressibleFlowSolver: Solving the Compressible Navier-Stokes Equations 118
15.1 Fundamental Theories of CompressibleFlowSolver . . . 118
15.2 Data Structure of CompressibleFlowSolver . . . 119
15.3 Flow Chart of CompressibleFlowSolver . . . 120

III Utilities 125

16 FieldConvert 127

17 NekMesh 128

IV NekPy: Python interface to *Nektar++* 129

18 Introduction 130
18.1 Features and functionality . . . 130

19 Installing NekPy 132
19.1 Compiling and installing Nektar++ . . . 132
19.1.1 macOS . . . 132
19.1.2 Linux: Ubuntu/Debian . . . 133
19.1.3 Compiling the wrappers . . . 133
19.2 Using the bindings . . . 133
19.2.1 Examples . . . 134

20 Package structure 135

21 NekPy wrapping guide 137
21.1 Defining a library . . . 137
21.2 Basic class wrapping . . . 138
21.2.1 `py::class_<>` . . . 139
21.2.2 Wrapping member functions . . . 139
21.2.3 Inheritance . . . 141
21.2.4 Wrapping enums . . . 142

22 Documentation 143

23 Memory management in NekPy 146
23.1 Memory management in C++ and Python . . . 146
23.1.1 C++ . . . 146
23.2 Passing C++ memory to Python . . . 147
23.2.1 Passing Python data to C++ . . . 153

24 FieldConvert in NekPy 161
24.1 Idea and motivation . . . 161
24.2 Design and implementation . . . 161
24.2.1 `FieldConverter` class . . . 161
24.2.2 User workflow . . . 162
24.2.3 Conversion process . . . 163
24.3 Further development and improvement . . . 163

Bibliography 164

Chapter 1

Preface

Like with any software project, people want to know it origins: what motivated it, what and who drove it, and what constrained it. *Nektar++* officially started as a project idea in 2004 in Salt Lake City, UT, USA, and we registered the first commit to SVN in 2006. The basic backstory is as follows: Mike Kirby (University of Utah) and Spencer Sherwin (Imperial College London) had both studied under George Karniadakis. Though Mike and Spencer did not overlap in terms of their studies (Spencer was a Princeton graduate while Mike was a Brown graduate), they both worked on *Nektar*, a spectral/*hp* element code supervised by George. George's research group has had a long history of involvement in various software projects, e.g. PRISM (which has continued its existence under Professor Hugh Blackburn at Monash University) and *Nektar*. Spencer and George initiated the *Nektar* code with triangular (2D) and tetrahedral (3D) spectral/*hp* elements in the early 1990s, and the code grew and evolved with additions by Dr. Igor Lomtev, Dr. Tim Warburton, Dr. Mike Kirby, etc., all under the PhD advisor direction of George. Spencer graduated from Princeton under George's supervision in 1995 and went on to Imperial College London as a faculty member; Mike graduated from Brown under George's supervision in 2002 and went on to the University of Utah in 2002. In 2004, Mike and Spencer teamed up to re-write *Nektar* in light of modern *C++* programming practices and in light of what had been learned in the decade or so since its foundation.

What were the observations that motivated *Nektar++*? The first observation was that *Nektar*'s origin was triangular and tetrahedral spectral/*hp* elements as applied to incompressible fluid mechanics problems (i.e., the incompressible Navier-Stokes equations). These ideas were extended to the compressible Navier-Stokes by Lomtev and the two parallel paths joined and extended into what is often called Hybrid *Nektar* by Tim Warburton. Hybrid *Nektar* (referred to as *Nektar* from here on out) used the *C++* programming paradigm to facilitate hybrid elements: triangles and quadrilaterals in 2D and tetrahedra, hexahedra, prisms and pyramids in 3D. In addition, Warburton structured the code in a way that allowed extension to the Arbitrary Lagrangian Eulerian (ALE) formulation within *Nektar*, as well as various other features such as the *Nektar* Magneto-Hydrodynamics (MHD) solver. Mike was mentored (as a student) by Warburton

and continued the expansion of *Nektar* (e.g. compressible ALE solver, fluid-structure interaction capabilities, etc.). The expansion of *Nektar*'s capabilities, under the direction of George Karniadakis, continues to this day (e.g., *Stress-Nektar*). The upside of this expansion was that *Nektar* could be expanded and used for solving more and more engineering problems; the downside, in our opinion, was that continually expanding and extending *Nektar* without re-evaluating its fundamental design meant that some components became quite cumbersome from the programming perspective.

The second observation that occurred was that spectral/*hp* elements, as used within the incompressible fluid mechanics world, could be viewed as a special case of the broader set of high-order finite element methods as applied to various fluid and solid mechanics systems. In fact, in a much wider context, these methods represent ways of discretizing various partial differential equations (PDEs) using their weak (variational) form. From this vantage point, one can see the commonality between strong-form methods such as spectral collocation and flux reconstruction, and weak-form methods such as traditional finite elements, spectral elements, and even discontinuous Galerkin methods. During the 1990s, it became more and more apparent that there was a broader context and a broader community for discussing and disseminating the ideas surrounding high-order finite elements, and correspondingly a fruitful environment for cross-pollination of ideas and programming practices.

With all this in mind, Mike and Spencer set out to re-architect *Nektar* from the ground up, and in homage to its *C++* core, this new software suite was called *Nektar++*. *Nektar++* is an open-source software framework designed to support the development of high-performance scalable solvers for partial differential equations using the spectral/*hp* element method. High-order methods are gaining prominence in several engineering and biomedical applications due to their improved accuracy over low-order techniques at reduced computational cost for a given number of degrees of freedom. However, their proliferation is often limited by their complexity, which makes these methods challenging to implement and use. *Nektar++* is an initiative to overcome this limitation by encapsulating the mathematical complexities of the underlying method within an efficient *C++* framework, making the techniques more accessible to the broader scientific and industrial communities. Given the commonalities and connections between various strong-form and weak-form methods and their implementations, we use the term spectral/*hp*methods to refer to the entire family of methods, from traditional FEM to discontinuous Galerkin (dG) to Flux Reconstruction (FR) and beyond.

The software supports a variety of discretization techniques and implementation strategies, supporting methods research as well as application-focused computation, and the multi-layered structure of the framework allows the user to embrace as much or as little of the complexity as they need. The libraries capture the mathematical constructs of spectral/*hp* element methods, while the associated collection of pre-written PDE solvers provides out-of-the-box application-level functionality and a template for users who wish to develop solutions for addressing questions in their own scientific domains.

After five years of laying the groundwork for *Nektar++*, Dr. Chris Cantwell and Dr. David Moxey joined Spencer's group at ICL. Their involvement in the project has greatly impacted the structure and capabilities of *Nektar++*, and has played a significant role in its success. In 2017, we formalized our Development Team structure. Kirby and Sherwin as Founders, along with Cantwell and Moxey, form the key Project Leaders. We established the following roles within the *Nektar++* community:

- User: Individuals or teams who use *Nektar++* as part of their research and who may interact with the community through the mailing list but do not directly contribute code.
- Contributor: Individuals or teams who use *Nektar++* as part of their work but also contribute modifications back into the code which arise as a direct consequence of their research.
- Developer: Individuals who use *Nektar++* for their research, and make code contributions which not only benefit their own research goals but also benefits the wider needs of the *Nektar++* community. Such contributions typically benefit multiple application domains, and developers will make the extra effort to generalize new functionality beyond their own needs. They also fix bugs, identified by others, in areas of the code with which they are familiar.
- Senior Developer: Senior Developers are involved in the development of *Nektar++* beyond their individual research area and interact in more of a transcendent way, making contributions widely across the codebase. Senior developers are entrusted with the tasks of reviewing and merging contributions made by others and maintaining the integrity of the code.
- Project Leader: These individuals meet all the requirements of Senior Developers but in addition direct how *Nektar++* evolves in terms of applications, solvers, library and educational outreach.

This developer's guide, like most of *Nektar++*, is not due to one sole person but an army of people each with different talents, skills and motivations. As we conclude this preface, we will now provide a short biography for the four editors of this volume, and also provide a listing of all the people who have contributed to this document.

Robert M. (Mike) Kirby: Prof. Mike Kirby is a Professor and the Associate Director of the School of Computing at the University of Utah.

Spencer J. Sherwin: Prof. Spencer Sherwin is a Professor of Computational Fluid Mechanics in the Department of Aeronautics at Imperial College London (UK).

Chris D. Cantwell: Dr. Chris Cantwell is a Senior Lecturer in Aeronautics in the Department of Aeronautics at Imperial College London (UK).

David Moxey: Dr. David Moxey is a Senior Lecturer in Engineering in the College of Engineering, Mathematics and Physical Sciences at the University of Exeter (UK).

Contributors: The thank all the various participants in the *Nektar++* Project who have contributed to this document[1]: Dr. Peter Vos (ICL) and Mr. Dav de St. Germain (Utah).

To see the full team and other current details about *Nektar++*, visit our website.

We hope that you find this developer's guide of use to you, that you find *Nektar++* helpful to your educational or industrial pursuits, and that like us, you will actively want to engage and contribute back to the *Nektar++* community.

Mike Kirby
Spencer Sherwin
Chris Cantwell
David Moxey

[1] Affiliation information is in reference to where the person was employed at the time of their contribution.

Chapter 2

Introduction

Nektar++ [15], is an open-source software framework designed to support the development of high-performance scalable solvers for partial differential equations using the spectral/*hp* element method. What in part makes *Nektar++* unique is that it is an initiative to overcome the mathematical complexities of the methods by encapsulating them within an efficient cross-platform *C++* software environment, consequently making the techniques more accessible to the broader scientific and industrial communities. The software supports a variety of discretization techniques and implementation strategies [61, 17, 16, 12], supporting methods research as well as application-focused computation, and the multi-layered structure of the framework allows the user to embrace as much or as little of the complexity as they need. The libraries capture the mathematical constructs of spectral/*hp* element methods in the form of a hierarchy of *C++* components, while the associated collection of pre-written PDE solvers provides out-of-the-box functionality across a range of application areas, as well as a template for users who wish to develop solutions for addressing questions in their own scientific domains.

The cross-platform nature of the software libraries enables the rapid development of solvers for use in a wide variety of computing environments. The code accommodates both small research problems, suitable for desktop computers, and large-scale industrial simulations, requiring modern HPC infrastructure, where there is a need to maintain efficiency and scalability up to many thousands of processor cores.

Nektar++ provides a single codebase with the following key features:

- Arbitrary-order spectral/*hp* element discretisations in one, two and three dimensions;
- Support for variable polynomial order in space and heterogeneous polynomial order within two- and three-dimensional elements;
- High-order representation of the geometry;

- Continuous Galerkin, discontinuous Galerkin and hybridised discontinuous Galerkin projections;
- Support for a Fourier extension of the spectral element mesh;
- Support for a range of linear solvers and preconditioners;
- Multiple implementation strategies for achieving linear algebra performance on a range of platforms;
- Efficient parallel communication using MPI showing strong scaling up to 8192-cores on Archer, the UK national HPC system;
- A range of time integration schemes implemented using generalised linear methods; and
- Cross-platform support for Linux, OS X and Windows operating systems.

In addition to the core functionality listed above, *Nektar++* includes a number of solvers covering a range of application areas. A range of pre-processing and post-processing utilities are also included with support for popular mesh and visualization formats, and an extensive test suite ensures the robustness of the core functionality.

2.1 The *Ethos* of Nektar++

As with any research effort, one is required to decide on a set of guiding principles that will drive the investigation. Similarly with a software development effort of this form, we spent considerable time early on considering which aspects we wanted to be distinctive about *Nektar++* and also what guiding principles would be most suitable to support both the goals and the boundaries of what we wanted to do. We did this for at least two reasons: (1) We acknowledged then and now that there are various software packages and open-source efforts that deal with finite element frameworks, and so we wanted to be able to understand and express to people those things we thought were distinctive to us – that is, our "selling points". (2) We also acknowledged, from our own experience on software projects, that if we did not set up some collection of guiding principles for our work, that we would gravitate towards trying to be "all things to all men", and in doing so be at odds with the first item. Below are a list of the guiding principles, the "ethos", of the Nektar++ software development effort.

Principles: The following are our three guiding principles for *Nektar++* which respect (1) above:

- **Efficiently:** *Nektar++* was to be a "true" high-order code. "True" is put in quotations because we acknowledge that high-order means different things to different communities. Based upon a review of the literature, we came to the conclusion that part of our *h-to-p* philosophy should be that we accommodate

polynomial degrees ranging from zero (finite volumes) or one (traditional linear finite elements) up to what is considered "spectral" (pseudospectral) orders of 16^{th} degree. As part of our early work [61], we established that in order to span this range of polynomial degrees and attempt to maintain some level of computational efficiency, we would need to develop *order-aware* algorithms: that is, we would need to utilize different (equivalent) algorithms appropriate for a particular order. This principle was the starting point of our *h-to-p efficiently* branding and continues to be a driving principle of our work.

- **Transparently:** *Nektar++* was to be agnostic as to what the "right" way to discretize a partial differential equation (PDE). "Right" is put in quotations to acknowledge that like the issue of polynomial order, there appear to be different "camps" who hold very strong views as to which discretization method should be used. Some might concede that continuous Galerkin methods are very natural for elliptic and parabolic PDEs and then work very laboriously to shoehorn all mixed-type PDEs (such as the incompressible Navier-Stokes equations) into this form. This holds true (similarly) for finite volume and dG proponents with regards to hyperbolic problems (and those systems that are dominated by hyperbolicity). Our goal was to be a high-order finite element framework that allowed users to experiment with continuous Galerkin (cG) and discontinuous Galerkin (dG) methods. After our original developments, we incorporated Flux-Reconstruction Methods (FR Methods) into our framework; this confirmed that generality of our approach as our underlying library components merely needed new features added to accommodate the FR perspective on dG methods. As stated earlier, this is in part why we use the term spectral/*hp* as it allows us to capture all these various methods (e.g., cG, dG and FR) under one common element-and-order terminology.

- **Seamlessly:** *Nektar++* was to be a code that could run from the desktop (or laptop) to exascale, seamlessly. It was our observation that many *grand challenge* efforts target petascale and exascale computing, with the idea that in the future what is done now on a supercomputer will be done on the desktop. The supercomputer of today is in a rack within five years and on our desktop in ten years. However, we also acknowledged that many of our end users were interested in computing *now*. That is, following from their engineering tradition they had an engineering or science problem to solve, and they wanted the ability to run on machines ranging from their laptops to exascale as the problem demanded. Not all problems fit on your laptop, and yet not all problems are exascale problems. Like with our *h-to-p efficiently* approach, we wanted our algorithms and code development to allow a range of choices in the hands of the engineer.

Based upon these principles, our long-term vision is a software framework that allows the engineer the flexibility to make all these critical choices (elemental discretization, polynomial order, discretization methodology, algorithms to employ, and architecture-specific details) while at the same time having *smart and adaptive defaults*. That is, we want to accommodate both the engineer that wants control over all the various knobs

that must be set to actualize a simulation run and the engineer that wants to remain at the high-level and let the software system choose what is best in terms of discretization, polynomial order, algorithm choice, etc. We want the flexibility to self tune, while targeting auto-tuning.

The three items above denote the principles of our development efforts. We now present the "guardrails". By guardrails, we mean the guidelines we use to help steer us along towards our goal as stated above. These are not meant to be absolutes necessarily, but are considerations we often use to try to keep us on track in terms of respecting our three guiding principles above.

Guardrails:

- We are principally concerned with advection-reaction-diffusion and conservation law problems. There are many simulation codes that are deal with solid mechanics, and we did not see ourselves as being a competitor with them. As such, we did not initially construct our framework to inherently deal with $H - div$ and $H - curl$ spaces (and their respective element types). Our perspective was to hold a system of scalar fields and, as needed, constrain them to respect certain mathematical properties.

- We rely heavily on the ability to do tensor-product-based quadrature. Although our more recent additions to the Point and Basis information within our library allows for unstructured quadrature, many of our algorithms are designed to try to capitalize on tensor-product properties.

- As we said earlier, we have designed our code to run at a range of orders, from one to sixteen. However, we acknowledge that for many simulation regimes, we often do not run at the lowest and/or the highest orders. For many applications, the sweet spot seems to be polynomials of degree four through eight. At the present time, a reasonable amount of time and effort has been spent optimizing for this range.

- The robustness of our testing is often dictated by the application areas that have funded our code development. At the present time, our incompressible Navier-Stokes solver is probably the most tested of our solvers, followed by our ADR solver. We make every attempt to test our codes thoroughly; however we benefit greatly from various users "stress-testing" our codes and providing feedback on things we can improve (or better yet, suggesting coding solutions).

2.2 The Structure of Nektar++

When we speak of *Nektar++*, it often means different things to different people, all under an umbrella of code. For the founders of *Nektar++*, the view was that the core of *Nektar++* was the library, and that everything else was to be built around or on top of our basic spectral/*hp* framework. For others, the heart of *Nektar++* are its solvers – the

collection of simulation codes, built upon the library, that enable users to solve science and engineering problems. For a smaller group of people, it is all the add-ons that we provide in our utilities, from our mesh generation techniques to our visualization software ideas.

For the purposes of this document, we will structure our discussions into three main parts: library functionality, solvers and utilities. In this section, we will provide an overview of the structure of the libraries. We will start by giving a quick overview of the basic subdirectories contained within *library* and their purpose. We will then provide a bottom-up description of how the library can be viewed, as well as a top-down perspective. Each perspective (bottom-up or top-down) is fully consistent with each other; the advantage of these approaches is that they help the future developer understand the library as someone trying to build up towards our solvers, or conversely someone trying to understand our solver functionality having already been a solver user and now trying to understand the library components on which it was built.

The basic subdirectories with the *library* are as follows:

LibUtilities: This library contains all the basic mathematical and computer science building blocks of the *Nektar++* code.

StdRegions: This library contains the objects that express "standard region" data and operations. In one dimension, this is the StdSegment. In two dimensions, this is the StdTri (Triangle) and StdQuad (Quadrilateral). In three dimensions, this is the StdTet (Tetrahedra), StdHex (Hexahedra), StdPrism (Prism) and StdPyr (Pyramid). These represent the seven different standardized reference regions over which we support differentiation and integration.

SpatialDomains: This library contains the mesh and elemental geometric information. In particular, this part of the library deals with the basic mesh data structures, and the mapping information (such as Jacobians) from StdRegions to LocalRegions.

LocalRegions: This library contains objects that express data and operations on individual physical elements of the computational mesh. Local regions are spectral/*hp* elements in world-space (either straight-/planar-sided or curved-sided). Using *C++* terminology, a local region object *is-a* standard region object and *has-a* spatial domain object.

MultiRegions: This library holds the data structures that represent sets of elements (local regions). At the most fundamental level, these represent the union of local regions into a (geometrically-contiguous) space. By specifying the interaction of these elements, the function space they represent and/or the approximation method is defined. It is at this point in the hierarchy that a set of (local region) elements can be thought of as representing a dG or cG field.

Collections: In this library we amalgamate, in a linear algebra sense, the action of key

operators on multiple (standard region or local region) elements into a single, memory-efficient block. These strategies depend on external factors such as BLAS implementation and the geometry of interest.

GlobalMapping: This library supports the analytical mapping of complex physical domains to simpler computational domains.

NekMeshUtils: This library contains processing modules for the generation (potentially from CAD geometries), conversion and manipulation of high-order meshes.

SolverUtils: This library contains data structures and algorithms which form the basis of solvers, or provide auxiliary functionality.

UnitTests: This part of the library contains unit tests that allow us to verify the correctness of the core functionality within *Nektar++*. These are useful for verifying that new additions or modifications to the lower-levels of the code do not compromise existing functionality or correctness.

Bottom-Up Perspective

The bottom-up perspective on the library is best understood from Figures 2.1 – 2.2. In Figure 2.1, we take the view of understanding the geometric regions over which we build approximations. Our starting point is within the StdRegions library, in which we define our canonical standard regions. There are seven fundamental regions supported by *Nektar++*: segments (1D), triangles and quadrilaterals (2D) and tetrahedra, hexahedra, prisms and pyramids (3D). Since we principally employ Gaussian quadrature, these regions are defined by various tensor-product and collapsing of the compact interval $[-1, 1]$. For the purposes of illustration, let us use a quadrilateral as our example. The StdQuadExp is a region defined on $[-1, 1]\times[-1, 1]$ over which we can build approximations $\phi^e(\xi_1, \xi_2)$. Typically ϕ^e is based upon polynomials in each coordinate direction; using linear functions in both directions yields the traditional $Q(1)$ space in traditional finite elements.

Since $\phi^e(\xi_1, \xi_2)$ lives on $\mathcal{Q} = [-1, 1] \times [-1, 1]$ (i.e. $\phi^e : \mathcal{Q} \to \mathbb{R}$) and is for this example polynomial, we can integrate it exactly (to machine precision) using Gaussian integration, and we can differentiate it by writing it in a Lagrange basis and forming a differential operator matrix to act on values of the function evaluated at points. If the function were not polynomial but instead only a smooth function, we could approximate it with quadrature and decide an appropriate basis by which to approximate its derivatives. All the routines needed for differentiating and integrating polynomials over various standard regions are contained within the StdRegions directory (and will be discussed in Chapter 5). A local region expansion, such as a basis defined on a quadrilateral element, QuadExp, is a linear combination of basis functions over its corresponding standard region as mapped by information contained within its spatial domain mapping. Local region class definitions are in the LocalRegions directory (and will be discussed in Chapter 7). Using the inheritance language of *C++*, we would say that a local region

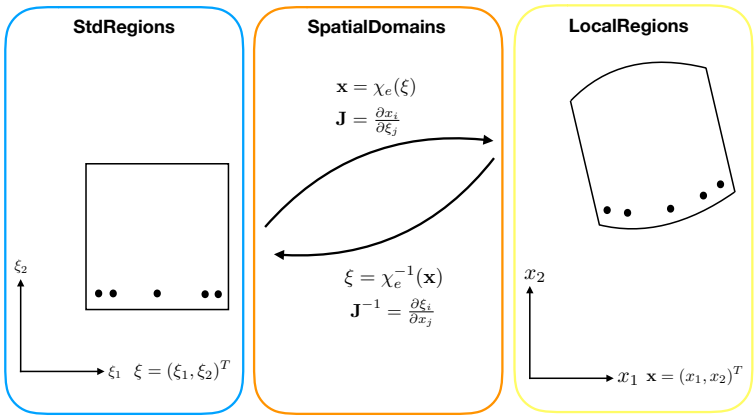


Figure 2.1: Diagram showing the LocalRegion (yellow) *is-a* StdRegion (blue) which *has-a* SpatialDomain (orange) in terms of coordinate systems and mapping functions. This diagram highlights how points (positions in space) are mapped between different (geometric) regions.

is-a standard region and *has-a* spatial domain object. The SpatialDomains directory contains information that expresses the mapping function $\chi_e(\cdot)$ from the standard region to a local region. SpatialDomains is explored in Chapter 6. In the case of our quad example, the SpatialDomain object held by a QuadExp would connect the local region to its StdRegion parent, and correspondingly would allow integration and differentiation in world space (i.e., the natural coordinates in which the local expansion lives). If $\mathcal{E}$ denotes our geometric region in world space and if $F : \mathcal{E} \to \mathbb{R}$ is built upon polynomials over its standard region, then we obtain $F(x_1, x_2) = \phi^e(\chi_e^{-1}(x_1, x_2))$. Note that even though ϕ^e is polynomial and χ_e is polynomial, the composition using the inverse of χ_e is not guaranteed to be polynomial: it is only guaranteed to be a smooth function.

Putting this in the context of MultiRegions, we arrive at Figure 2.2. The MultiRegions directory (which will be discussed in Chapter 9) contains various data structures that combine local regions. One can think of a multi-region as being a set of local region objects in which some collection of geometric and/or function properties are enforced. Conceptually, one can have a set of local region objects that have no relationship to each other in space. This is a set in the mathematical sense, but not really meaningful to us for solving approximation properties. Most often we want to think of sets of local regions as being collections of elements that are geometrically contiguous – that is, given any two elements in the set, we expect that there exists a path that allows us to trace from one element to the next. Assuming a geometrically continuous collection of elements, we can now ask if the functions built over those elements form a piece-wise discontinuous approximation of a function over our Multiregion or a piece-wise continuous C^0 approximation of a function over our Multiregion.

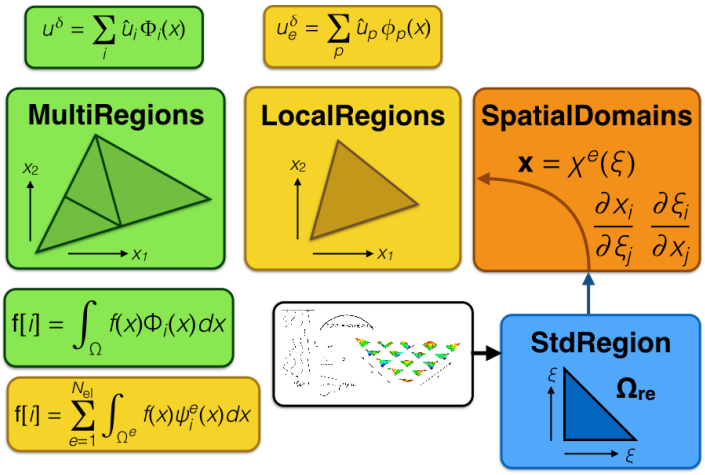


Figure 2.2: Diagram showing how a MultiRegion (green) contains a collection of LocalRegions (yellow), where a LocalRegion *is-a* StdRegion (blue) which *has-a* SpatialDomain (orange) in terms of coordinate systems and mapping functions. This diagram highlights the expansions of the solution formed over each geometric domain.

Top-Down Perspective

The top-down perspective on the library is best understood from Figure 2.3. From this perspective, we are interested in understanding *Nektar++* from the solvers various people have contributed.

2.3 Assumed Proficiencies

This developer's guide is designed for the experienced spectral/*hp* user who wishes to go beyond using various *Nektar++* solvers and possibly to add new features or capabilities at the library, solver or utilities levels. Since the focus of this document is *Nektar++*, we cannot recapitulate all relevant mathematical or computer science concepts upon which our framework is built. In this section we provide a listing of areas and/or topics of assumed knowledge, and we provide a non-exhaustive list of references to help the reader see the general areas of additional reading they may need to benefit fully from this manual.

We assume the reader has a familiarity and comfort-level with the following areas:

1. Finite Element Methods (FEM) [40, 54, 9] and more generally the mathematical ideas surrounding continuous Galerkin (cG) methods. This includes basic calculus of variation concepts, basic partial differential equation knowledge, and general forms of discretization and approximation.

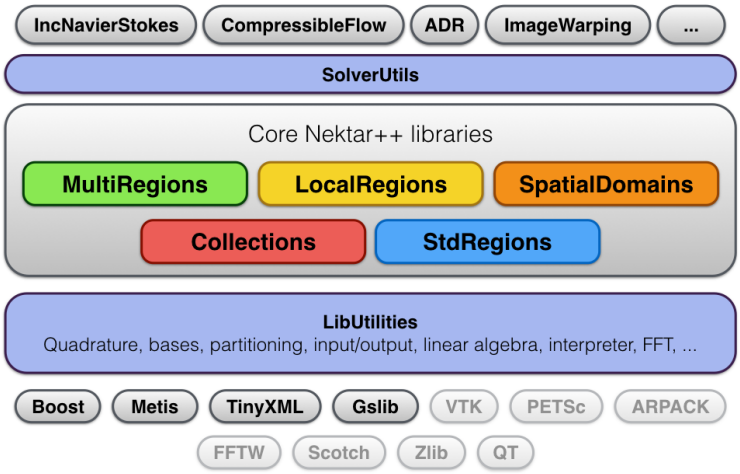


Figure 2.3: Diagram showing the top-down perspective on library components. Various solvers (along the top) can capitalize on generic SolverUtils and are built upon the core *Nektar++* libraries consisting of MultiRegions (green), Collections (red), LocalRegions (yellow), SpatialDomains (orange) and StdRegions (blue). The most generic mathematical and computer science features are contained within LibUtilities, which in part draws from various community resources such as Boost, Metis, etc.

2. Polynomial Methods [19, 32, 36], and in particular concepts surrounding polynomial spaces, basis functions and numerical differentiation and quadrature.

3. Spectral Element Methods (SEM) [25, 44].

4. Discontinuous Galerkin Methods [20, 39].

5. Scientific Computing [35, 43]. We assume a graduate-level proficiency in basic computational techniques such as dealing with numerical precision (accuracy and conditioning), root-finding algorithms, differentiation and integration, and basic optimization.

6. Linear Algebra [59, 24]. We assume a strong knowledge of linear algebra and a reasonable comfort level with the various numerical algorithms used for the solution of symmetric and non-symmetric linear systems.

2.4 Other Software Implementations and Frameworks

In the last ten years a collection of software frameworks has been put forward to try to bridge the gap between the mathematics of high-order methods and their implementation. A number of software packages already exist for fluid dynamics which implement high-order finite element methods, although these packages are typically targeted at a specific domain or provide limited high-order capabilities as an extension. A major challenge

many practitioners have with spectral/*hp* elements and high-order methods, in general, is the complexity (in terms of algorithmic design) they encounter. In this section, we give an incomplete but representative summary of several of these attempts to overcome this challenge.

The *Nektar flow solver* is the predecessor to *Nektar++* and implements the spectral/*hp* element method for solving the incompressible and compressible Navier-Stokes equations in both 2D and 3D. While it is widely used and the implementation is computationally efficient on small parallel problems, achieving scaling on large HPC clusters is challenging. Semtex [11] implements the 2D spectral element method coupled with a Fourier expansion in the third direction. The implementation is highly efficient, but can only be parallelised through Fourier-mode decomposition. Nek5000 [29] is a 3D spectral element code, based on hexahedral elements, which has been used for massively parallel simulations up to 300,000 cores. The Non-hydrostatic Unified Model of the Atmosphere (NUMA) [33] is a spectral element framework that employs continuous and discontinuous Galerkin strategies for solving a particular problem of interest, but in a way on which others could adopt and build. Hermes [60] implements hp-FEM for two-dimensional problems and has been used in a number of application areas. Limited high-order finite element capabilities are also included in a number of general purpose PDE frameworks including the DUNE project [23] and deal.II [10]. FEniCS [47] is a collaborative project for the development of scientific computing tools, with a particular focus on the automated solution of differential equations by finite element methods (FEM). Through the use of concepts such as meta-programming, FEniCS tries to keep the solving of PDEs with FEM, from the application programmers' perspective, as close to the mathematical expressions as possible without sacrificing computational efficiency. A number of codes also implement high-order finite element methods on GPGPUs including nudg++, which implments a nodal discontinuous Galerkin scheme [37], and PyFR [64], which supports a range of flux reconstruction techniques.

2.5 How to Use This Document

In the next chapter, we will introduce the reader to various computer science tools and ideas upon which we rely heavily in our development and deployment of *Nektar++*. The remainder of this developer's guide is then partitioned into three parts.

In Part I, we provide an overview of the various data structures and algorithms that live within the *library* subdirectory. We think of the library as containing all the basic building blocks of the *Nektar++* code, as discussed above. In this part of the developer's guide, we have dedicated a chapter to each subdirectory within the library. Each chapter contains three main sections. The first section of a chapter provides an overview of the mathematical concepts and terminology used within the chapter. This is not meant to be a detailed tutorial, but rather a reminder of basic concepts. The second section of a chapter provides a detailed description of the data structures introduced in that part of the library. The third section of a chapter is dedicated to explaining the algorithmic aspects of the library. Instead of going function by function or method by method, we

have decided to structure this section in the style of “frequently asked questions" (FAQ). Based upon our long experience with students, postdocs and collaborators, we have distilled down a collection of questions that we will use (from the pedagogical perspective) to allow us to drill down into key algorithmic aspects of the library.

In Part II, we provide an overview of the many solvers implemented on top of the *Nektar++* library. Each chapter is dedicated to a different solver, and correspondingly may take on a slightly different style of presentation to match the depth of mathematical, computer science and engineering knowledge to understand the basic data structures and algorithms represented there.

In Part III, we provide an overview of many of the utilities often used in our simulation pipeline – from things that aid the user in the preprocessing steps such as setting up parameter files and meshing to the postprocessing step of field conversion for visualization.

How you engage with this material is based upon your goals. If your goal is to:

- Introduce a new solver, then you would want to jump to Part II and see how other solvers have been built. Then, depending on what library functions are needed, you may need to step back into the library parts of this manual to understand how to use some of our basic library functionality.

Chapter 3

Preliminaries

3.1 Summary of Development Tools and Best Practices

3.1.1 General Resources

Mailing List

Please make sure that you sign up to the *Nektar++* mailing list. This can be done at the following URL:

https://mailman.ic.ac.uk/mailman/listinfo/nektar-users

While the name suggests it is for *users* of Nektar++, this list is also used to ask questions about *Nektar++* development.

Blog

The *Nektar++* blog (https://www.nektar.info/cat/community) provides a broad range of posts on topics such as compiling the code on specific machines, to discussions of *Nektar++* in specific application areas, to recently published papers which have made use of the code.

Contributing to this resource is a valuable way to support the project, as well as promoting your own work.

Annual Workshop

In 2015, we held the first *Nektar++* workshop, which was a great success and followed by a similar event in 2016. It is now an annual event and allows first-hand access to the core *Nektar++* development team as well as a range of other *Nektar++* users.

3.1.2 Version Control (git)

The *Nektar++* code is managed in a distributed version control system called *git*. To obtain the code directly from the repository and add new content to the repository requires the `git` command-line tool (or a suitable GUI equivalent). This is available for Linux, Mac OSX and Windows.

If you plan to work with the *Nektar++* community, you will need to have a reasonable understanding of the *git* software. While it is beyond the scope of this document to discuss how to use *git*, it is important for someone new to *git* to spend time understanding how the tool works. For this purpose, we highly recommend familiarizing yourself with it using any of the many online resources (such as `https://git-scm.com`).

Anonymous Access

Nektar++ may be cloned anonymously using the following command:

```
git clone http://gitlab.nektar.info/clone/nektar/nektar.git nektar++
```

Your local copy of the code can be updated to include the latest changes in the repository using the command:

```
git pull
```

The *Anonymous Access* approach will provide you with a complete copy of the code, but you will be unable to push changes or new developments back to the repository. For this, you should use *Collaborative Access*.

Collaborative Access

Once you are familiar with *git*, have introduced yourself to the development community (see the mailing list information above), and are ready to become a contributing developer, you will need to register an account on the *Nektar++* Gitlab website:

https://gitlab.nektar.info.

To use use authenticated access, using your new account, you must first upload the **public** part of your SSH key to your Gitlab profile. Generating and managing SSH keys is beyond the scope of this document. However, on Linux and OSX, one generally finds these keys in the `\$HOME/.ssh` directory.

To upload the key, visit: https://gitlab.nektar.info/profile, select *SSH keys* from the menu on the left and follow the instructions.

Registering for an account and uploading your SSH key need only be done once.

To clone *Nektar++*, use the *git* command:

```
git clone git@gitlab.nektar.info:nektar/nektar.git nektar++
```

Note the different URL to use authenticated access, in comparison to the anonymous access.

Managing and Contributing code

Code contribution then follows three basic steps:

1. Create an *issue* to describe the code updates you are making;
2. Branch the *Nektar++* master branch and make your changes on that branch; and
3. Submit a merge request on the *Nektar++* Gitlab website that your updates are ready to be merged into the master branch.

More details regarding the concepts mentioned above is found below. The information is also available in the `CONTRIBUTING.md` file in the root directory of the code, or online at: https://gitlab.nektar.info/nektar/nektar/blob/master/CONTRIBUTING.md

- Issues - The initial step for those who wish to add code to the master repository is to create an *issue* ticket that describes the defect, bug, proposed additions, changes, updates, etc. This is done on the Gitlab website at:

  https://gitlab.nektar.info/nektar/nektar/issues

  Please ensure you provide sufficient detail when creating the issue to cover all of the following (as required): Describe what is being requested, why it is important / necessary, an initial list of files that may be effected, any potential problems the change/addition may cause, and any other information that will help the development team understand the request and alert others to your work to avoid duplication of effort.

- Branches - The second step is to create a *git* branch in which to do the actual code development. In your local copy of the *Nektar++* repository, run the command:

  ```
  git branch <branch-name>
  ```

  replacing `<branch-name>` with a suitable name for your branch. The naming convention used for branches reflects the nature of the change and is composed of a prefix and descriptor, separated by a forward slash. Prefixes are one of:

  - `feature`: used for developments which constitute a new capability in the code;
  - `fix`: used for changes to fix a bug in the code;
  - `tidy`: used for changes which improve the quality or readability of the code but do not change its function;

– `ticket`: can be used to reference changes which address a specific issue.

Please choose concise descriptors in all-lowercase, using hyphens to separate words. An example `<branch-name>` might therefore be: `feature/low-energy-preconditioner`

Now make your new branch active by running the command:

```
git checkout <branch-name>
```

Confirm you are on the correct branch by running:

```
git status
```

which should print something similar to

```
On branch <branch-name>
```

At this point you are all set to make the required modifications to the code in your branch. As you modify your branch, you can use *git* to save and track your changes.

The following examples show how you can add a file to the list of local files that are being tracked, display differences between the current file and the original file, and commit the file.

```
git add library/LibUtilities/BasicUtils/my_new_file.cpp
git diff --cached
```

Note `--cached` is necessary because `my\_new\_file.cpp` was staged using the `git add` command above. Note, before you add (stage) the file, you can just use `git diff`.

To actually create the commit from the staged files, run:

```
git commit -m "Added␣X,␣Y,␣Z..."
```

This commits the file to your local repository. The first line of the log message should be a concise summary of the changes. You can use subsequent lines to provide more details if needed. Use `git log` to see the list of previous commits and their messages.

These changes are still local to your computer. To push them up to the main *Nektar++* repository, use the command

```
git push origin
```

3.1.3 Building (CMake)

Nektar++ uses the *CMake* tool to manage the build process for the three supported operating systems: Linux, Windows, and OSX. For detailed instructions on how to use *CMake* to build *Nektar++*, including a list of required software dependencies and *CMake* option flags, please refer to the *Nektar++* User Guide section 1.3.

3.1.4 Testing (CTest)

Before you are ready to have your code merged into the *Nektar++* trunk, you should make sure that it passes the built-in test suite - in addition to any new tests that you have added for your updates. To run the test suite, on the command line type:

```
ctest [-j#]
```

The `-j#` optional argument will run `#` tests in parallel taking advantage of multiple cores on your machine. It is highly recommended that your use all available cores to minimize the amount of time spent waiting for the tests to complete. There are currently several hundred built-in unit tests for *Nektar++*.

For more information on testing, see "Software Testing Approaches" below.

3.1.5 Merge Requests (Gitlab)

The final step in contributing your code to the Nektar master repository is to submit a merge request to the development team using the *Nektar++* gitlab website:

https://gitlab.nektar.info/nektar/nektar/merge_requests

Submitting a merge request will automatically trigger the continuous integration (CI) system, which will build and test your code on a range of platforms. You can monitor the progress of these tests from the merge request page. Selecting individual workers will take you to the buildbot website from which you can examine any failures which have occurred.

3.2 Documentation and Tutorials

Documentation for Nektar++ is provided in a number of forms:

- User Guide (LaTeX, compiled to pdf or html)
- Source code documentation (Doxygen compiled to html)

3.2.1 Dependencies

To build the User Guide and Developer's Guide, the following dependencies are required:

- texlive-base
- texlive-latex-extra
- texlive-science
- texlive-fonts-recommended
- texlive-pstricks
- imagemagick

3.2.2 Compiling the User Guide

To compile the User Guide:

1. Configure the Nektar++ build tree as normal.
2. Run

   ```
   make user-guide-pdf
   ```

   to make the PDF version, or run

   ```
   make user-guide-html
   ```

   to make the HTML version.

3.2.3 Developers Guide

To compile the Developer's Guide:

1. Configure the Nektar++ build tree as normal.
2. Run

   ```
   make developer-guide-pdf
   ```

   to make the PDF version, or run

   ```
   make developer-guide-html
   ```

   to make the HTML version.

3.2.4 Compiling the code documentation

To build the Doxygen documentation, the following dependencies are required:

- doxygen
- graphviz

To compile the code documentation enable the `NEKTAR_BUILD_DOC` option in the `ccmake` configuration tool.

You can then compile the HTML code documentation using:

```
make doc
```

3.3 Compiling Tutorials

If you are using a clone of the *Nektar++* git repository, you can also download the source for the *Nektar++* tutorials which is available as a *git submodule*.

1. From a *Nektar++* working directory (e.g. `$NEKPP`):

   ```
   git submodule init
   git submodule update --remote
   ```

2. From your build directory (e.g. `$NEKPP/build`), re-run `cmake` to update the build system to include the tutorials

   ```
   cmake ../
   ```

3. Compile each required tutorial, for example

   ```
   make flow-stability-channel
   ```

3.4 Core Nektar++ Programming Concepts

This section highlights some of the programming features that are used extensively within *Nektar++*. While much of the code consists of standard C++ practices, in some of the core infrastructure there are several practices that may be only familiar to programmers who have developed code using more advanced C++ features. Below we give a short summary of these entities in order to provide a starting point when working with these features. We begin with more well known features and end with some advanced techniques. Note, it is not the purpose of the following sections to cover in

detail each of these important concepts, but instead to give a brief overview of them such that the developer may look to other, more in-depth, sources if they require further guidance.

3.4.1 Namespaces

Many C++ software projects place their code in a namespace so as to avoid conflicts with other code when included in larger applications. It is important to note that *Nektar++* uses a hierarchy of namespaces for most of the defined data structures. The top level namespace is always "Nektar", with the second level usually corresponding to the name of the library to which the code belongs. For example:

```
namespace Nektar
{
namespace StdRegions
{
    ...
}
}
```

With this in mind, when you see something like `Nektar::SpatialDomains::...`, you can usually assume that the second item (in this case `SpatialDomains`) is a namespace, and not a class.

Note: To make better use of the 80 character width, generally enforced across the *Nektar++* source code, we choose not to indent the contents of `namespace` blocks.

3.4.2 C++ Standard Template Library (STL)

Nektar++ uses of the C++ STL extensively. This consists of common data structures and algorithms, such as map and vector, as well as many of the extensions once found in the Boost library that have become part of the C++ standard and are now used directly.

One of the most important of these features is the use of Shared Pointers (`std::shared_ptr`). Most developers are somewhat familiar with "smart pointers" (pointers used to track memory allocation and to automatically deallocate the memory when it is no longer being used) for data blocks that are shared by multiple objects. These smart pointers are used extensively in *Nektar++* and one should be familiar with the `dynamic_pointer_cast` function and the concept of the `weak_ptr`. Dynamic casting allows for safely converting one type of variable into its base type (or vice versa). For example:

```
std::shared_ptr<FilterCheckpoint> sptr =
        std::dynamic_pointer_cast<FilterCheckpoint>( m_filters[k] );
if( sptr != nullptr )
{
   // Cast succeeded!
}
```

The advantage of using the dynamic cast, in comparison to the C style cast, is that you can check the return value at run time to verify that the casting was valid. A `weak_ptr` is a pointer to shared data with the explicit contract that the weak pointer does not own the data (and thus will not be responsible for deallocating it). Weak pointers are used mostly for short-term access to shared data.

Another modern code utility used by *Nektar++* to support shared pointers can be seen in *Nektar++* classes which inherit from `std::enable_shared_from_this`. This allows a class member function to return a shared pointer to itself. Specifically, it makes available the function `shared_from_this()` which returns a shard pointer to the object in the given context.

While C++ shared pointers are a powerful resource, there are a number of intricacies that must be understood and followed when creating classes and using objects that will be managed by them. For those not familiar with the C++11 (or previously Boost) implementation, it is highly recommended that you study them in more detail then presented here.

3.4.3 typedefs

Like most other large codes, *Nektar++* uses `typedefs` to create short names for new variable types. You will see examples of this throughout the code and taking a few minutes to look at the definitions will help make it easier to follow the code. In the following example, we create (and explicitly name) the type `ExpansionSharedPtr` to make the code that uses this type easier to follow. This is particularly true of nested STL data structures where repeated template declarations would make the code harder to follow. A couple of examples are shown below:

```
1 typedef std::shared_ptr<Expansion> ExpansionSharedPtr;
2 typedef std::shared_ptr<std::vector<std::pair<GeometrySharedPtr, int>>>
3 GeometryLinkSharedPtr;
```

If you are not familiar with the use of typedefs, you should take time to read about them (there are many short summaries available on the web).

3.4.4 Forward Declarations

There are two ways that an existing class type can be specified when declaring a new class in a header file. The existing class can either be declared in name only, or declared in its entirety, before being used. In the latter case, one typically includes the header file declaring the full class. If the new class declaration only references the existing class in the form of a pointer or reference then the entire class declaration is not needed and the compiler only needs an assurance that the class exists. For this case, we can use a *forward declaration* which tells the compiler the name of the existing class. However, if functions of the existing class are called (within the new header file) or the class is used by value, then the full declaration is needed.

Forward declaring a class is achieved as shown in the following example:

```
1 class LinearSystem;
```

This statement tells the compiler the class `LinearSystem` exists and, as long as we only make reference to it as a pointer (`LinearSystem* l`) or by reference (`const LinearSystem & l`), then the compiler does not require any further information.

An advantage to using forward declarations where possible is that the header file does not need to `#include` the entire existing class and any header files referenced within. This allows for a cleaner header files and faster compilation as the compiler can process (often significantly) fewer lines of code.

Note: The full class declaration is most likely needed in the new class implementation file (.cpp) as reference to the existing class's members will presumably be made.

3.4.5 Templated Classes and Specialization

Most C++ developers are familiar with basic class templating. However, many have not needed to use explicit template specialization. This is the process of implementing customised behaviour for one or more of the specific instances of a template when the compiler will not be able to instantiate a generic version for the class, or when different code is needed based on different versions of the class. For example:

```
1 template<typename Dim, typename DataType>
2 class Array;
3
4 template<typename DataType>
5 class Array<OneD, const DataType> {
6   // Explicit coding of class methods and variables specific to
7   // this version of Array are found here.
8 };
```

In the above example, on the first line the generic templated `Array` class is declared. There are two template parameters: the dimension and the element type. The second line shows an explicit template specialisation of the `Array` class for a one-dimensional (version of) `Array`. When explicitly specialising a class, the programmer will write code that is specific to the datatype used in specifying the class. This includes explicitly writing code for one, some, or all of the methods of the class.

It is important to understand template specialization when dealing with the *Nektar++* core libraries so that the developer can determine which (specialized version of the) class is being used, and to know that when updating classes with varied specializations, that it may be required to update code in several places (ie, for each of the specializations).

3.4.6 Multiple Inheritance and the `virtual` Keyword

When diving into many *Nektar++* classes, you will see the use of multiple inheritance (where a class inherits from more than one parent class). When the parent class does not inherit from other classes, then the inheritance is straightforward and should not cause any confusion. However, when a class has grandparents, many times that grandparent class is the same class but is inherited through multiple parents. To account for this, class inheritance should use the `virtual` keyword. This specifies that if a class has multiple grandparents (that happen to be the same class), that only one copy of the grandparent class members should actually be instantiated. For example:

```
class Expansion2D : virtual public Expansion,
                    virtual public StdRegions::StdExpansion2D
{...}
```

3.4.7 Virtual Functions and Inheritance

Within *Nektar++*, classes that inherit from a parent class and override one of the parent class methods, use the concept of virtual functions. The function is prefixed with a `v_`, such as `v_Function()`, as a naming convention. This is a visual reminder that the function overrides a parent class function. For example:

```
NekDouble TriExp::v_Integral(const Array<OneD, const NekDouble> &inarray)}
```

3.4.8 Const keyword

While the `const` keyword is known to most C++ developers, it is used (as it should be) liberally in *Nektar++* for functions, function parameters, returning pointers to class data, and variable constants within functions. It is easy to neglect using *const* to mark all cases where a variable should be considered constant. However, its use can substantially reduce accidental errors and allow for accelerated debugging. The `const` qualifier should be used wherever a variable does not change including 1) parameters passed to functions, 2) variables in functions (or classes) that do not change value during their lifetime, 3) on the return type of functions that return pointers to data that should not be changed, and 4) on methods that do not change data within the class. The compiler will then produce an error if we (accidentally) attempt to make a change which violates a `const`.

3.4.9 Function pointers and bind

Function pointers (`std::function`) are similar to pointers to data, except that they point to functions - and thus allow a function to be invoked indirectly (in other words, without explicitly writing the function call (name) directly in code). This technique is used by *Nektar++* in a number of places, with `NekManager` being a prime example. The `NekManager` class is used to create objects of a specific type during the execution of the program. When a `NekManager` is created (constructed), it is provided with a pointer to a function that will (later) be called to generate the objects to be managed when required. While the *creation* function that is provided to the `NekManager` takes a number of parameters,

in many cases some of the values to those parameters will be fixed. To handle this situation, *Nektar++* uses the `std::bind( f )` function, which creates a new function based on supplied original function `f`, but specifies that one or more parameters of `f` are fixed at the time that `f` is created and only those bound parameter values will be used when `f` is later invoked.

3.4.10 Memory Pools and 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

```
1 struct TenD {
2     static unsigned int Value = 10;
3 };
```

- `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:

```
1 class Sample {
2 public:
3     Array<OneD, const double>& getData() const { return m_data; }
4     void getData(Array<OneD, const double>& out) const { out = m_data; }
5
6 private:
7     Array<OneD, double> m_data;
8 };
```

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.

3.5 Design Patterns

3.5.1 Template pattern

The template pattern is used frequently within *Nektar++* to provide a common interface to a range of related classes. The base class declares common functionality or algorithms, in the form of public functions, deferring to protected virtual functions where a specific implementation is required for each derived class. This ensures code external to the class hierarchy sees a common interface.

In the top level parent class you will find the interface functions, such as `Function()`, declared as public members. In some cases, these may implement a generic algorithm common to all classes. In the limit that the function is entirely dependent on the derived class, it may call through to a virtual counterpart, usually named `v_Function()`. These functions are usually protected to allow them to be called directly by other classes in the inheritance hierarchy, without exposing them to external classes.

As an example of this, let us consider a triangle element (`TriExp`). The `TriExp` class (eventually) inherits from the `StdExpansion` class. The `StdExpansion` defines the `Integral()` function which is used to provide integration over an element. However, in this case, the implementation is shape-specific. Therefore `StdExpansion::Integral()` calls the (in this case) `TriExp::v_Integral()` function. We should also note that while `TriExp::v_Integral()` does setup work, it then makes use of its parent's `StdExpansion2D::v_Integral()` function to calculate the final value. This is only possible if the `v_Integral()` function was declared as protected.

3.5.2 Abstract Factory Pattern

Nektar++ makes extensive use of the *Factory Pattern*. Factories are used to create (allocate) instances of classes using class-specific creator functions. More specifically, a factory will create a new object of some sub-class type but return a base class pointer to the new object. In general, there are two ways that a factory knows what specific type of object to generate: 1) The Factory's build function (`CreateInstance()`) is passed a key that details what to build; or 2) The factory may have some intrinsic knowledge detailing what objects to create. The first case is almost exclusively used throughout *Nektar++*. The factory pattern 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.

Implementing the factory pattern

The `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 tasks involve adding mostly standard boilerplate code.

It is assumed that we are writing a code which implements a particular *concept* or *capability* within the code, for which there are (potentially) multiple implementations. The reasons for multiple implementations may be low level, such as alternative algorithms for solving a linear system, or high level, such as selecting from a range of PDEs to solve. The `NekFactory` can be used in both cases and applied in exactly the same way.

Creating a concept (base class)

A base class must be defined which prescribes an implementation-independent interface. In Nektar++, the template method pattern (see Section 3.5.1 above) is used, requiring public interface functions to be defined which call through to protected virtual implementation methods. This is because the factory returns the newly created object via a base-class pointer and the objects will almost always be used via this base class pointer. Without a public interface in the base class, much of the benefits and generalisation of code offered by the factory pattern would be lost. The virtual functions will be overridden in the specific implementation classes. In the base class these virtual methods should normally be defined as pure virtual, since there is typically no implementation and we will never be explicitly instantiating this base class.

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

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

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

```
1 class MyConcept
```

```
2
3 // Datatype for the MyConcept factory
4 typedef LibUtilities::NekFactory< std::string, MyConcept,
5             ParamType1,
6             ParamType2 > MyConceptFactory;
7 MyConceptFactory& GetMyConceptFactory();
```

The template parameters to the `NekFactory` 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 datatype (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. Since we have not yet declared the base class type `MyConcept`, we have forward-declared it above the `NekFactory` type definition.

The normal definition of our base class then follows:

```
1 class MyConcept
2 {
3     public:
4         MyConcept(ParamType1 p1, ParamType2 p2);
5         ...
6 };
```

We must also define a shared pointer for our base class, which should be declared outside the base class declaration.

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

Creating a specific implementation (derived class)

A new class, derived from the base class above, is defined for each specific implementations 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

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

We then define the derived class as one would normally:

```
1 class MyConceptImpl1 : public MyConcept
2 {
3 ...
4 };
```

In order for the factory to work, it must know two things:

- that `MyConceptImpl1` exists; and

- how to create an instance of it.

To allow the factory to create instances of our class we define a *creator* function in our class, which may have arbitrary name, but is usually called `create` out of convention:

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

The example above the `create` function simply creates an instance of `MyConceptImpl1` using the *Nektar++* memory manager and the supplied parameters. It must be a `static` function because we are not operating on any existing instance and it should return a shared pointer to a base class object (rather than a `MyConceptImpl1` shared pointer), since the point of the factory is that the calling code does not know about specific implementations. An advantage of having each class providing a creator function is that it allows for *two-stage initialisation* – for example, initialising base-class variables based on the derived type.

The final task is to register each of our implementations with the factory. This is done using the `RegisterCreatorFunction` member function of the `NekFactory`. 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)! One solution is to use the function to initialise a static variable: it will force the function to be executed prior to the start of the `main()` routine, and can be located within the class it is registering, satisfying our code decoupling requirements.

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

```
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`

```
string MyConceptImpl1::className
        = GetMyConceptFactory().RegisterCreatorFunction(
                                "Impl1",
                                MyConceptImpl1::create,
                                "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 the static function which

should be used by the factory to instantiate our class. The third parameter provides a description which can be printed when listing the available `MyConcept` implementations. A specific implementation can be registered with the factory multiple times if there are multiple keys which should instantiate an object of this class.

Instantiating classes

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

```
1 #include <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 recompilation and the executable relinking.

We create an instance by retrieving the `MyConceptFactory` and calling the `CreateInstance` member function of the factory, for example,

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

Note that the instance of the specific implementation 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.

3.6 Software Testing Approaches

3.6.1 Unit Tests

Unit testing, sometimes called "module testing" or "element testing", is a software testing method by which individual "units" of source code are tested to determine whether they are fit for use [42]. Unit tests are added to *Nektar++* through the CMake system, and implemented using the Boost test framework. As an example, the set of linear algebra unit tests is listed in this file:

`.../library/UnitTests/LibUtilities/LinearAlgebra/CMakeLists.txt`

and the actual tests are implemented in this file:

`.../library/UnitTests/LibUtilities/LinearAlgebra/TestBandedMatrixOperations.cpp`

To register a new test, you use `BOOST_AUTO_TEST_CASE( TestName )`, implement the unit test, and test the result using `BOOST_CHECK_CLOSE(...)`, `BOOST_CHECK_EQUAL(...)`, etc. Unit tests are invaluable in maintaining the integrity of the code base and for localizing, finding,

and debugging errors entered into the code. It is important to remember a unit test should test very specific functionality of the code - in the best case, a single function should be tested per unit test.

While it is beyond the scope of this document to go into more detail on writing unit tests, a good summary of the Boost test system can be found here:

http://www.boost.org/doc/libs/1_63_0/libs/test/doc/html/.

3.6.2 Integration, System and Regression Tests

Integration testing involves testing ecosystems of components and their interoperability. System testing tests complete applications and regression testing focuses on ensuring previously fixed bugs do not resurface. In *Nektar++* all of these are often colloquially referred to as *regression testing*. It is not *white-box* in that it does not examine how the code arrives at a particular answer, but rather in a *black-box* fashion tests to see if code when operating on certain data yields the predicted response [42].

3.6.3 Continuous Integration

Nektar++ uses the *buildbot* continuous integration to perform testing of the code across multiple operating systems. Builds are automatically instigated when merge requests are opened and subsequently when the associated branches receive additional commits.

For more information, go to:

http://buildbot.nektar.info

Part I

Building-Blocks of Our Framework (Inside the Library)

Chapter 4

Inside the Library: LibUtilities

In this chapter, we walk the reader through the different components of the LibUtilities Directory. We have ordered them in alphabetical order by directory name, not by level of importance or relevance to the code. Since all of these items are considered foundational to Nektar++, they should all be considered equally important and relevant. Along the same lines – since all of these areas of the code represent the deepest members of the code hierarchy, these items should rarely be modified.

4.1 BasicConst

This directory contains two important files for all of *Nektar++*: NektarUnivConsts.hpp and NektarUnivTypeDefs.hpp.

The file NektarUnivConsts.hpp contains various default constants used within *Nektar++* as seen here:

```
1         static const NekDouble kNekMinResidInit = 1e16;
2         static const NekDouble kVertexTheSameDouble  = 1.0e-8;
3         static const NekDouble kGeomFactorsTol = 1.0e-8;
4         static const NekDouble kNekZeroTol = 1.0e-12;
5         static const NekDouble kGeomRightAngleTol = 1e-14;
6         static const NekDouble kNekSqrtTol = 1.0e-16;
7         static const NekDouble kNekIterativeTol = 1e-09;
8         static const NekDouble kNekSparseNonZeroTol = 1e-16;
9
10        // Tolerances for mesh generation and CAD handling
11        static const NekDouble GeomTol = 1E-2;
12        static const NekDouble CoinTol = 1E-6;
13    }
```

The file NektarUnivTypeDefs.hpp contains the low level typedefs such as: NekDouble, NekInt, OneD, TwoD, ThreeD, FourD, and enumerations such as Direction (xDir, yDir and zDir) and OutputFormat.

4.2 BasicUtils

This directory contains some of the lowest level basic computer science routines within *Nektar++*. The directory currently contains the following:

ArrayEqualityComparison.cpp	ArrayPolicies.hpp	CompressData.cpp
CompressData.h	ConsistentObjectAccess.hpp	CsvIO.cpp
CsvIO.h	Equation.cpp	Equation.h
ErrorUtil.hpp	FieldIO.cpp	FieldIO.h
FieldIOHdf5.cpp	FieldIOHdf5.h	FieldIOXml.cpp
FieldIOXml.h	FileSystem.cpp	FileSystem.h
H5.cpp	H5.h	HashUtils.hpp
Metis.hpp	**NekFactory.hpp**	**NekManager.hpp**
OperatorGenerators.hpp	ParseUtils.cpp	ParseUtils.h
Progressbar.hpp	PtsField.cpp	PtsField.h
PtsIO.cpp	PtsIO.h	RawType.hpp
Scotch.hpp	SessionReader.cpp	SessionReader.h
ShapeType.hpp	SharedArray.hpp	Tau.hpp
Thread.cpp	Thread.h	ThreadBoost.cpp
ThreadBoost.h	Timer.cpp	Timer.h
VDmath.hpp	VDmathArray.hpp	Vmath.cpp
Vmath.hpp	VmathArray.hpp	VtkUtil.hpp

We have used **bold** to denote (as examples) routines at our used throughout *Nektar++*. They are in this sense "fundamental". Note that this list includes input/output routines (e.g. CompressData, FieldIo, H5), partitioning (e.g. Metis and Scotch) and Threading (e.g. Thread and ThreadBoost).

4.3 Communication

This directory contains files related to our distributed memory communication model. In particular, this directory contains files that help encapsulate MPI (Message Passing Interface) routines, as well as the Gather-Scatter (GS) and Xxt routines of Paul Fisher (Argonne National Lab and UIUC).

Comm.cpp	CommMpi.cpp	GsLib.hpp
Comm.h	CommMpi.h	Transposition.cpp
CommDataType.cpp	CommSerial.cpp	Transposition.h
CommDataType.h	CommSerial.h	Xxt.hpp

4.4 FFT

This directory contains files related to the use of Fast Fourier Transforms within *Nektar++*. The two groups of files are as follows:

- NekFFTW.h/NekFFTW.cpp : Wrapper around FFTW library; and
- NektarFFT.h/NektarFFT.cpp : Fast Fourier Transform base class in *Nektar++*.

4.5 Foundations

The two basic building blocks of all that is done in Nektar++ are the concepts of Points and of a Basis. The Point objects denote positions in space, either on compact domains (normally $[-1, 1]^d$ where d is the dimension in a reference domain mapped to world-space) or periodic domains such as $(0, 2\pi]$ (i.e. in the case of points used in Fourier expansions). The Basis objects denote functions (e.g. polynomials) evaluated at a given set of points.

We rely heavily on the concept of "managers", and this concept has its start right here in Foundations. When we started our development of *Nektar++*, we realized that a pure encapsulation strategy for Expansions would consist of every expansion holding a pointer to the basis (of some sort) that was used to specify it, and then correspondingly the basis pointing to a collection of points at which it was evaluated. From the computer science perspective, this is all very reasonable except for the fact that there are many cases in spectral/*hp* element expansions in which multiple elements use the same reference space basis functions, and correspondingly those basis functions are often evaluated at the same set of points (quadrature points). This led us to the conclusion that if we were going to have a code that could run over large number of elements (at the time tens of millions) without excessive memory usage, we would have to try to avoid as much of this duplication as possible. This led us to introduce to *Nektar++* two important computer science tools: *smart pointers* and *managers* (i.e. a factory pattern).

Smart pointers were originally introduced as an add-on from the *Boost* library and later directly incorporated into *Nektar++* when they were natively handled by C++11. With the tradition view of pointers, one has a variable that holds an address that points to a place in memory. Typically, in type-safe programming, the way this memory is to be interpreted is denoted by the type of pointer: a double pointer (i.e. double*) is a pointer to memory that should be interpreted as a double. The downside of this traditional view occurs when multiple pointers all point to the same place in memory. The question then arises: which pointer (or more properly which thread of control containing a pointer) is responsible for deleting the memory when it is no longer needed? Again, in the traditional view that each pointer points to unique memory and hence when the memory is not longer needed the memory can be released no longer is viable when multiple pointers all point to the same piece of memory. This probably was solved with the invention of the smart pointer, an object that is wrapped up to look like a pointer, and yet contains a reference counter within it. When a smart pointer is initiated and memory is originally

assigned to it, the reference counter is incremented. As more and more references to the piece of memory are created, the increment counter continues to increase. As pointers go out of scope and are longer valid, the increment is decremented. Only when all the references to a particular piece of memory are removed can a *free* (i.e. delete) operation be accomplished. The introduction of smart pointers into C++ and into *Nektar++* were particularly advantageous to us as they allowed us to have lots of different elements, expansions, etc. point to the same fundamental data structure in memory (for instance, a Basis object evaluated at a particular set of points) without having to worry about improper deallocation or memory leakage due to pointer mishandling.

The second important feature we exploited within the redesign of *Nektar++* was the idea of *managers* (i.e. a factory pattern). You will see this use of the term manager throughout *Nektar++*. For us, a manager has the following characteristics: it expects as input a *key*, which gives the manager sufficient information to know the object for which it is receiving a request. If the manager already has one of those objects in storage, it passes back a (smart) pointer that that object. The manager itself continues to hold on to a pointer for the object (in case other requests are made of it for that object). If the manager does not have a copy of the requested object, then it has the ability to generate the object "on the fly" and store it for future use. This last component of the manager is the factory feature that we mentioned earlier – a manager has registered with it a collection of methods that know how to generate the various objects that might be requested of it.

Based upon these two concepts: two fundamental managers that exist at this level of the library are the PointsManager and the BasisManager. These are contained within ManagerAccess.h/cpp, and are defined as singletons and defined globally so that they can be accessed throughout *Nektar++*. Within these files you will find the RegisterCreator method calls that link particular Create methods with various keys. Only in rare cases will these every need to be modified, and only infrequently possibly added to (if additional Points or Basis information is given).

4.5.1 Points

One of the two primary data structures in this directory is Points. A points object consists of a PointsKey and then extra information needed to facilitate various operations at points. The basic layout of the data structure is shown in Figure 4.1.

The PointsKey object has three basis data members: the number of points, an enum specifying the point distribution, and a optional scaling factor. In almost all of *Nektar++*, only the first two are needed to uniquely specify a collection of points. The latter variable was added as part of the extensions of *Nektar++* to accommodate NekMesh. As stated above, if you pass a PointsKey to a PointsManager you will receive back a pointer to a Points object with its various data members populated, among them a PointsKey identifier for the object. As can be seen by the diagram, the Points object has a collection (array) of three pointers to point positions. These three array pointers allow Points objects to point to 1D, 2D or 3D point positions. Corresponding to the dimension and

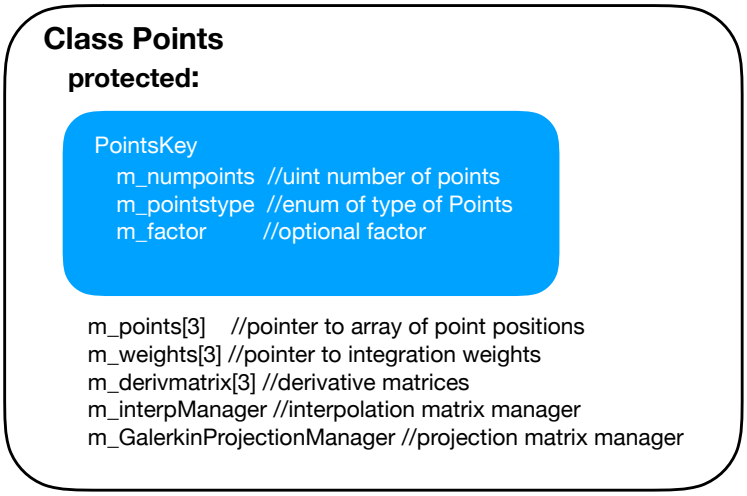


Figure 4.1: The basic Points class data object. It consists of a PointsKey (used by a PointsManager) and various other data members needed for point operations.

associated with each point position is a weight associated with integration. This was done to facilitate both non-tensor product and tensor product constructions. To demonstrate non-tensor product constructions, consider if you were to be dealing with a 2D point set that is natively 2D (that is, not based upon tensor product construction such as the Electrostatic Points), then the number of points in the PointsKey would be the size of each array associated with `m_points`, and only one of the `m_weight` arrays (associated with `m_weight[0])` would be valid such at:

$$\int_{\Omega_{2D}} f(x_0, x_1)\, dx_0\, dx_1 \approx \sum_{i=0}^{numpoints-1} \omega_i f(z_{0,i}, z_{1,i})$$

where $z_{0,i}$ and $z_{1,i}$ denote the points associated with `m_points[0][i]` and `m_points[1][i]` respectively. The weights in this case are calculated by routines found in NodalUtils via forming a Vandermonde system, solving for Lagrange basis functions that pass through a particular set of points, and then integrating those basis functions to obtain weights. This operation is done sufficiently often that we will try to spell it out after the Fekete Point paragraph below.

If one is dealing with a tensor product constructed 2D nodal point set, one would have:

$$\int_{\Omega_{2D}} f(x_0, x_1)\, dx_0\, dx_1 \approx \sum_{i=0}^{numpoints-1} \left[\sum_{j=0}^{numpoints-1} \omega_i \omega_j f(z_{0,i}, z_{1,j}) \right]$$

where $z_{0,i}$ and $z_{1,i}$ denote the points associated with `m_points[0][i]` and `m_points[1][i]`

respectively and ω_i and ω_j are the weights associated with the two coordinate directions respectively. The tensor product construction above was implemented in *Nektar++* assuming we would create derived nodal point sets for quadrilaterals and hexahedron; however, throughout much of *Nektar++*, we use the tensor product forms explicitly by specifying 1D point sets in the different directions as needed. This particular point will become more apparent when you get to reading the Chapter on StdRegions (Chapter 5). You will encounter, for instance, that a Standard Quadrilateral Expansion (StdQuadExp) requires two 1D Point objects denoting the two coordinates for integration. In this case, the number of points and number of weights match up per direction.

Electrostatic Points: The Electrostatic (Nodal) Points on the triangle and tetrahedron are based upon the work of Hesthaven [38]. They are an attempt to find the minimizer of a potential energy function based upon an electrostatic point source analogy, and provide points that correspondingly have low Lebesgue constant. The point positions as given in the paper are stored in the files NodalTriElecData.h and NodalTetElecData.h. Hesthaven uses a condensed format for the nodal positions which rely on the symmetries of the points (if you count the points in the file, you will see that fewer points are given than are needed to span the polynomial space. For example, for degree one, a single point it given. This point along with its three-fold symmetries represent all three points needed to support the linear space). The routine `CalculatePoints()` expands the condensed format of the points to the full set based upon the one, three (a and b denoting two types of three-symmetries) and six symmetries. Note that the ordering of the nodes expanded from the file does not respect any particular geometric ordering, so we reorder the points using `NodalPointReorder2d()`, which reorganizes the points in vertex points, edge points, and then interior points (i.e. a geometric decomposition that aligns more naturally with the way we organize nodes/modes within *Nektar++*). In Figure 4.2 we show an example of this reordering done for the points needed for expressing a total degree three expansion over a triangle.

Fekete Points: The Fekete (Nodal) Points on a triangle are based upon the work of Taylor and Wingate [56, 57], and are an alternative nodal point distribution on both triangles and tetrahedra. As an alternative strategy to the one mentioned above, Taylor and Wingate explicitly attempt to find a point distribution that minimizes the Lebesgue constant. For their particular point set, the minimization if over the Lebesgue function itself (not the electrostatic potential function, which can be viewed as a proxy for this function). The point positions as given in the paper are stored in the file NodalTriFeketeData.h and NodalTetFeketeData.h. Taylor and Wingate use a condensed format for the nodal positions similar to that used by Hesthaven.

Building Lagrange Interpolating Polynomials: One common operation done in both the case of the Electrostatic Points and the Fekete Points is to build the Lagrange interpolating polynomial associated with a point set. This is needed, for instance, for computing the integration weights associated with a particular collection of points (as mentioned above). This operation is done sufficiently often that we will try to spell it out

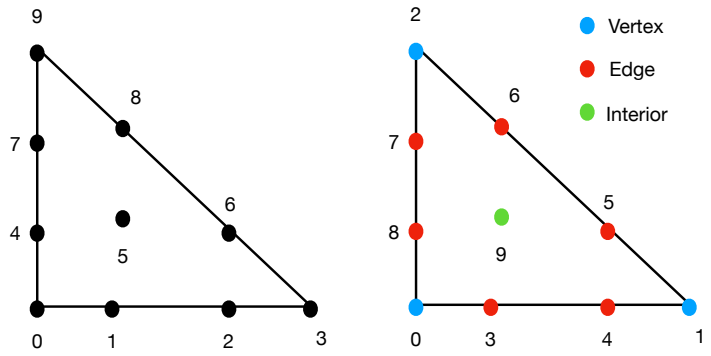


Figure 4.2: Example of how nodes are reordered to meet a geometric ordering. On the left, we show the nodal positions that would be used for building a total degree three polynomial space over a triangle ordered in canonical form. On the right, we show the reordering of those nodes to follow a vertex, edge and interior ordering. Note that in the case of a tetrahedron, the ordering would be vertex, edge, face and then interior.

here. Forming the Lagrange basis functions for a particular set of points, in particular in 2D and 3D, is non-trivial. The most common way of arriving at a set of actionable basis functions is to use a known basis (for which we are able to explicitly write down its form) and find the linear combination of those basis functions that yield the Lagrange basis. For example, let us assume we are dealing with the set of Electrostatic Points on a triangle. The number of points, `numpoints`, will specify the number of "modes" (coefficients) we expect to find. The total degree of the space is given by $N = \frac{(M+1)(M+2)}{2}$ where N is the degree of the polynomials and M is the number of points. As an example: note that when $M = 3$, we get $N = 1$ – that is, with three points we can support a polynomial of total degree one over a triangle. Let ϕ_i denote a known non-interpolating basis function. Although not done in practice, imagine that this is, for instance, a monomial. In 1D, this would equate to $\phi_i(x) = x^i$, and in multiple dimensions this would equate to $\phi_i(x, y) = x^{i_1} y^{i_2}$ with some index mapping function $i = \sigma(i_1, i_2)$ that gives us an index ordering. The total number of basis functions to span our degree N space is given by M. We first form the Vandermonde matrix $\mathbf{V}$ as follows:

$$\mathbf{V} = \begin{bmatrix} \phi_0(x_{0,0}, x_{1,0}) & \phi_1(x_{0,0}, x_{1,0}) & \dots & \phi_N(x_{0,0}, x_{1,0}) \\ \phi_0(x_{0,1}, x_{1,1}) & \phi_1(x_{0,1}, x_{1,1}) & \dots & \phi_N(x_{0,1}, x_{1,1}) \\ & \vdots & & \\ \phi_0(x_{0,N}, x_{1,N}) & \phi_1(x_{0,N}, x_{1,N}) & \dots & \phi_N(x_{0,N}, x_{1,N}) \end{bmatrix}$$

where each row i denotes the evaluation of our N basis functions at a given point $(x_{0,i}, x_{1,i})$. Since we have selected the number of basis functions to be what can be uniquely resolved by our N points, this matrix is square and invertible. The conditioning of the matrix is based upon our basis choice; hence we know that monomials are not a

good choice beyond approximately cubics so in general we use our internal modified basis for this system. With $\mathbf{V}$ now available, we can form the linear system:

$$\mathbf{Vc} = \mathbf{b}$$

where $\mathbf{c}$ is the vector of coefficients and where $\mathbf{b} = b_i$ is a binary vector where the i^{th} entry is set to one when finding the coefficients for the i^{th} Lagrange basis function associated with a particular set of points. For each Lagrange basis function definition we seek to find, we update the right-hand-side vector and solve the linear system for the set of coefficients that denote the combination of our known basis that yields a Lagrange basis function. Note that since this operation (of "inverting" this linear system) must be done for each Lagrange basis function, we can optimize our operations by accomplishing LU decomposition on $\mathbf{V}$ first, which can be done in $\mathcal{O}(N^2)$ operations, and then each solution for the coefficient vector can be done in $\mathcal{O}(N)$ operations through backsolves.

Finding Integration Weights: When the points are related to Gaussian quadrature, the array `m_weights` will contain the appropriate Gaussian quadrature weights computed using point and weight routines found in Foundations (our polymath library functions). In the case of weights associated with points sets that do not lie at the zeros of Jacobi polynomials, we must compute the weights directly based upon Lagrange interpolation through those points. Although this can be done in general for any point distribution, we will focus here on evenly-spaced points (note that when one applies this approach to Chebyshev points, one arrives at Clenshaw-Curtis quadrature [58]). Within *Nektar++*, given evenly-spaced points on the interval $[-1, 1]$ – if only one point is given, then the weight is set to 2.0 denoting a midpoint integration rule; otherwise, the weights are given by the following expression:

$$w_j = \int_{-1}^{1} \ell_j(\xi)\, d\xi$$

where $\ell_j(\xi)$ is the Lagrange basis function defined over an evenly-spaced set of points $\xi_k \in [-1, 1]$. For 2D and 2D nodal point sets (such as Electrostatic and Fekete Points), this same procedure is used. We first form the Lagrange interpolating functions via the algorithm given above (using the Vandermonde system), and then for each quadrature weight compute the integral of that Lagrange basis function using some known quadrature rule (such as tensor product quadrature via Duffy transformation; this point will be more easily understood after reading Chapter 5)

4.5.2 Basis

The second of the two primary data structures in this directory is Basis. A basis object consists of a BasisKey and the extra information needed to facilitate various operations on the basis. The basis layout of the data structure is shown in Figure 4.3.

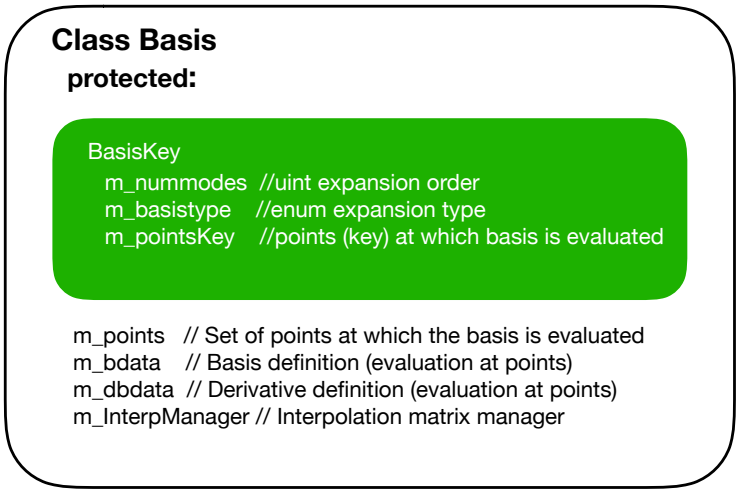


Figure 4.3: The basic Basis class data object. It consists of a BasisKey (used by a BasisManager) and various other data members needed for point operations.

The Basis object consists of a BasisKey (which in turn holds a PointsKey), a pointer to the positions at which the basis is evaluated and then arrays which hold the evaluation of the basis (`m_bdata`) and the evaluation of the derivates of the basis (`m_dbdata`). The storage layout for these data structures are shown in Figure 4.3. The number of rows is given by the number of modes associated with a particular expansion, and the number of columns is given by the number of points. These arrays are stored as contiguous memory to help avoid excessive memory hopping.

4.6 Interpreter

This directory contains two files, AnalyticExpressionEvaluator.hpp and AnalyticExpressionEvaluator.cpp, both of which provide parser and evaluator functionality of analytic expressions.

4.7 Kernel

This directory contains two files, kernel.h and kernel.cpp, both of which are related to the line-SIAC (L-SIAC) filtering [26, 41] of FEM solutions. SIAC filtering takes the form:

$$u^*(x) = \int_{-\infty}^{\infty} K_H \left(\frac{\xi - x}{H} \right) u_h(\xi)\, d\xi$$

where u_h denotes the FEM (continuous Galkerkin or discontinuous Galerkin) solution and K_H represents a kernel function:

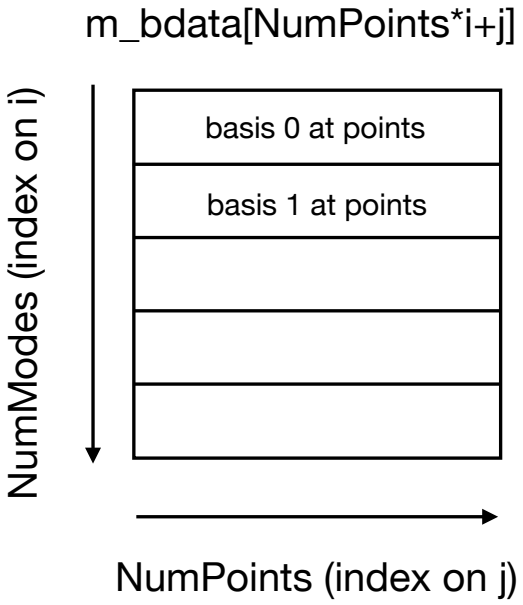


Figure 4.4: Diagram of the basic memory layout of the arrays associated with the basis and its derivatives evaluated at points.

$$K_H(x) = \frac{1}{H} \sum_{\gamma=0}^{M} c_\gamma \psi \left(\frac{x}{H} - \zeta_\gamma \right).$$

In this expression, the functions $\psi(x)$ are B-Splines. The two kernel files in this directory provide all the tools needed to construct the B-Spline functions and the kernel function described above.

4.8 Linear Algebra

This directory contains files related to linear algebra operations. Files such as NekMatrix, BlockMatrix and NekLinearSystem are fundamental to many of the operations we do within *Nektar++*. These files represent our attempt to encapsulate linear algebra operations in a way that make sense from the programmer perspective but for which we do not loose efficiency. As can be seen in these routines and throughout the code, we rely heavily on BLAS (Basic Linear Algebra Subroutines).

Arpack.hpp	Blas.hpp	BlasArray.hpp
BlockMatrix.cpp	BlockMatrix.hpp	CanGetRawPtr.hpp
ExplicitInstantiation.h	Lapack.hpp	MatrixBase.cpp
MatrixBase.hpp	MatrixFuncs.cpp	MatrixFuncs.h
MatrixOperations.cpp	MatrixOperations.hpp	MatrixStorageType.h
MatrixType.h	MatrixVectorMultiplication.cpp	NekLinAlgAlgorithms.hpp
NekLinSys.hpp	NekMatrix.hpp	NekMatrixFwd.hpp
NekPoint.hpp	NekTypeDefs.hpp	NekVector.cpp
NekVector.hpp	NekVectorFwd.hpp	NistSparseDescriptors.hpp
PointerWrapper.h	ScaledMatrix.cpp	ScaledMatrix.hpp
SparseDiagBlkMatrix.cpp	SparseDiagBlkMatrix.hpp	SparseMatrix.cpp
SparseMatrix.hpp	SparseMatrixFwd.hpp	SparseUtils.cpp
SparseUtils.hpp	StandardMatrix.cpp	StandardMatrix.hpp
StorageSmvBsr.cpp	StorageSmvBsr.hpp	TransF77.hpp
blas.cpp		

4.9 Memory

This directory contains three files, NekMemoryManager.hpp, ThreadSpecificPools.hpp and ThreadSpecificPools.cpp. The strategy used within *Nektar++* was to preallocate a pool of arrays that could be used for various operations and then released back to the pool. This idea came about through profiling of the code early on – noticing that the new/delete operation of lots of small arrays used for temporary calculations was greatly slowing down the code. Like with our manager idea, we decided to invest in having a memory pool object what preallocated blocks of memory that could be requested and then returned back to the pool.

If *Nektar++* is compiled with `NEKTAR_MEMORY_POOL_ENABLED`, the MemoryManager allocates from thread specific memory pools for small objects. Large objects are managed with the system supplied new/delete. These memory pools provide faster allocation and deallocation of small objects (particularly useful for shared pointers which allocate many 4 byte objects).

All memory allocated from the memory manager must be returned to the memory manager. Calling delete on memory allocated from the manager will likely cause undefined behavior. A particularly subtle violation of this rule occurs when giving memory allocated from the manager to a shared pointer.

4.10 Polylib

This directory contains polylib.h and polylib.cpp. These files contain foundational routines used for computing various operations related to Jacobi polynomials. The following abbreviations are used throughout the file:

- 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

Points and Weights: The following routines are used to compute points and weights:

- 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: The following routines are used to compute derivative matrices:

- 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: The following routines are used to compute Lagrange interpolation matrices:

- 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: The following routines are used to compute various interpolation operators:

- 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: The following routines are used to evaluate Jacobi polynomials.

- jacobfd – Returns value and derivative of Jacobi polynomial at point z
- jacobd – Returns derivative of Jacobi polynomial at point z (valid at $z = -1, 1$)

4.11 Time Integration

This directory consists of the following files:

TimeIntegrationScheme.h	TimeIntegrationScheme.cpp
TimeIntegrationWrapper.h	TimeIntegrationWrapper.cpp

The original time-stepping classes (found in TimeIntegrationScheme) where implemented by Vos et al. [62] and a detailed explanation of the mathematics and computer science concepts are provided there. After the original implementation, Cantwell et al. extended Vos' original work by adding the time-stepping routines into a factory pattern (found in TimeIntegrationWrapper).

4.11.1 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 such as Adams-Bashforth. 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 [14].

4.11.2 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 containing the variables).

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

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

4.11.3 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 partitioned $(s + r) \times (s + r)$ matrix

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

4.11.4 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 $N \times N$.

4.11.5 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 description of the formulation and a deeper insight of the numerical method see [62].

4.11.6 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 explicitly 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.

4.11.7 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
BDFImplicitOrder1	BDF multi-step scheme of order 1 (implicit)
BDFImplicitOrder2	BDF multi-step scheme of order 2 (implicit)
ClassicalRungeKutta4	Runge-Kutta multi-stage scheme 4th order explicit (old name)
RungeKutta4	Runge-Kutta multi-stage scheme 4th order explicit (new name for ClassicalRungeKutta4)
RungeKutta3_SSP	Nonlinear SSP RungeKutta3 explicit
RungeKutta2_ImprovedEuler	Improved RungeKutta2 explicit (old name meaning Heun's method)
RungeKutta2_SSP	Nonlinear SSP RungeKutta2 explicit (surrogate for RungeKutta2_ImprovedEuler)
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 (old name)
RungeKutta2	Classical RungeKutta2 method (new name for Midpoint)
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>
```

4.11.8 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

```
NekDouble timestep    = 0.1;
NekDouble time        = 0.0;
int NumSteps          = 1000;

YourClass solver(Input);

LibUtilities::TimeIntegrationMethod  TIME_SCHEME;
LibUtilities::TimeIntegrationSchemeOperators ODE;

ODE.DefineOdeRhs(&YourClass::YourExplicitOperatorFunction,solver);
ODE.DefineProjection(&YourClass::YourProjectionFunction,solver);
ODE.DefineImplicitSolve(&YourClass::YourImplicitOperatorFunction,solver);

Array<OneD, LibUtilities::TimeIntegrationSchemeSharedPtr> IntScheme;

LibUtilities::TimeIntegrationSolutionSharedPtr ode_solution;

Array<OneD, Array<OneD, NekDouble> > U;

TIME_SCHEME = LibUtilities::eForwardEuler;
int numMultiSteps=1;
IntScheme = Array<OneD,LibUtilities::TimeIntegrationSchemeSharedPtr>(
    numMultiSteps);
LibUtilities::TimeIntegrationSchemeKey IntKey(TIME_SCHEME);
IntScheme[0] = LibUtilities::TimeIntegrationSchemeManager()[IntKey];
ode_solution = IntScheme[0]->InitializeScheme(timestep,U,time,ODE);

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

We can distinguish three different sections in the code above

Definitions

```
NekDouble timestep    = 0.1;
NekDouble time        = 0.0;
int NumSteps          = 1000;

YourClass equation(Input);

LibUtilities::TimeIntegrationMethod  TIME_SCHEME;
LibUtilities::TimeIntegrationSchemeOperators ODE;

ODE.DefineOdeRhs(&YourClass::YourExplicitOperatorFunction,equation);
ODE.DefineProjection(&YourClass::YourProjectionFunction, equation);
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.

Initialisations

```
Array<OneD,LibUtilities::TimeIntegrationSchemeSharedPtr> IntScheme;

LibUtilities::TimeIntegrationSolutionSharedPtr ode_solution;

Array<OneD, Array<OneD, NekDouble> > U;

TIME_SCHEME = LibUtilities::eForwardEuler;
int numMultiSteps=1;
IntScheme = Array<OneD,LibUtilities::TimeIntegrationSchemeSharedPtr>(
    numMultiSteps);
LibUtilities::TimeIntegrationSchemeKey IntKey(TIME_SCHEME);
IntScheme[0] = LibUtilities::TimeIntegrationSchemeManager()[IntKey];
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.

Integration

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

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.

```
class YourCalss
{
public:
    YourClass(INPUT);

    ~YourClass(void);

    void YourExplicitOperatorFunction(
        const Array<OneD, Array<OneD, NekDouble> > & inarray,
              Array<OneD, Array<OneD, NekDouble> > & outarray,
        const NekDouble time);

    void YourProjectionFunction(
        const Array<OneD, Array<OneD, NekDouble> > & inarray,
              Array<OneD, Array<OneD, NekDouble> > & outarray, const
              NekDouble time);

    void YourImplicitOperatorFunction(
        const Array<OneD, Array<OneD, NekDouble> > & inarray,
              Array<OneD, Array<OneD, NekDouble> > & outarray, const
              NekDouble time,
        const NekDouble lambda);

    void InternalMethod1(...);

    NekDouble internalvariale1;

protected:
    ...
```

```
30
31 private:
32     ...
33 };
```

4.11.9 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. $\boldsymbol{L}^{\mathcal{HD}}\boldsymbol{y}^{\mathcal{D}}+\boldsymbol{L}^{\mathcal{HH}}\boldsymbol{y}^{\mathcal{H}}$ 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:

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

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. $\boldsymbol{H}^{\mathcal{HH}}\boldsymbol{y}^{\mathcal{H}} = \boldsymbol{b}^{\mathcal{H}} - \boldsymbol{H}^{\mathcal{HD}}\boldsymbol{y}^{\mathcal{D}}$

4.11.10 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)

4.11.11 Examples of already implemented time stepping schemes

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

Forward Euler

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

Backward Euler

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

2^{nd} order Adams-Bashforth

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

1^{st} order IMEX Euler-Backward/ Euler-Forward

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

2^{nd} order IMEX Backward Different Formula & Extrapolation

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

with

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



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.

3rdorder IMEX Backward Different Formula & Extrapolation

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

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.

2^{nd} order Adams-Moulton

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

Midpoint Method

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

RK4: the standard fourth-order Runge-Kutta scheme

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

2^{nd} 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^{rd} 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^{rd} order L-stable, three stage IMEX DIRK(3,4,3)

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

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

with

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

Chapter 5

Inside the Library: StdRegions

In this chapter, we walk the reader through the different components of the StdRegions Directory. We begin with a discussion of the mathematical fundamentals, for which we use the book by Karniadakis and Sherwin [44] as our principle reference. We then provide the reader with an overview of the primary data structures introduced within the StdRegions Directory (often done through C++ objects), and then present the major algorithms – expressed as either object methods or functions – employed over these data structures.

5.1 The Fundamentals Behind StdRegions

The idea behind standard regions can be traced back to one of the fundamental principles we learn in calculus: differentiation and integrating on arbitrary regions are often difficult; however, differentiation and integration have been worked out on canonical (straight-sided or planar-sided), right-angled, coordinate aligned domains. In the case of *Nektar++*, we do not need to work on truly arbitrary domains, but rather on seven fundamental domains: segments in 1D, triangles and quadrilaterals in 2D, and tetrahedra, hexahedra, prisms and pyramids in 3D. Since *Nektar++* deals with polynomial methods, the natural domain over which reference elements should be build is $[-1, 1]$ as this is the interval over which Jacobi Polynomials are defined and over which Gaussian quadrature is defined [19]. We show in Figure 5.1 a pictorial representation of the standard segment, standard quadrilateral (often shorthanded quad) and standard triangle.

The standard quad and standard hexahedra (shorthanded 'hex') are geometrically tensor-product constructions defined on $[-1, 1]^d$ for $d = 2$ and $d = 3$ respectively. The standard triangle is constructed by taking $\xi_1 \in [-1, 1]$ and taking $\xi_2 \leq -\xi_1$. The standard tetrahedra (shorthanded 'tet') is built upon the standard triangle and has all four faces being triangles, with the two triangles along the coordinate directions looking like the standard triangle. The standard prism consists of a standard triangle along the $\xi_1 - -\xi_2$ plane extruded into the third direction (yielding three quadrilateral faces). The standard pyramid consists of a standard quadrilateral at the base with four triangular faces reaching

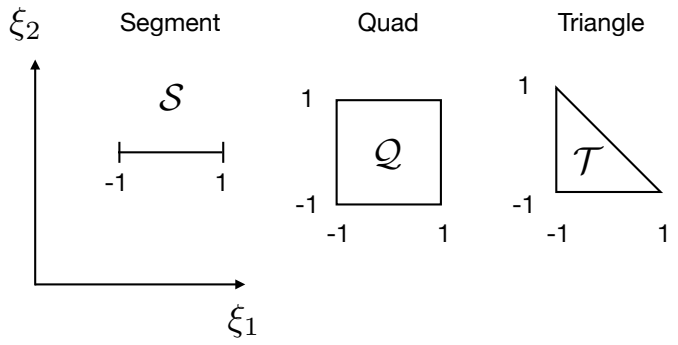


Figure 5.1: Example of the 1D and 2D reference space elements (segment, quad and triangle).

up to its top vertex We show this pictorially in Figure 5.2.

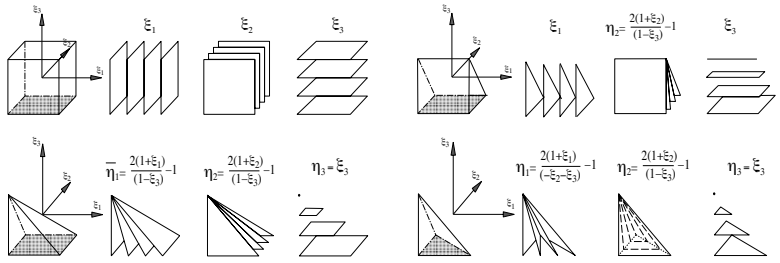


Figure 5.2: Hexahedron to tetrahedron transformation showing how to go from an element on $[-1,1]^3$ (i.e. a standard hexahedron) to the three other element types which are subsets on $[-1,1]^3$. Image taken from [44].

Regardless of the particular element type, we use the fact one can build polynomial spaces over these geometric objects. In the case of triangles and tetrahedra, these are polynomial spaces in the mathematical sense, i.e. $\mathcal{P}(k)$ spaces. In the case of quadrilaterals and hexahedra, these are bi– and tri-polynomial spaces, i.e. $\mathcal{Q}(k)$. In the case of prisms and pyramids, the spaces are more complicated and are a mixture in some sense of these spaces, but they are indeed polynomial in form. This allows us to define differentiation exactly and to approximate integrals exactly up to machine precision.

In what is to follow, we describe (1) how you can build differentiation and integration operators over standard regions by mapping them to a domain over which you can exploit index separability; and then (2) how you can build differentiation and integration operators natively over various standard region shapes. The former allows one to benefit from tensor-product operators, and is at present the primary focus of all *Nektar++* operators. The latter as implemented for nodal element types is currently an *is-a* class definition extended from the (tensor-product-based) standard element definitions.

5.1.1 Reference Element Transformations That Facilitate Separability

Differentiation and integration over the standard elements within *Nektar++*, in general, always try to map things back to tensor-product (e.g. tensor-contraction indexing). This allows one to transform operators that would normally have iterators that go from $i = 0, \ldots, N^d$ where d is the dimension of the shape to operators constructed with $d \cdot N$ operations (i.e., the product of one-dimensional operations).

As an example (to help the reader gain an intuitive understanding), consider the integration of a function $f(\vec{x})$ over a hexahedral element. If we had a quadrature rule in 3D that needed N^d points to integrate this function exactly, we would express the operation as:

$$\int_E f(\vec{x})\, d\vec{x} \approx \sum_{i=0}^{N^d} \omega_i\, f(\vec{z}_i)$$

where E denotes our d-dimensional element and the set $Q = \{\vec{z}_i, \omega_i\}$ denotes the set of points and quadrature weights over the element E. Now, suppose that both our function $f(\vec{x})$ and our Q were separable: that is, $f(\vec{x})$ can be written as $f_1(x_1) \cdot f_2(x_2) \cdot f_3(x_3)$ where $\vec{x} = (x_1, x_2, x_3)^T$ and Q can we written in terms of 1D quadrature: $\vec{z}_i = z^{(1)}_{i_1} \cdot z^{(2)}_{i_2} \cdot z^{(3)}_{i_3}$ with the index handled by an index map σ defined by $i = \sigma(i_1, i_2, i_3) = i_1 + N \cdot i_2 + N^2 \cdot i_3$, and similarly for the weights ω_i. We can then re-write the integral above as follows:

$$\begin{aligned}
\int_E f(\vec{x})\, d\vec{x} &= \int_E f(x_1) \cdot f(x_2) \cdot f(x_3)\, dx_1\, dx_2\, dx_3 \\
&\approx \sum_{i=0}^{N^d} \omega_i\, f(\vec{z}_i) \\
&= \sum_{i_1=0}^{N} \sum_{i_2=0}^{N} \sum_{i_3=0}^{N} \omega_{\sigma(i_1,i_2,i_3)} \left[f(z^{(1)}_{i_1}) \cdot f(z^{(2)}_{i_2}) \cdot f(z^{(3)}_{i_3}) \right] \\
&= \left[\sum_{i_1=0}^{N} \omega_{i_1}\, f(z^{(1)}_{i_1}) \right] \cdot \left[\sum_{i_2=0}^{N} \omega_{i_2}\, f(z^{(2)}_{i_2}) \right] \cdot \left[\sum_{i_3=0}^{N} \omega_{i_3}\, f(z^{(3)}_{i_3}) \right]
\end{aligned}$$

As you can see, when the functions are separable and the quadrature (or collocating points) can be written in separable form, we can take $\mathcal{O}(N^d)$ operators and transform them into $\mathcal{O}(d \cdot N))$ operators. The above discussion is mainly focusing on the mathematical transformations needed to accomplish this; in subsequent sections, we will point out the memory layout and index ordering (i.e. now σ is ordered and implemented) to gain maximum performance.

The discussion of tensor-product operations on quadrilaterals and hexahedra may seem quite natural as the element construction is done with tensor-products of the segment;

however, how do you create these types of operations on triangles (and anything involving triangles such as tetrahedra, prisms and pyramids)? The main mathematical building block we use that allows such operators is a transformation due to Duffy [28], which was subsequently used by Dubiner [27] in the context of finite elements and was extended to general polyhedral types by Ainsworth [7, 6].

An image denoting the Duffy transformation is shown in Figure 5.3. On the left is an example of the right-sided standard triangle with coordinate system (ξ_1, ξ_2), and on the right is the separable tensor-product domain with coordinates (η_1, η_2). We often refer to this transformation as "collapsed coordinates" as it appears as the "collapsing" of a quadrilateral domain to a triangle. Note that the edge along $\eta_1 = 1$ on the quadrilateral collapses down to the single point $(\xi_1, \xi_2) = (1, 1)$. This, in general, does not cause issues with our integration over volumes as this is a point of measure zero. However, special case will need be taken when doing edge integrals. We will make it a point to highlight those places in which special care is needed.

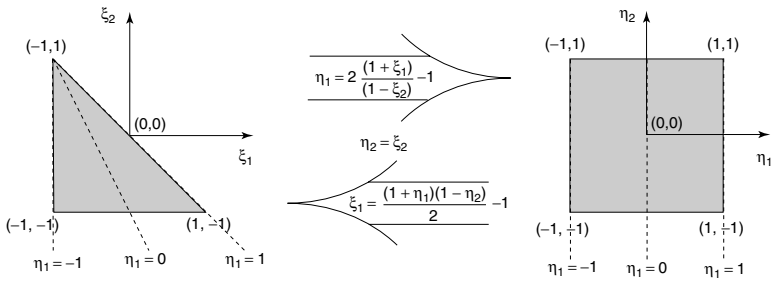


Figure 5.3: Duffy mapping between a right-sided triangle and a square domain.

With the 2D Duffy transformation in place, we can actually create all four 3D element types from the starting point of a hexahedra (our base tensor-product shape). One collapsing operation yields a prism. The second collapsing operation yields a pyramid. Lastly, a final collapsing operation yields a tetrahedra. A visual representation of these operations are shown in Figure 5.4.

This is meant merely to be a summary of tensor-product operations, enough to allow us to discuss the basic data types and algorithms within *Nektar++*. If more details are needed, we encourage the reader to consult Karniadakis and Sherwin [44] and the references therein.

5.1.2 Reference Elements On Primitive Geometric Types

Although the primary backbone of *Nektar++* is tensor-product operations across all element types through the use of collapsed coordinates, we have implemented two commonly used non-tensorial basis set definitions. These are based upon Lagrange polynomial construction from a nodal (collocating) point set. As derived element types (in the $C++$ sense), we have implemented NodalStdTriangle (and Tet) based upon

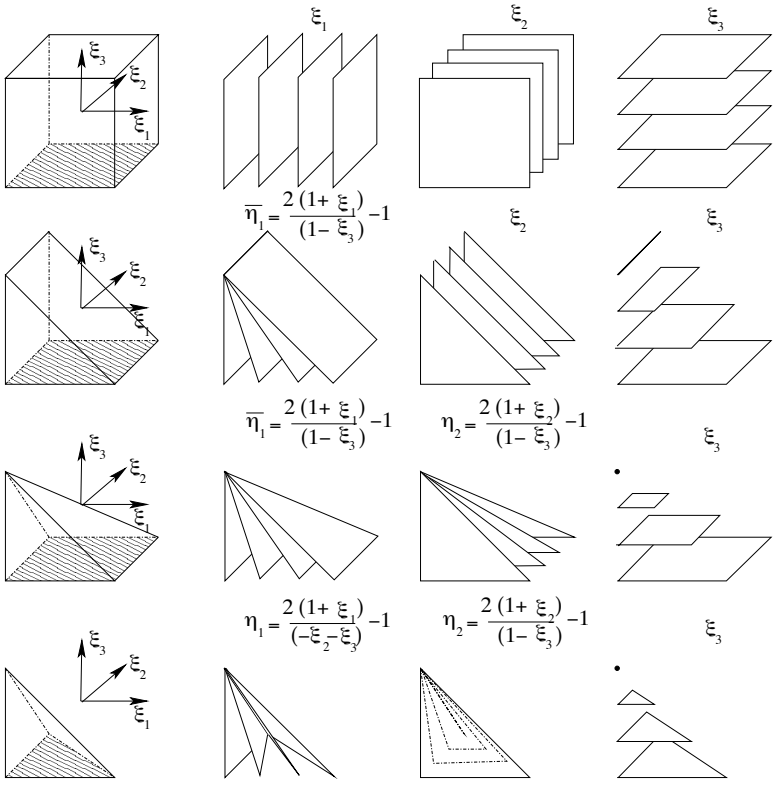


Figure 5.4: Duffy mapping application to generate all the 3D element types. Image taken from [44].

the electrostatic points of Hesthaven [38] and the Fekete points of Taylor and Wingate [56, 57].

Although these types of elements, at first glance, might seem to be “sub-optimal" (in terms of operations), there are various reasons why people might choice to use them. For instance, the point set you use for defining the collocation points dictates the interpolative projection operator (and correspondingly the properties of that operator) – an important issue when evaluating boundary conditions, initial conditions and for some non-linear operator evaluations. Within the *Nektar++* team, we started an examination of this within the paper by Kirby and Sherwin [45]. There has since been various papers in the literature (beyond the scope of this developer’s guide) that give the pros and cons of tensor-product (separable) and non-tensor-product element types.

5.2 The Fundamental Data Structures within StdRegions

In almost all object-oriented languages (which includes $C++$), there exists the concepts of *class attributes* and *object attributes*. Class attributes are those attributes shared by all object instances (both immediate and derived objects) of a particular class definition, and object attributes (sometimes called data members) are those attributes whose values vary from object to object and hence help to characterize (or make unique) a particular object. In $C++$, object attributes are specified a header file containing class declarations; within a class declaration, attributes are grouped by their accessibility: *public* attributes, *protected* attributes and *private* attributes. A detailed discussion of the nuances of these categories are beyond the scope of this guide; we refer the interested reader to the following books for further details: [55, 51]. For our purposes, the main thing to appreciate is that categories dictate access patters within the inheritance hierarchy and to the "outside" world (i.e. access from outside the object). We have summarized the relationships between the categories and their accessibility in Tables 5.1, 5.2 and 5.3 [1].

Table 5.1: Accessibility in Public Inheritance

Accessibility	private variables	protected variables	public variables
Accessibility from own class?	yes	yes	yes
Accessibility from derived class?	no	yes	yes
Accessibility from 2nd derived class?	no	yes	yes

Table 5.2: Accessibility in Protected Inheritance

Accessibility	private variables	protected variables	public variables
Accessibility from own class?	yes	yes	yes
Accessibility from derived class?	no	yes	yes (inherited as protected variable)
Accessibility from 2nd derived class?	no	yes	yes

Table 5.3: Accessibility in Private Inheritance

Accessibility	private variables	protected variables	public variables
Accessibility from own class?	yes	yes	yes
Accessibility from derived class?	no	yes (inherited as private variable)	yes (inherited as private variable)
Accessibility from 2nd derived class?	no	no	no

[1] These tables are based upon information provided at http://www.programiz.com/cpp-programming/public-protected-private-inheritance, accessed 6 April 2018.

Within the StdRegions directory of the library, there exists a class inheritance hierarchy designed to try to encourage re-use of core algorithms (while simultaneously trying to minimize duplication of code). We present this class hierarchy in Figure 5.5.

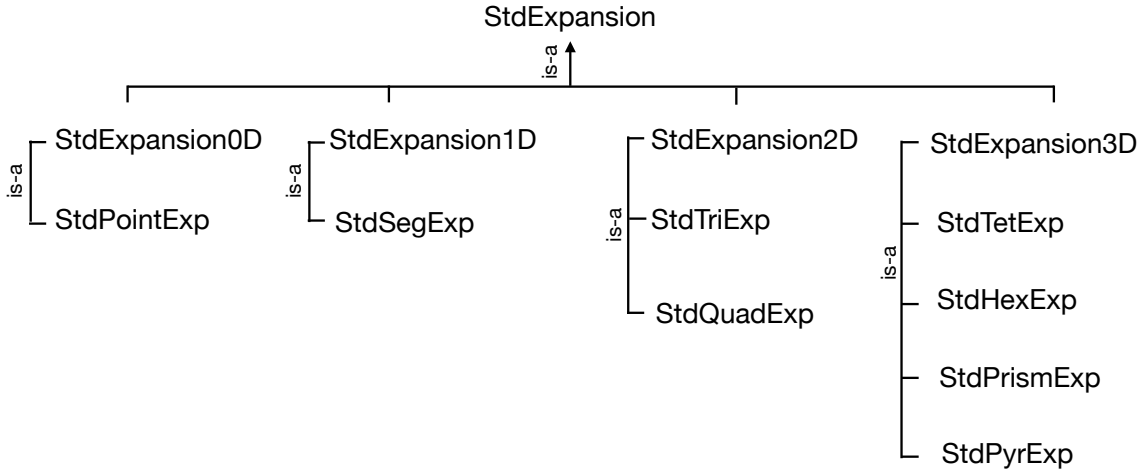


Figure 5.5: Class hierarchy derived from StdExpansion, the base class of the StdRegions Directory.

As is seen in Figure 5.5, the StdRegions hierarchy consists of three levels: the base level from which all StdRegion objects are derived is StdExpansion. This object is then specialized by dimension, yielding StdExpansion0D, StdExpansion1D, StdExpansion2D and StdExpansion3D. The dimension-specific objects are then specialized based upon shape.

The object attributes (variables) at various levels of the hierarchy can be understood in light of Figure 5.6. At its core, an expansion is a means of representing a function over a canonically-defined region of space evaluated at a collection of point positions. The various data members hold information to allow all these basic building blocks to be specified.

The various private, protected and public data members contained within StdRegions are provided in the subsequent sections.

5.2.1 Variables at the Level of StdExpansion

Private: There are private methods but no private data members within StdExpansion.

Protected:

- Array of Basis Shared Pointers: `m_base`
- Integer element id: `m_elmt_id`

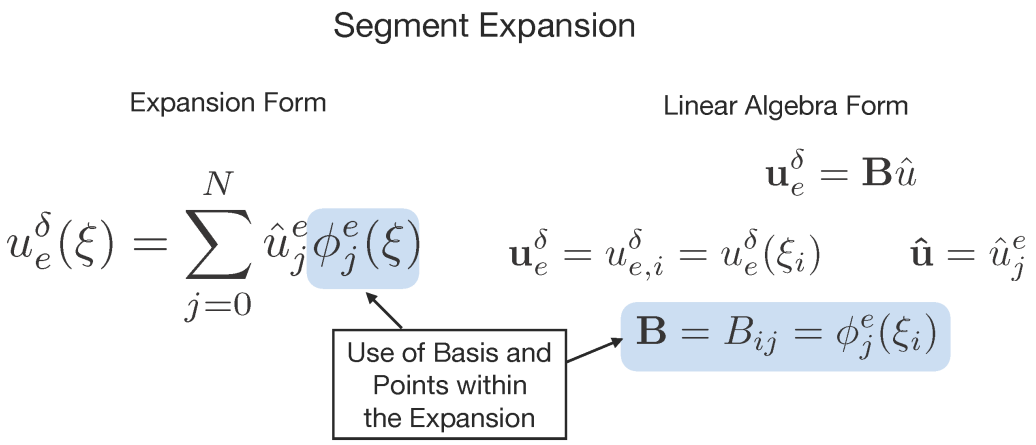


Figure 5.6: Diagram to help understand the various data members (object attributes) contained within StdRegions and how they connect with the mathematical representation presented earlier.

- Integer total number of coefficients in the expansion: `m_ncoeffs`
- Matrix Manager: `m_stdMatrixManager`
- Matrix Manager: `m_stdStaticCondMatrixManager`
- IndexKeyMap Matrix Manager: `m_IndexMapManager`

Public: There are public methods but no public data members within StdExpansion.

5.2.2 Variables at the Level of StdExpansion$D for various Dimensions

Private: There are private methods but no private data members within StdExpansion$D.

Protected:

- 0D and 1D: std::map<int, NormalVector> `m_vertexNormals`

- 2D: Currently does not have any data structure. It should probably have `m_edgeNormals`
- 3D: std::map<int, NormalVector> `m_faceNormals`
- 3D: std::map<int, bool> `m_negatedNormals`

Public: There are public methods but no public data members within StdExpansion$D.

5.2.3 Variables at the Level of Shape-Specific StdExpansions

Private:

Protected:

Public:

5.2.4 General Layout of the Basis Functions in Memory

5.2.5 General Layout

Basis functions are stored in a 1D array indexed by both mode and quadrature point. The fast index runs over quadrature points while the slow index runs over modes. This was done to match the memory access pattern of the inner product, which is the most frequently computed kernel for most solvers.

Bases are built from the tensor product of three different equation types (subsequently called Type 1, Type II and Type III respectively):

$$\phi_p(z) = \begin{cases} \frac{1-z}{2} & p = 0 \\ \frac{1+z}{2} & p = 1 \\ \left(\frac{1-z}{2}\right)\left(\frac{1+z}{2}\right) P_{p-1}^{1,1}(z) & 2 \leq p < P \end{cases} \tag{5.1}$$

$$\phi_{pq}(z) = \begin{cases} \phi_q(z) & p = 0 & 0 \leq q < P \\ \left(\frac{1-z}{2}\right)^p & 1 \leq p < P & q = 0 \\ \left(\frac{1-z}{2}\right)^p \left(\frac{1+z}{2}\right) P_{q-1}^{2p-1,1}(z) & 1 \leq p < P, & 1 \leq q < P-p \end{cases}$$

$$\phi_{pqr}(z) = \begin{cases} \phi_{qr} & p = 0 & 0 \leq q < P & 0 < r < P-q \\ \left(\frac{1-z}{2}\right)^{p+q} & 1 \leq p < P, & 0 \leq q < P-p, & r = 0 \\ \left(\frac{1-z}{2}\right)^{p+q} \left(\frac{1+z}{2}\right) P_{r-1}^{2p+2q-1,r}(z) & 1 \leq p < P, & 0 \leq q < P-p, & 1 \leq r < P-p-qr. \end{cases}$$

Here, P is the polynomial order of the basis and $P_p^{\alpha,\beta}$ are the p^{th} order jacobi polynomial.

A Note Concerning Adjustments For C_0 Continuity

Before going further it is worth reviewing the spatial shape of each node. The term $\frac{1+z}{2}$ is an increasing function which is equal to zero at $z = -1$ and equal to one at $z = 1$. Similarly, $\frac{1-z}{2}$ is a decreasing function which is equal to one at $z = -1$ and equal to zero at $z = +1$. These two functions are thus non-zero at one of each of the boundaries. If we need to maintain C_0 continuity with surrounding elements (as we do in the continuous Galerkin method), then these local modes must be assembled together with the correct local modes in adjacent elements to create a continuous, global mode. For instance $\frac{1+z}{2}$ in the left element would be continuous with $\frac{1-z}{2}$ in the right element. The union of these two modes under assembly form a single "hat" function. By contrast, functions of the form

$$\frac{1-z}{2}\frac{1+z}{2}$$

are zero at both end points $z = \pm 1$. As a result, they are trivially continuous with any other function which is also equal to zero on the boundary. These "bubble" functions may be treated entirely locally and thus are used to construct the interior modes of a basis. Only bases with $p > 1$ have interior modes.

All of this holds separately in one dimension. Higher dimensional bases are constructed via the tensor product of 1D basis functions. As a result, we end up with a greater number of possibilities in terms of continuity. When the tensor product is taken between two bubble functions from different bases, the result is still a bubble function. When the tensor product is taken between a hat function and a bubble function we get "edge" modes (in both 2D and 3D). These are non-zero along one edge of the standard domain and zero along the remaining edges. When the tensor product is taken between two hat functions, they form a "vertex" mode which reaches its maximum at one of the vertices and is non-zero on two edges. The 3D bases are constructed similarly.

Based upon this convention, the 1D basis set consists of vertex modes and bubble modes. The 2D basis function set consists of vertex modes, edge modes and bubble modes. The 3D basis set contains vertex modes, edge modes, face modes and bubble modes.

5.2.6 2D Geometries

Quadrilateral Element Memory Organization

Quads have the simplest memory organization. The quadrilateral basis is composed of the tensor product of two Type I functions $\phi_p(\xi_{0,i})\phi_q(\xi_{1,j})$. This function would then be indexed as

```
1 basis0[p*nq0 + i] * basis1[q*nq1 + j]
```

where nq<b> is the number of quadrature points for the b^{th} basis. Unlike certain mode orderings (e.g. Karniadakis and Sherwin [44]), the two hat functions are accessed as the

first and second modes in memory with interior modes placed afterward. Thus,

```
basis[i], basis[nq + i]
```

correspond to $\frac{1-z}{2}$ and $\frac{1+z}{2}$ respectively.

Triangle Element Memory Organization

Due to the use of collapsed coordinates, triangular element bases are formed via the tensor product of one basis function of Type I, and one of Type II, i.e. $\phi_p(\eta_{0,i}\phi_p q(\eta 1, j))$. Since ϕ_p is also a Type I function, its memory ordering is identical to that used for quads. The second function is complicated by the mixing of ξ_0 and ξ_1 in the construction of η_1.

In particular, this means that the basis function has two modal indices, p and q. While p can run all the way to P, The number of q modes depends on the value of the p index q index such that $0 \leq q < P - p$. Thus, for $p = 0$, the q index can run all the way up to P. When p $= 1$, it runs up to $P - 1$ and so on. Memory is laid out in the same way starting with $p = 0$. To access all values in order, we write

```
mode = 0
for p in P:
    for q in P - p:
        out[mode*nq + q] = basis0[p*nq]*basis1[mode*nq + q]
    mode += P-p
```

Notice the use of the extra "mode" variable. Since the maximum value of q changes with p, basis1 is not simply a linearized matrix and instead has a triangular structure which necessitates keeping track of our current memory location.

The collapsed coordinate system introduces one extra subtlety. The mode

$$\phi_1(\eta_1)\phi_1(\eta_2)$$

represents the top right vertex in the standard basis. However, when we move to the standard element basis, we are dealing with a triangle which only has three vertices. During the transformation, the top right vertex collapses into the top left vertex. If we naively construct an operators by iterating through all of our modes, the contribution from this vertex to mode Φ_{01} will not be included. To deal with this, we add its contribution as a correction when computing a kernel. The correction is $\Phi_{01} = \phi_0\phi_{01} + \phi_1\phi_{10}$ for a triangle.

5.2.7 3D Geometries

Hexahedral Element Memory Organization

The hexahedral element does not differ much from the quadrilateral as it is the simply the product of three Type I functions.

$$\Phi_{pqr} = \phi_p(\xi_0)\phi_q(\xi_1)\phi_r(\xi_2).$$

Prismatic Element Memory Organization

Cross sections of a triangular prism yield either a quad or a triangle based chosen direction. The basis, therefore, looks like a combination of the two different 2D geometries.

$$\Phi_{pqr} = \phi_p(\eta_0)\phi_q(\xi_1)\phi_{pr}(\eta_1).$$

Taking $\phi_p\phi_{pr}$ on its own produces a triangular face while taking $\phi_p\phi_q$ on its own produces a quadrilateral face. When the three basis functions are combined into a single array (as in the inner product kernel), modes are accessed in the order p,q,r with r being the fastest index and p the slowest. The access pattern for the prism thus looks like

```
mode_pqr = 0
mode_pr = 0
for p in P:
    for q in Q:
        for r in P - p:
            out[mode_pqr*nq + r] = basis0[p*nq]*basis1[q*nq]*basis2[mode_pr
 + r]
        mode_pqr += P - p
    mode_pr += P - p
```

As with the triangle, we have to deal with complications due to collapsed coordinates. This time, the singular vertex from the triangle occurs along an entire edge of the prism. Our correction must be added to a collection of modes indexed by q

$$\Phi_{0q1} += \phi_1\phi_q\phi_{10}.$$

Tetrahedral Element Memory Organization

The tetrahedral element is the most complicated of the element constructions. It cannot simply be formed as the composition of multiple triangles since η_2 is constructed by mixing three coordinate directions. We thus need to introduce our first Type III function.

$$\Phi_{pqr}(\eta_0, \eta_1, \eta_2) = \phi_p(\eta_0)\phi_{pq}(\eta_1)\phi_{pqr}(\eta_2).$$

The r index is constrained by both p and q indices. It runs from $P - p - q$ to 1 in a similar manner to the Type II function. Our typical access pattern is thus

```
mode_pqr = 0
mode_pq = 0
for p in P:
    for q in P - p:
        for r in P - p - r:
            out[mode_pqr*nq + r] = basis0[p*nq]*basis1[mode_pq + q]*basis2[
    mode_pqr + r]
        mode_pqr += (P - p - r)
    mode_pq += (P - p)
```

The tetrahedral element also has to add a correction due to collapsed coordinates. Similar to the prism, the correction must be applied to an entire edge indexed by r

$$\Phi_{01r} += \phi_1\phi_1\phi_{11r}.$$

Pyramidic Element Memory Organization

Like the tetrahedral element, a pyramid contains a collapsed coordinate direction which mixes three standard coordinates from the standard region. Unlike the tetrahedra, the collapse only occurs along one axis. Thus it is constructed from two Type I functions and one Type III function

$$\Phi_{pqr} = \phi_p(\eta_1)\phi_q(\eta_2)\phi_{pqr}(\eta_3).$$

The product $\phi_p\phi_q$ looks like the a quad construction which reflects the quad which serves as the base of the pyramid. A typical memory access looks like

```
mode_pqr = 0
for p in P:
    for q in P - p:
        for r in P - p - r:
            out[mode_pqr*nq + r] = basis0[p*nq]*basis1[q*nq]*basis2[
    mode_pqr*nq + r]
        mode_pqr += (P - p - r)
```

5.3 The Fundamental Algorithms within StdRegions

As stated in the introduction, this section of this guide is structured in question-answer form. This is not meant to capture every possible question asked of us on the *Nektar++* users list; however, this set of (ever-growing) questions are meant to capture the "big ideas" that developers want to know about how StdRegions work and how they can be used.

In this section, we will through question and answer format try to cover the following basic algorithmic concepts that are found within the StdRegions part of the library:

- xx

With the big ideas in place, let us now start into our questions.

Question:

Chapter 6

Inside the Library: SpatialDomains

In this chapter, we walk the reader through the different components of the SpatialDomains Directory. We begin with a discussion of the mathematical fundamentals, for which we use the book by Karniadakis and Sherwin [44] as our principle reference. We then provide the reader with an overview of the primary data structures introduced within the SpatialDomains Directory (often done through C++ objects), and then present the major algorithms – expressed as either object methods or functions – employed over these data structures.

The SpatialDomains Directory and its corresponding class definitions serve two principal purposes:

1. To hold the elemental geometric information (i.e. vertex information, curve information and reference-to-world mapping information); and

2. To facility reading in and writing out geometry-related information.

When designing Nektar++, developing a class hierarchy for StdRegions (those fundamental domains over which we define integration and differentiation) and LocalRegions (i.e. elements in world-space) was fairly straightforward following [44]. For instance, a triangle in world-space *is-a* standard triangle. The first question that arose was where to store geometric information, as information within the LocalRegions element or as information encapsulated from the element so that multiple Expansions could all point to the same geometric information. The decision we made was to store geometric information – that is, the vertex information in world-space that defines an element and the edge and face curvature information – in its own data structure that could be shared by multiple Expansions (functions) over the same domain (element) in world-space. Hence SpatialDomains started as the directory containing Geometry and GeomFactors class definitions to meet the first item listed above. A LocalRegion *is-a* StdRegion and *has-a* SpatialDomain (i.e. Geometry and GeomFactors).

We then realized that in order to jump-start the process of constructing elements and combining them together into MultiRegions (collections of elements that represent a (sub)-domain of interest), we needed devise a light-weight data structure into which we could load geometric information from our geometry file and from which we could then construct Expansions (with their mappings, etc.). The light-weight data structure we devised was MeshGraph, and it was meant to meet the second item listed above.

6.1 The Fundamentals Behind SpatialDomains

As mentioned in our discussions of the fundamentals of StdRegions (i.e. Section 5.1), one of the most fundamental tools from calculus that we regularly employ is the idea of mapping from general domains to canonical domains. General domains are the regions in world space over which we want to solve engineering problems, and thus want to be able to take derivatives and compute integrals. But as we learned in calculus, it is often non-trivial to accomplish differentiation or integration over these regions. We resort to the mapping arbitrary domains back to canonical domains over which we can define various operations. We introduced our canonical domains, which we call standard regions, in Chapter 5. We refer to our world space regions as local regions, which we will present in Chapter 7. How these two are connected are via SpatialDomains.

As will be further discussed in Section 6.2, there are two fundamental purposes served by SpatialDomains: (1) holding basic geometric information (e.g. vertex values and curvature information) and (2) holding geometric factors information. The former information relates to the geometric way we map standard regions to local regions. The latter information relates to how we use this map to allow us to accomplish differentiation and integration of functions in world space via operations on standard regions with associated map (geometric) information. In this section, we will highlight the important mathematical principles that are relevant to this section. We will first discuss the mapping itself: vertex and curvature information and how it is used. We will then discuss how geometric factor information is computed. We break this down into two subsections following the convention of the code. We will first discussed what is labeled in the code as *Regular*, and denotes mappings between elements of the same dimension (i.e. standard region triangles to 2D triangles in world space). Although our notation will be slightly different, we will use [44] as our guide; we refer the interested reader there for a more in-depth discussion of these topics. We will then discuss what is labeled in the code as *Deformed*, and denotes the mappings between standard region elements and their world space variants in a higher embedded dimension (i.e. standard region triangles to triangles lying on a surface embedded in a 3D space representing a manifold). Since the manifold work within *Nektar++* was introduced as an area of research, we will use [18] as our guide. The notation therein is slightly different than that of [44] because of the necessity to use broader coordinate system transformation principles (e.g. covariance and contravariance of vectors, etc.). We will abbreviate the detailed mathematical derivations here, but encourage the interested reader to review [18] and references therein as needed. For those unfamiliar with covariant and contravariant spaces, we encourage the reader to review [13].

6.1.1 Vertex and Curvature Mapping Information

When we load in a mesh into *Nektar++*, elements are often described in world space based upon their vertex positions. In traditional FEM formats, this can be as simple as a list of d-dimensional vertex coordinates, followed by a list of element definitions: each row holding four integers (in the case of tetrahedra) denoting the four vertices in the vertex list that comprise an element. *Nektar++* uses an HTML-based geometry file with a more rich definition of the basic geometric information that just described; we encourage developers and users to review our User Guide(s) for the organization and conventions used within our geometry files. For the purposes of this section, the important pieces of information are as follows. Let us assume that for each element, we have through our MeshGraph data structure (described in the next section) access to the vertex positions of an element. In general, each vertex $\vec{v}_j$ is a n-tuple of dimension d denoting the dimension in which the points are specified in world space. For example, when considering a quadrilateral on the 2D plane, our vertex points each contain two coordinates denotes the x– and y–coordinates. As shown in Figure 6.1, we can express the mapping between a world space quadrilateral and the standard region quadrilateral on $[-1,1] \times [-1,1]$ via a bi-linear mapping function $\chi(\vec{\xi})$.

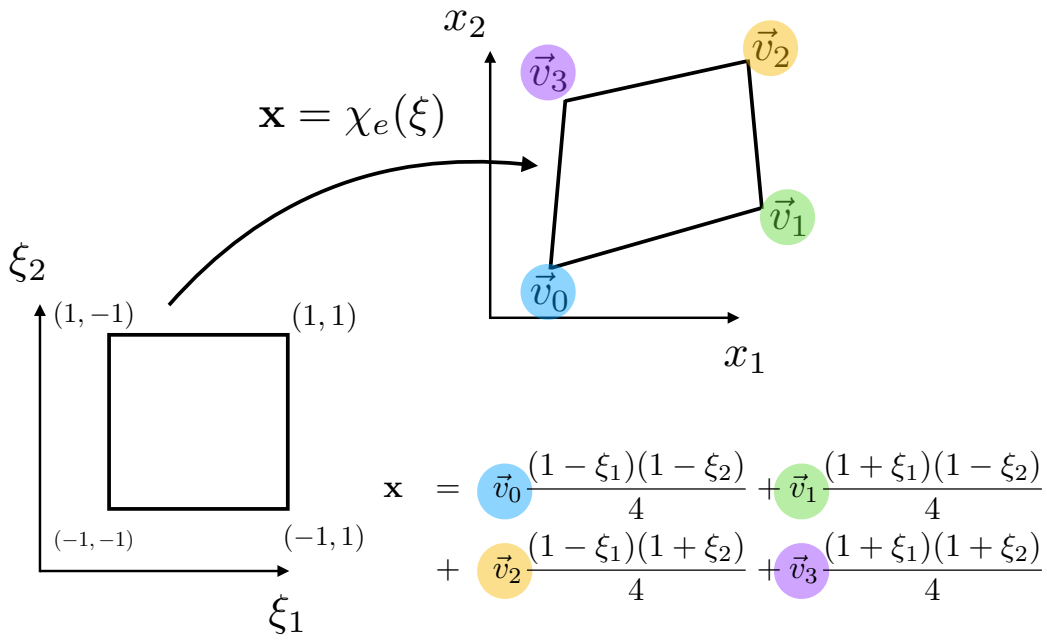


Figure 6.1: Reference space to world space mapping of a 2D quadrilateral to a straight-sided (2D) quadrilateral in the plane via a bi-linear (i.e., $Q(1)$) mapping.

In the case of segments, this mapping function $\chi(\vec{\xi})$ is a function of one variable and is merely an affine mapping. In two dimensions, the mapping of a straight-sided triangle is a linear mapping (i.e., P(1) in the language of traditional finite elements – a total

degree $k = 1$ space) and the mapping of a straight-sided quadrilateral is a bi-linear mapping (i.e, Q(1) in the language of traditional finite elements – a degree $k = 1$ map along each coordinate direction combined through tensor-product). In three dimensions, the mapping of a planar-sided tetrahedron is also a linear mapping, the mapping of a planar-sided hexahedron is a tri-linear mapping, and the prism and pyramid are mathematically somewhere in-between these two canonical types as given in [44]. The key point is that in the case of straight-sided (or planar-sided) elements, the mapping between reference space and world space can be deduced solely based upon the vertex positions. Furthermore in these cases, as denoted in Figure 6.1, the form of the mapping function is solely determined by type (shape) of the element. If only planar-sided elements are used, pre-computation involving the mapping functions can be done so that when vertex value information is available, all the data structures can be finalized.

As presented in **??**, there are many components of *Nektar++* that capitalize on the geometric nature of the basis functions we use. We often speak in terms of vertex modes, edge modes, face modes and internal modes – i.e., the coefficients that provide the weighting of vertex basis functions, edge basis functions, etc. It is beyond the scope of this developer guide to go into all the mathematical details of their definitions, etc. However, we do want to point out a few common developer-level features that are important. In the case of straight-sided (planar-sided) elements, the aforementioned mapping functions can be fully described by vertex basis functions. The real benefit of this approach (of connecting the mapping representation with a geometric basis) is seen when moving to curved elements.

Consider Figure 6.2 in which we modify the example given above to accommodate on curved edge. From the mathematical perspective, we know that the inclusion of this (quadratic) edge will require our mapping function to now be in $Q(2)$. If we were not to use the fact that our basis is geometric in nature, we would be forced to form a Vandermonde system for a set of coefficients used to combine the tensor-product quadratic functions (nine basis functions in all), and use the five pieces of information available to us (the four vertex values and the one point $\vec{c}_0$ that informs the curve on edge 1. As shown in Figure 6.2, we would expect that this updated (to accommodate a curved edge) mapping function to consist of the bi-linear mapping function with an additional term $C_1(\xi_1, \xi_2)$ that encompasses the new curvature information.

The basis we use, following [44], allows us to precisely specify $C_1(\xi_1, \xi_2)$ using the edge basis function associated with edge 1, and to use the point value $\vec{c}_0$ to specify the coefficient to be used. In the figure, we assume that the form of the function is collocating, but in practice it need not be so.

In practice, edge (and face) information can be given either as a set of point positions in world space that correspond to a particular point distribution in the reference element (i.e., evenly-spaced points or GLL points) or modal information corresponding to the geometric basis we use internally. Our geometric file formats assume the former – that curve information is provided to us as physical values at specified positions from which

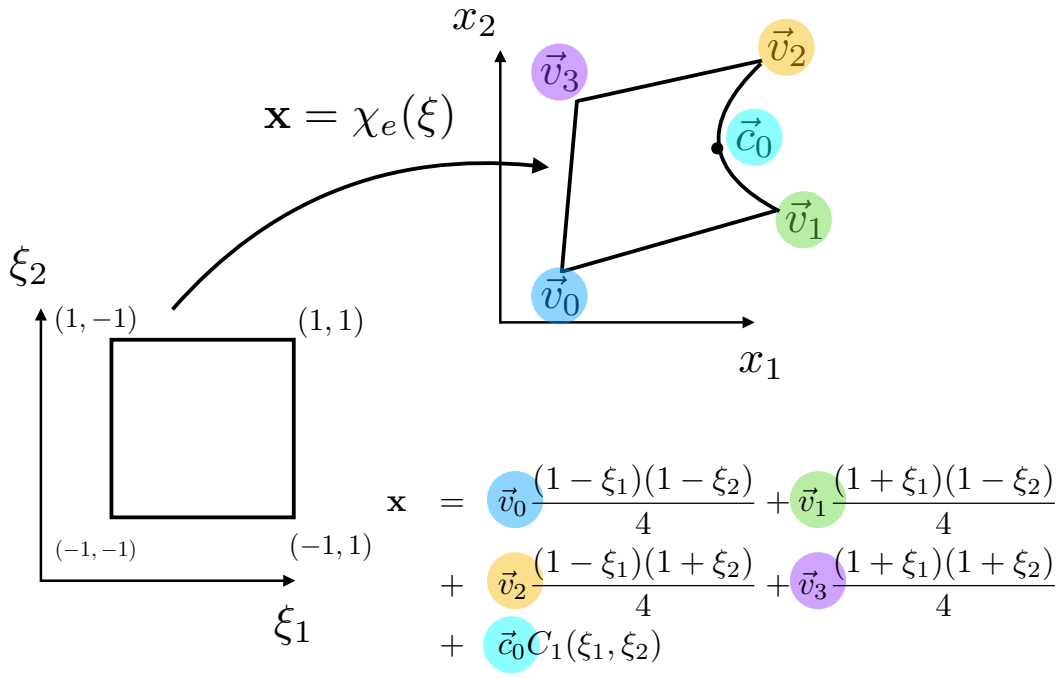


Figure 6.2: Reference space to world space mapping of a 2D quadrilateral to a curve (2D) quadrilateral in the plane via a bi-quadratic (i.e., $Q(2)$) mapping.

we infer (calculate) the modal values and store these values within SpatialDomain data structures.

Assuming we now have, for each element, a way of specifying the mapping function $\chi(\vec{\xi})$, we can now move to how we compute the geometric factors for regular as well as for deformed mappings. We use the term “regular mappings" to refer to mappings from reference space elements to world space elements with the same dimension. For instance, a collection of triangles that lie in a plane can be represented by vertex coordinates that an only 2-tuples and the mappings between the reference space triangle the and world space triangle does not need to consider a larger embedded dimension. This was a fundamental assumption of the original Nektar code: segments lived on the 1D line, triangles and quadrilaterals lived on the 2D plane and that hexahedra, tetrahedra, etc., lived in the 3D volume. When redesigning *Nektar++*, we purposefully enabled geometric entities to live in a high-dimensional embedding space, different from their parameterized dimension. For instance, in *Nektar++*, it is possible to define segment expansions (functions that live over one dimensional parameterized curves) that are embedded in 3D. The same is true for quadrilaterals and triangles – although their parameterized dimension is two, both may live in a higher dimensional embedding space and thus represent a *manifold* of co-dimension one in that space. For mappings that maintain the co-dimension is the opposite of the dimension (i.e., quadrilateral in a plane represented by vertices with two coordinates mapped to a two parameter reference quadrilateral), we keep to the mapping

conventions originally outlined in [44] and denote these mapping operations in the code by the enumerated value *Regular*; for mappings in which the co-dimension is greater than zero, we following the modified convention outlined in [18] and denote these mapping operations in the code by the enumerated value *Deformed*.

6.1.2 Regular Mappings: Geometric Factors

Following [44], we assume that the vertex positions of an element in world space are given by $\vec{x} = x_i$ and that our reference space coordinates are given by $\vec{\xi} = \xi_j$, there i and j run from zero to $dim - 1$ where dim is the parameter dimension of the element (e.g. for a triangle, $dim = 2$).

The Jacobian (matrix) is correspondingly defined as:

$$\mathbf{J} = \frac{\partial x_i}{\partial \xi_j}. \tag{6.1}$$

Note that this matrix is always square, but also note that it is not always constant across an element. Only in special cases such as the linear mapping of triangles and tetrahedra does the Jacobian matrix reduce to a constant (matrix) over the entire element. In general, this matrix can be evaluated at any point over the element for which it is constructed.

There are two (high-level) times in which this information is needed: when computing derivatives and when computing integrals. When computing derivatives, we employ the chain rule for differentiation, which in Einstein notation is given by the following expression:

$$\frac{\partial}{\partial x_i} = \frac{\partial \xi_j}{\partial x_i}\frac{\partial}{\partial \xi_j}.$$

Note that this expression requires the reciprocal of the expression above – that is, $\mathbf{J}^{-1}$. The polynomial mappings we use in *Nektar++* are defined in terms of their forward mappings (reference to world). If the determinant of the mapping is non-zero, the inverse of the mapping exists but is not available analytically (except is special cases). As a consequence, we in general limit the places at which we compute the inverse of the Jacobian. Typically, the quadrature point positions are the places at which you need these values (since it as at these points we take physical space derivates and then use integration rules to construct weak form operators). Thus, procedurally, we do the following:

1. For a given element, compute the Jacobian matrix using the expression given in Equation 6.2 for each quadrature point position on the element (or for any points positions within an element at which it is needed).

2. Explicitly form the inverse of the matrix at each point position.

Because we accomplish the inversion of the Jacobian matrix at particular positions, this introduces an approximation to this computation. Although the inverse Jacobian matrix can be computed exactly at each point – when we then correspondingly use this information in the inner product over an element – we are in effect assuming that we are using a polynomial interpolative projection of this operator. The approximation error is introduced at the point of our quadrature approximation. Although the forward mapping is polynomial and hence we could find a polynomial integration rule and number of points/weights to integrate our bilinear forms exactly, our use of the inverse mapping in our bilinear forms means that we can only approximate our integrals. From our experiments, the impact of his is negligible in most cases, and only becomes in a concern in highly curved geometries. In such cases, over-integration might be requires to minimize the errors introduced due this approximation.

The other place at which we need the Jacobian matrix is to compute its determinant to be used in integration. The determinant of the Jacobian matrix (sometimes also called “the Jacobian" of the mapping) provides us the scaling of the metric terms used in integration. In all our computations, we assume that the determinant of the Jacobian matrix is strictly positive. In the area of mesh generation, the value of the determinant is used to estimate how good or bad the quality of the mapping is – in effect, if you have reasonable elements in your mesh. Negative Jacobian elements are inadmissible but even elements with small Jacobian determinants might cause issues. At the level of the library and solvers, we assume that these issues have been addressed by the user prior to attempting to run *Nektar++* solvers and interpret their results.

6.1.3 Deformed Mappings: Geometric Factors

Following [18], we again assume that the vertex positions of an element in world space are given by $\vec{x} = x_i$ and that our reference space coordinates are given by $\vec{\xi} = \xi_j$, there i runs from zero to $dim - 1$ where dim is the embedding dimension and j runs from zero to $M - 1$ where M is the parameter dimension of the element (e.g.for a triangle on a manifold embedded in 3D with vertex values in 3D, $dim = 3$ whilst $M = 2$).

The Jacobian (matrix) is correspondingly defined as:

$$\mathbf{J} = J^i_j = \frac{\partial x_i}{\partial \xi_j}. \tag{6.2}$$

In this case, we use the notational conventions of [8] which delineate covariant and contravariant forms. In general, this matrix is not square, also also note that it is not always constant across an element. In general, this matrix can be evaluated at any point over the element for which it is constructed. The metric tensor related to this transformation can be formed as:

$$\mathbf{g} = \mathbf{J}\mathbf{J}^T$$

and the Jacobian factor associated with this mapping is then given by:

$$g = \det \mathbf{g}.$$

Because various mappings are necessary when dealing with covariant and contravariant vectors, we have encapsulated all these routines into the directory GlobalMapping (see Chapter 10). At this stage, we do not implement general Piola transformations [50] that further respect $H(div)$ and $H(curl)$ constraints on these mappings as would be needed in solid mechanics or electromagnetics; however, there is nothing inherent within the *Nektar++* framework that would preclude someone from adding these additional features as necessary.

6.2 The Fundamental Data Structures within SpatialDomains

As mentioned earlier, in almost all object-oriented languages (which includes $C++$), there exists the concepts of *class attributes* and *object attributes*. For a summary of attributes and access patterns, please review Section 5.2. Within the SpatialDomains directory of the library, there exists a class inheritance hierarchy designed to try to encourage re-use of core algorithms (while simultaneously trying to minimize duplication of code). We present this class hierarchy in Figure 6.3.

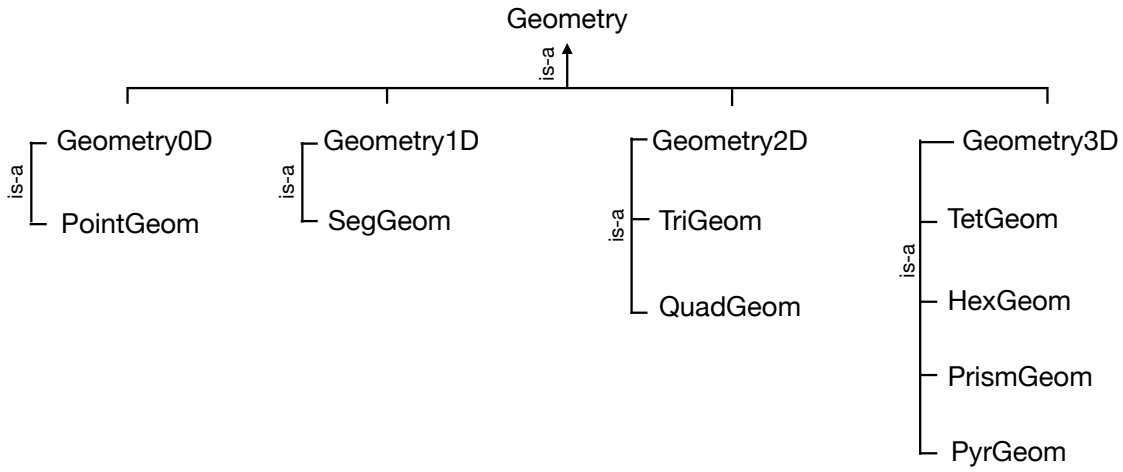


Figure 6.3: Class hierarchy derived from Geometry, the base class of the SpatialDomains Directory.

At its core, the items contained within SpatialDomains are meant to represent the mapping of StdRegion information into world-space. The various attributes contained

herein related to this geometric (mesh, curvature and mapping) information. The various private, protected and public data members contained within StdRegions are provided in the subsequent sections.

6.2.1 Variables at the Level of Geometry

Private:

Protected:

Public:

6.2.2 Variables at the Level of Geometry$D for various Dimensions

Private:

Protected:

Public:

6.2.3 Variables at the Level of Shape-Specific Geometry Information

Private:

Protected:

Public:

6.2.4 Reference to World-Space Mapping

Geometry

GeomFactors

6.2.5 MeshGraph

6.3 The Fundamental Algorithms within SpatialDomains

As stated in the introduction, this section of this guide is structured in question-answer form. This is not meant to capture every possible question asked of us on the *Nektar++* users list; however, this set of (ever-growing) questions are meant to capture the "big ideas" that developers want to know about how SpatialDomains work and how they can be used.

In this section, we will through question and answer format try to cover the following basic algorithmic concepts that are found within the SpatialDomains part of the library:

- xx

With the big ideas in place, let us now start into our questions.

Question:

Chapter 7

Inside the Library: LocalRegions

In this chapter, we walk the reader through the different components of the LocalRegions Directory. We begin with a discussion of the mathematical fundamentals, for which we use the book by Karniadakis and Sherwin [44] as our principle reference. We then provide the reader with an overview of the primary data structures introduced within the LocalRegions Directory (often done through C++ objects), and then present the major algorithms – expressed as either object methods or functions – employed over these data structures.

7.1 The Fundamentals Behind LocalRegions

The idea behind local regions is strongly connected to that of standard regions, but from the top-down perspective. As an example: in standard regions, we only had to consider one type of triangle, the one that is straight-sided, right-angled, and whose principle horizontal and vertical sides where aligned with the coordinate axes. Of course, meshes of elements consist of elements that may are may not be right-angled, planar-sided, etc. The starting point for us is the question of how to build build basis functions that exist of a *world-space* element – that is, an element whose vertex positions lie in the engineering (PDE) coordinate system of interest. Such an expansion is a local region. In *Nektar++*, each local region *is-a* standard region and *has-a* spatial domain data structure. The local region inherits common expansion methods from its standard region parent, and it uses its spatial domain information to specialize its operators to its local coordinate system.

7.2 The Fundamental Data Structures within LocalRegions

As mentioned earlier, in almost all object-oriented languages (which includes $C++$), there exists the concepts of *class attributes* and *object attributes*. For a summary of attributes and access patterns, please review Section 5.2. Within the LocalRegions directory of the library, there exists a class inheritance hierarchy designed to try to encourage re-use of core algorithms (while simultaneously trying to minimize duplication of code). We present this class hierarchy in Figure 7.1.

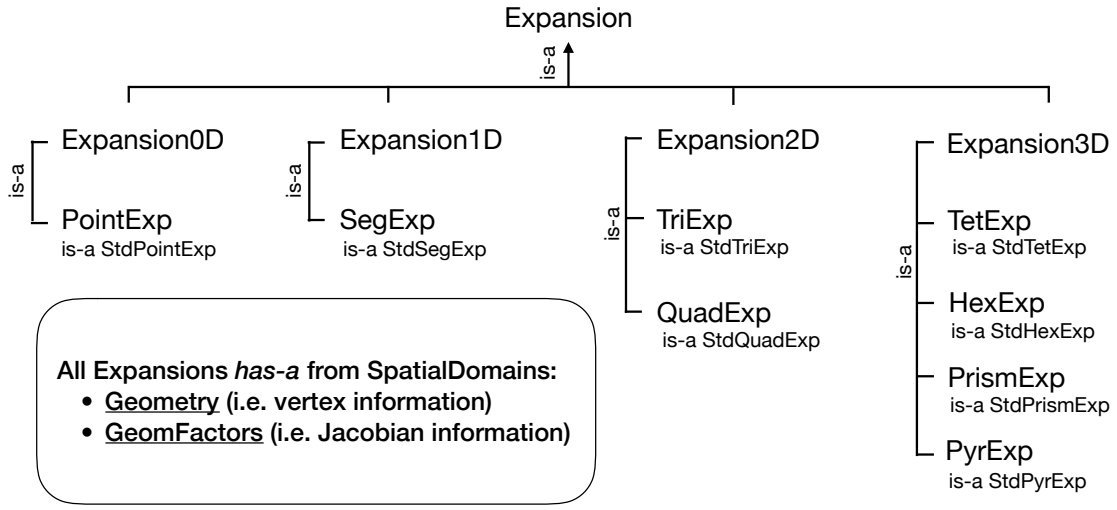


Figure 7.1: Class hierarchy derived from Expansion, the base class of the LocalRegions Directory.

As is seen in Figure **??**, the LocalRegions hierarchy consists of three levels: the base level from which all LocalRegion objects are derived is Expansion. This object is then specialized by dimension, yielding Expansion0D, Expansion1D, Expansion2D and Expansion3D. The dimension-specific objects are then specialized based upon shape.

The object attributes (variables) at various levels of the hierarchy can be understood in light of Figure 5.6. At its core, an expansion is a means of representing a function over a world-space region evaluated at a collection of point positions. The various data members hold information to allow all these basic building blocks to be specified. Many of the attributes are inherited from StdRegions as they are not unique to LocalRegions; however, each LocalRegion Expansion is uniquely defined based upon its geometric factors (which it stores via SpatialDomain information).

The various private, protected and public data members contained within LocalRegions are provided in the subsequent sections.

7.2.1 Variables at the Level of Expansion

Private: There are private methods but no private data members within Expansion.

Protected: As discussed above, the primary data in LocalRegions that distinguishes it from StdExpansions is the *has-a* relationship with SpatialDomains, given by the following:

- SpatialDomains::GeometrySharedPtr `m_geom`
- SpatialDomains::GeomFactorsSharedPtr `m_metricinfo`

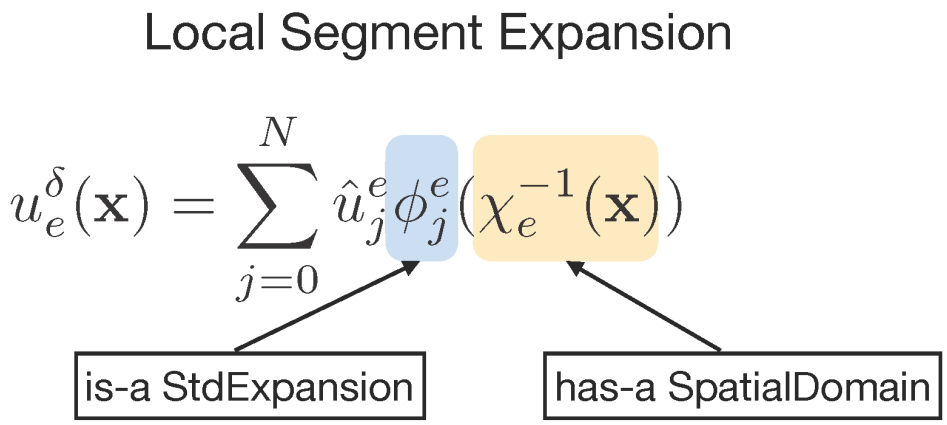


Figure 7.2: Diagram to help understand the various data members (object attributes) contained within LocalRegions and how they connect with the mathematical representation presented earlier. Recall that a LocalRegion *is-a* StdRegion and *has-a* SpatialDomain.

- MetricMap `m_metrics`

Public: There are public methods but no public data members within Expansion.

7.2.2 Variables at the Level of Expansion$D for various Dimensions

Private:

Protected:

Public:

7.2.3 Variables at the Level of Shape-Specific Expansions

Private:

Protected:

Public:

7.3 The Fundamental Algorithms within LocalRegions

As stated in the introduction, this section of this guide is structured in question-answer form. This is not meant to capture every possible question asked of us on the *Nektar++*

users list; however, this set of (ever-growing) questions are meant to capture the "big ideas" that developers want to know about how LocalRegions work and how they can be used.

In this section, we will through question and answer format try to cover the following basic algorithmic concepts that are found within the LocalRegions part of the library:

- xx

With the big ideas in place, let us now start into our questions.

Question:

Chapter 8

Inside the Library: Collections

In this chapter, we walk the reader through the different components of the Collections Directory. We begin with a discussion of the mathematical fundamentals, for which we use the book by Karniadakis and Sherwin [44] as our principle reference. We then provide the reader with an overview of the primary data structures introduced within the Collections Directory (often done through C++ objects), and then present the major algorithms – expressed as either object methods or functions – employed over these data structures.

8.1 The Fundamentals Behind Collections

The concept of "collections" is that there are many operations that we (conceptually) accomplish on an element-by-element basis that can be aggregated together into a single operation. The basic idea is that whenever you see a piece of code that consists of a loop in which, in the body of the loop, you are accomplishing the same operation (e.g. function evaluation) based upon an input variable tied to the loop index – you may be able to make the operation a "collection". For instance, consider if you were given the following snippet of code in Matlab notation:

```
do j = 1:N,
     y(j) = dot(a,x(:,j))
end
```

where y is a row vector of length N, x is an $M \times N$ matrix and a is a row vector of length M. Note that in this code snipped, the vector a remains constant; only the vector x involved in the dot product is changing based upon the index (i.e. selecting a column of the array). From linear algebra, you might recognize this as the way we accomplish the product of a row vector with a matrix – we march through the matrix accomplishing dot products. Although these two things are equivalent mathematically, they are not equivalent computationally *from the perspective of computational efficiency.* Most linear algebra operations within *Nektar++* are done using BLAS – Basic Linear

Algebra Subroutines – a collection of specialized routines for accomplishing Level 1 (single loop), Level 2 (double nested loop) and Level 3 (triple nested loop) operations. A dot product is an example of a Level 1 BLAS operation; A matrix-vector multiplication is an example of a Level 2 BLAS operation; and finally a matrix-matrix multiplication is an example of a Level 3 BLAS operation. The general rule of thumb is that as you move up the levels, the operations can be made more efficient (using various algorithmic reorganization and memory layout strategies). Hence when you see a loop over Level 1 BLAS calls (like in the above piece of code), you should always ask yourself if this could be transformed into a Level 2 BLAS call. Since the vector a is not changing, this piece of code should be replaced with an appropriate vector-matrix multiply.

As seen in StdRegions and LocalRegions, we often conceptually thing of many of our basic building block operations as being elemental. We have elemental basis matrices, local stiffness matrices, etc. Although it is correct to implement operations as an iterator over elements in which you accomplish an action like evaluation (done by taking a basis evaluation matrix times a coefficient vector), this operation can be made more efficient by ganging all these elemental operations together into a *collection*.

As mentioned earlier, one of the caveats of this approach is that you must assume some level of consistency of the operation. For instance, in the case of physical evaluations, you must assume that a collection consists of elements that are all of the same time, have the same basis number and ordering, and are evaluated at the same set of points – otherwise the operation cannot be expressed as a single (basis) matrix multiplied by a matrix whose columns consist of the modes of all the elements who have joined the collective operation.

As will be seen in the data structure and algorithms sections of this discussion, these assumptions lead us to two fundamental types of collections: collections that live at the StdRegions level (i.e. collection operations that act on a group of elements as represented in reference space) and collections that live at the LocalRegions level (i.e. collection operations that act on a group of elements as represented in world space).

8.2 The Fundamental Data Structures within Collections

8.3 The Fundamental Algorithms within Collections

As stated in the introduction, this section of this guide is structured in question-answer form. This is not meant to capture every possible question asked of us on the *Nektar++* users list; however, this set of (ever-growing) questions are meant to capture the "big ideas" that developers want to know about how Collections work and how they can be used.

In this section, we will through question and answer format try to cover the following basic algorithmic concepts that are found within the Collections part of the library:

- xx

With the big ideas in place, let us now start into our questions.

Question:

Chapter 9

Inside the Library: MultiRegions

In this chapter, we walk the reader through the different components of the MultiRegions Directory. We begin with a discussion of the mathematical fundamentals, for which we use the book by Karniadakis and Sherwin [44] as our principle reference. We then provide the reader with an overview of the primary data structures introduced within the MultiRegions Directory (often done through C++ objects), and then present the major algorithms – expressed as either object methods or functions – employed over these data structures.

9.1 The Fundamentals Behind MultiRegions

Up to now in the library, the various data structures and methods associated with standard regions, spatial domains and local regions are not specifically dictated by any particular numerical method. In fact, at this stage, they can all be viewed in light of approximation theory. With local regions in place, we have a region in world space over which we can represent an expansion (i.e. linear combination of basis functions) and form its derivatives and its integral. It is at the level of MultiRegions that we now combine two fundamental concepts: the idea of a collection of elements together to form a "global" expansion and the idea of how these (local) elements communicate (in the sense of how does one form approximations of a PDE of these collections of elements). Hence MultiRegions is important because it gives us a way of dealing with general tessellation and also because it is the first place at which we can connect to a specific numerical PDE approximation methods of choice (i.e. continuous Galerkin FEM methods, discontinuous Galerkin finite volume methods, etc.).

Because MultiRegions is both about grouping of elements in space (to form a domain) and about solving PDEs over these domains, you will find two primary collections of routines contained within the MultiRegions directory. You will find things related to collections of elements: ExpList (Expansion List), DisContField (discontinuous field) and ConField (continuous field); and you will find objects related to the linear systems formed based upon the particular numerical method once selects (i.e. GlobalLinSys,

which stands for Global Linear System).

At present, the *Nektar++* framework supports three types of numerical PDE discretizations for conservation laws:

- **Discontinuous Galerkin Methods:** These weak-form (variational) methods do not require element continuity, but do put restrictions on the flux of information between elements. In general, these methods can be thought of as being in the class of finite volume (FV) methods. One feature of these methods that is often exploited computationally is that many operations can be considered as elemental. See [20, 39] and references therein for a more complete summary.

- **Continuous Galkerin Methods:** These weak-form (variational) methods require at least C^0 continuity. Mathematically, there have been extensions to higher levels of continuity, e.g. Isogeometric Analysis [21], these are not implemented in *Nektar++* and would require further constraints on our SpatialDomain representations than we currently accommodate. In general, these methods can be thought of as being in the class of finite element methods (FEM). Although these methods are technically (mathematically) formulated as global methods, their elemental construction and compact basis types allow for many local operations. Many (but not all) of the linear system routines that are contained with the MultiRegions directory are focussed on this discretization type. See [54, 25, 44] and references therein for a more complete summary.

- **Flux Reconstruction Methods:** These strong-form methods do not require element continuity, but like dG methods they impose restrictions on the flux of information between elements. In general, these methods can be though of as being in the class of generalized finite difference (FD) or collocating methods. See [46, 63] and references therein for a more complete summary.

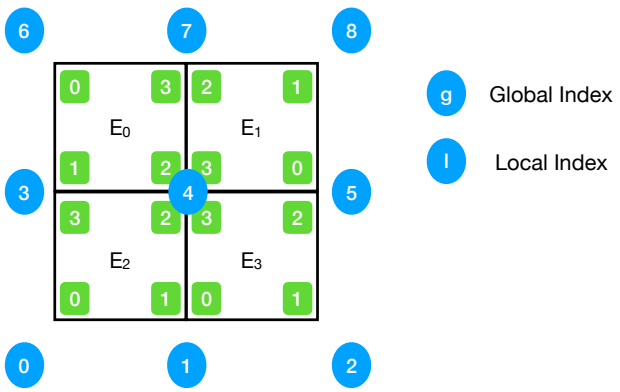


Figure 9.1: Diagram to help explain assembly. UPDATE.

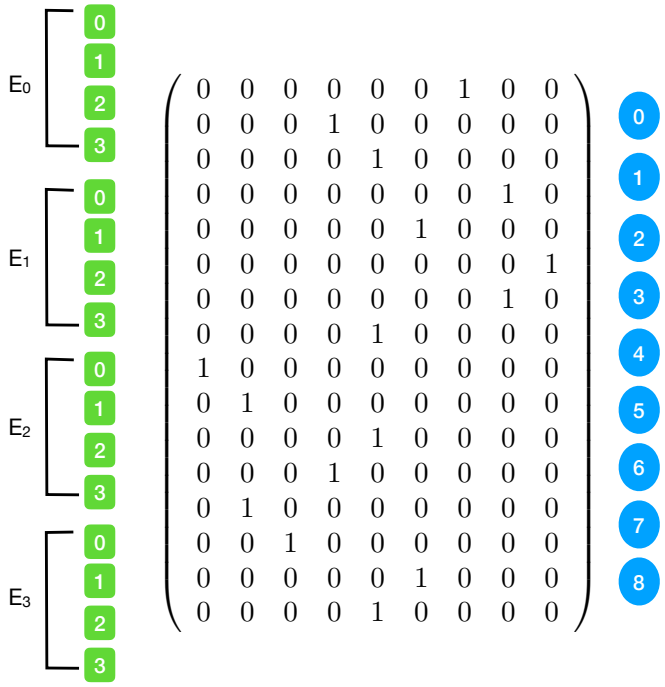


Figure 9.2: Diagram to help explain assembly. UPDATE.

Assembly Map

9.2 The Fundamental Data Structures within MultiRegions

As mentioned earlier, in almost all object-oriented languages (which includes $C++$), there exists the concepts of *class attributes* and *object attributes*. For a summary of attributes and access patterns, please review Section 5.2. Within the MultiRegions directory of the library, there exists a class inheritance hierarchy designed to try to encourage re-use of core algorithms (while simultaneously trying to minimize duplication of code). We present this class hierarchy in Figure 9.3.

At its core, the items contained within MultiRegions are meant to represent sets of LocalRegions. In the most abstract sense, an ExpList is merely a set of LocalRegion objects that may or may not have any relevance to each other. We then, as we do in other parts of the library, specialize on the dimension of the objects these sets will contain. At the subsequent levels of the hierarchy, we now involve information about we want to treat a collection of elements when evaluating them as a *field*. We consider a *field* to be the representation of a function over a (sub–)domain. A domain consists of a collection of elements that are *connected* (that is, they form a contiguous region in space). We think of the region of space as being tiled by elements over which expansions are build. If we consider a MultiRegion field to be a collection of these expansions with no constraints on their continuity, we arrive at the DisContField family of class definitions (which vary by

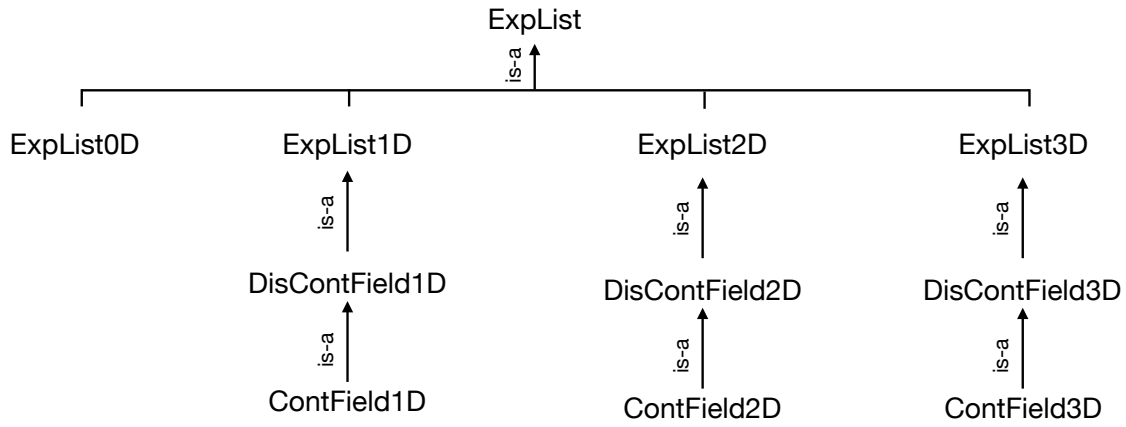


Figure 9.3: Class hierarchy derived from ExpList, the base class of the MultiRegions Directory.

dimension). For the currently available solvers within *Nektar++*, this level of field is used for the solution of PDEs via the discontinuous Galerkin (dG) and Flux-Reconstruction (FR) methods. If we consider a function as a collection of expansion in which we require continuity (in *Nektar++*, only C^0 continuity), then we employ a further derived class called ContField (which again varies by dimension). For the currently available solvers within *Nektar++*, this level of field is used for the solution of PDEs via the continuous Galerkin (cG) method. Since many of the operations at the continuous field level do not rely upon the continuity of the field, we have structured the continuous MultiRegion object as with an *is-a* relationship with the DisContFields.

The various private, protected and public data members contained within MultiRegions are provided in the subsequent sections.

9.2.1 Variables at the Level of ExpList

Private:

Protected:

Public:

9.2.2 Variables at the Level of ExpList$D for various Dimensions

Private:

Protected:

Public:

9.2.3 Variables at the Level of Discontinuous Field Expansions

Private:

Protected:

Public:

9.2.4 Variables at the Level of Continuous Field Expansions

Private:

Protected:

9.3 The Fundamental Algorithms within MultiRegions

As stated in the introduction, this section of this guide is structured in question-answer form. This is not meant to capture every possible question asked of us on the *Nektar++* users list; however, this set of (ever-growing) questions are meant to capture the "big ideas" that developers want to know about how MultiRegions work and how they can be used.

In this section, we will through question and answer format try to cover the following basic algorithmic concepts that are found within the MultiRegions part of the library:

- xx

With the big ideas in place, let us now start into our questions.

Question:

9.4 Preconditioners

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

$$\nabla^2 u(\mathbf{x}) + \lambda u(\mathbf{x}) = f(\mathbf{x}), \tag{9.1}$$

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.

9.4.1 Mathematical formulation

The standard spectral/*hp* approach to discretise (9.1) starts with an expansion in terms of the elemental modes:

$$u^{\delta}(\mathbf{x}) = \sum_{n=0}^{N_{\text{dof}}-1} \hat{u}_n \Phi_n(\mathbf{x}) = \sum_{e=1}^{N_{\text{el}}} \sum_{n=0}^{N_m^e-1} \hat{u}_n^e \phi_n^e(\mathbf{x}) \tag{9.2}$$

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 (9.2), we adopt a Galerkin discretisation of equation (9.1) 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.

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:

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

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.

9.4.2 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.

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 $\mathbf{M}^{-1} = (h_{ii}^{-1})$.

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.

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 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 $\mathbf{M}^{-1}$.

9.4.3 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:

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

where $\mathrm{diag}[(\mathbf{S}_2)_{vv}]$ 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 10

Inside the Library: GlobalMapping

In this chapter, we walk the reader through the different components of the GlobalMapping Directory. We begin with a discussion of the mathematical fundamentals, for which we use research article by Cantwell et al. [18] and the book [8] as our principle references. We then provide the reader with an overview of the primary data structures introduced within the GlobalMapping Directory (often done through C++ objects), and then present the major algorithms – expressed as either object methods or functions – employed over these data structures.

10.1 The Fundamentals Behind GlobalMapping

Based upon the appendix in [18], we outline a rigorous derivation of the Laplace-Beltrami operator. We use the convention that indices appearing once in the upper position and once in the lower position are considered dummy indices and are implicitly summed over their range, while non-repeated indices are considered free to take any value. Derivatives are also denoted using the lower-index comma notation, for example $g_{ij,k}$. *Covariant* vectors such as the gradient are those which, under a change of coordinate system, change under the same transformation in order to maintain coordinate system invariance. In contrast, *contravariant* vectors, such as velocity, should remain fixed under a coordinate transformation requiring that their components change under the inverse of the transformation to maintain invariance. With this in mind, we now construct the fundamental differential operators we require for a 2-dimensional manifold embedded in a 3-dimensional space. In order to express these operators in curvilinear coordinates we start by assuming that we have a smooth surface parametrization given by

$$\mathbf{x}(\xi^1, \xi^2) := (x^1(\xi^1, \xi^2), x^2(\xi^1, \xi^2), x^3(\xi^1, \xi^2)).$$

Next we will define the Jacobian of $\mathbf{x}$ as the tensor

$$J_i^j = \frac{\partial x^j}{\partial \xi^i}$$

where J_i^j can be viewed as a covariant surface vector (by fixing the upper index) or as a contravariant space vector (by fixing the lower index). The surface metric tensor g_{ij} can be defined in terms of the J_i^j as

$$g_{ij} = \sum_{k=1}^{3} J_i^k J_j^k. \tag{10.1}$$

which can be considered to transform a contravariant quantity to a covariant quantity. Similarly the conjugate tensor g^{ij}, which does the reverse transformation, is given by

$$g^{11} = g_{22}/g, \quad g^{12} = g^{21} = -g_{12}/g, \quad g^{22} = g_{11}/g, \tag{10.2}$$

where g is the determinant of g_{ij}. The metric tensor and its conjugate satisfy the condition

$$\delta_i^j = g_{ik}g^{jk} = \begin{cases} 1, & \text{if } i = j, \\ 0, & \text{if } i \neq j. \end{cases}$$

To construct the divergence operator we will also need the derivative of g with respect to components of the metric, g_{ij}. We know that $\mathbf{g}$ is invertible and from linear algebra we have that the inverse of the metric (10.2) satisfies $\mathbf{g}^{-1} = \frac{1}{g}\tilde{\mathbf{g}}^\top$, where $\tilde{\mathbf{g}}$ is the cofactor matrix of $\mathbf{g}$. Therefore $\tilde{\mathbf{g}}^\top = g\boldsymbol{g}^{-1}$, or in components $\tilde{g}^{ij} = g(\boldsymbol{g}^{-1})_{ji}$. Using Jacobi's formula for the derivative of a matrix determinant with respect its entries, and since $\mathbf{g}$ is invertible, the derivative of the metric determinant is

$$\frac{\partial g}{\partial g_{ij}} = \operatorname{tr}\left(\tilde{\mathbf{g}}^\top \frac{\partial \mathbf{g}}{\partial g_{ij}}\right) = \tilde{g}^{ij} = g(\boldsymbol{g}^{-1})_{ji} = gg^{ij}. \tag{10.3}$$

10.1.1 Divergence operator

The partial derivative of a tensor with respect to a manifold coordinate system is itself not a tensor. In order to obtain a tensor, one has to use *covariant* derivative, defined below. The covariant derivative of a contravariant vector is given by

$$\nabla_k a^i = a^i_{,k} + a^j \Gamma^i_{jk}. \tag{10.4}$$

where Γ^i_{jk} are *Christoffel Symbols of the second kind.* The *Christoffel symbols of the first kind* are defined by

$$\Gamma_{ijk} = \frac{1}{2}\left[g_{kj,i} + g_{ik,j} - g_{ij,k}\right].$$

Here we note that Γ_{ijk} is symmetric in the first two indices. To obtain the Christoffel symbols of the second kind we formally raise the last index using the conjugate tensor,

$$\Gamma^l_{ij} = \Gamma_{ijk}g^{kl} \tag{10.5}$$

which retains the symmetry in the lower two indices. We can now express the derivatives of the metric tensor in terms of the Christoffel symbols as

$$g_{ij,k} = \Gamma_{ikj} + \Gamma_{jki} = g_{lj}\Gamma^l_{ik} + g_{li}\Gamma^l_{jk}.$$

We now define the divergence operator on the manifold, $\nabla \cdot \mathbf{X} = \nabla_k X^k$. Consider first the derivative of the determinant of the metric tensor g with respect to the components of some local coordinates system ξ^1, ξ^2. We apply the chain rule, making use of the derivative of the metric tensor with respect to components of the metric (10.3) and the relationship (10.5), to get

$$\frac{\partial g}{\partial \xi^k} = \frac{\partial g}{\partial g_{ij}}\frac{\partial g_{ij}}{\partial \xi^k} = g g^{ij} g_{ij,k} = g g^{ij}(\Gamma_{ikj} + \Gamma_{jki}) = g(\Gamma^i_{ik} + \Gamma^j_{jk}) = 2g\Gamma^i_{ik}. \tag{10.6}$$

We can therefore express the Christoffel symbol Γ^i_{ik} in terms of this derivative as

$$\Gamma^i_{ik} = \frac{1}{2g}\frac{\partial g}{\partial \xi^k} = \frac{1}{\sqrt{g}}\frac{\partial \sqrt{g}}{\partial \xi^k}. \tag{10.7}$$

Finally, by substituting for Γ^i_{ik} in the expression for the divergence operator

$$\begin{aligned}\nabla_k X^k &= X^k_{,k} + X^i \Gamma^k_{ki} \\ &= X^k_{,k} + X^k \Gamma^i_{ik} \\ &= X^k_{,k} + X^k \frac{1}{\sqrt{g}}(\sqrt{g})_{,k}\end{aligned}$$

we can deduce a formula for divergence of a contravariant vector as

$$\nabla \cdot \mathbf{X} = \nabla_k X^k = \frac{\left(X^k \sqrt{g}\right)_{,k}}{\sqrt{g}} \tag{10.8}$$

10.1.2 Laplacian operator

The covariant derivative (gradient) of a scalar on the manifold is identical to the partial derivative, $\nabla_k \phi = \phi_{,k}$. To derive the Laplacian operator we need the contravariant form of the covariant gradient above which can be found by raising the index using the metric tensor, giving

$$\nabla^k \phi = g^{kj}\phi_{,j}, \tag{10.9}$$

and substituting (10.9) for X^k in (10.8) to get the Laplacian operator on the manifold as

$$\Delta_M \phi = \frac{1}{\sqrt{g}}\left(\sqrt{g} g^{ij}\phi_{,j}\right)_{,i}. \tag{10.10}$$

10.1.3 Anisotropic diffusion

We now extend the above operator to allow for anisotropic diffusion in the domain by deriving an expression for the surface conductivity from the ambient conductivity. The gradient of a surface function scaled by the ambient conductivity tensor $\tilde{\nabla}^p$ is given by

$$\tilde{\nabla}^p f = g^{mp} J^l_m \sigma_{kl} J^k_j g^{ij} \frac{\partial f}{\partial x^i}. \tag{10.11}$$

The surface gradient is mapped to the ambient space through the Jacobian J^k_j, scaled by the ambient conductivity, and mapped back to the surface through J^l_m. The anisotropic Laplacian operator is given by

$$\tilde{\nabla}^2 f = \nabla_k \tilde{\nabla}^k f = \nabla_k \tilde{\sigma}^{ij} \nabla_j f \tag{10.12}$$

Therefore the surface conductivity tensor can be computed using the Jacobian tensor and the inverse metric as

$$\tilde{\sigma} = \mathbf{g^{-1}} \mathbf{J} \sigma \mathbf{J}^\top \mathbf{g^{-1}}. \tag{10.13}$$

10.1.4 Anisotropic Laplacian operator

Anisotropic diffusion is important in many applications. In the ambient Euclidean space, this can be represented by a diffusivity tensor σ in the Laplacian operator as

$$\Delta_M = \nabla \cdot \sigma \nabla.$$

On our manifold, we seek the generalisation of 10.10, in the form

$$\tilde{\Delta}_M \phi = \nabla_j \tilde{\sigma}^j_i \nabla^i \phi.$$

where the $\tilde{\sigma}^j_i$ are entries in the surface diffusivity. For a contravariant surface vector a^j we can find the associated space vector A^i as $A^i = J^i_j a^j$. Similarly if A_i is a covariant space vector, then $a_j = J^i_j A_i$ is a covariant surface vector. Using these we can construct the anisotropic diffusivity tensor $\tilde{\sigma}$ on the manifold by constraining the ambient diffusivity tensor σ to the surface. The contravariant surface gradient $\nabla^i \phi$ is mapped to the corresponding space vector, which lies in the tangent plane to the surface. This is scaled by the ambient diffusivity and then projected back to a covariant surface vector. Finally, we use the conjugate metric to convert back to a contravariant form. The resulting surface Laplacian is

$$\tilde{\Delta}_M \phi = \nabla_m g^{lm} J^k_l \sigma_{jk} J^j_i \nabla^i \phi.$$

Following on from this we deduce that

$$\tilde{\sigma}^j_i = g^{jm} J^l_m \sigma_{lk} J^k_i.$$

It can be seen that in the case of isotropic diffusion that $\tilde{\sigma}^j_i = \delta^i_j \Leftrightarrow \sigma = \mathbf{I}$,

$$\tilde{\sigma}^j_i = g^{im} J^k_m \sigma_{lk} J^l_j = g^{im} J^k_m J^k_j = g^{im} g_{mj} = \delta^i_j.$$

10.2 The Fundamental Data Structures within GlobalMapping

10.3 The Fundamental Algorithms within GlobalMapping

Chapter 11

Inside the Library: FieldUtils

In this chapter, we walk the reader through the different components of the FieldUtils Directory. We begin with a discussion of the mathematical fundamentals, for which we use the book by Karniadakis and Sherwin [44] as our principle reference. We then provide the reader with an overview of the primary data structures introduced within the FieldUtils Directory (often done through C++ objects), and then present the major algorithms – expressed as either object methods or functions – employed over these data structures.

11.1 The Fundamentals Behind FieldUtils

11.2 The Fundamental Data Structures within FieldUtils

11.3 The Fundamental Algorithms within FieldUtils

Chapter 12

Inside the Library: SolverUtils

In this chapter, we walk the reader through the different components of the SolverUtils Directory. We begin with a discussion of the mathematical fundamentals, for which we use the book by Karniadakis and Sherwin [44] as our principle reference. We then provide the reader with an overview of the primary data structures introduced within the SolverUtils Directory (often done through C++ objects), and then present the major algorithms – expressed as either object methods or functions – employed over these data structures.

12.1 The Fundamentals Behind SolverUtils

12.2 The Fundamental Data Structures within SolverUtils

12.3 The Fundamental Algorithms within SolverUtils

Part II

Solvers

In this part of the developer's guide, we walk you through development aspects of the various solvers that are available within publicly-available *Nektar++*repository. For each of these solvers, user guides have been developed that help you see how to use these for various engineering applications. We encourage you to use these solvers in your science and engineering work flows. If you find that the current solvers do not meet your needs and consequently you end up building a new solver based upon our framework, we encourage you to consider submitting this to us for incorporation in the publicly-available master branch.

Chapter 13

ADRSolver: Solving the Advection-Reaction-Diffusion Equation

In this chapter, we walk the reader through our Advection-Reaction-Diffusion Solver (ADRSolver).

Chapter 14

IncNavierStokesSolver: Solving the Incompressible Navier-Stokes Equations

In this chapter, we walk the reader through our 2D and 3D incompressible Navier-Stokes Solver (IncNavierStokesSolver).

Chapter 15

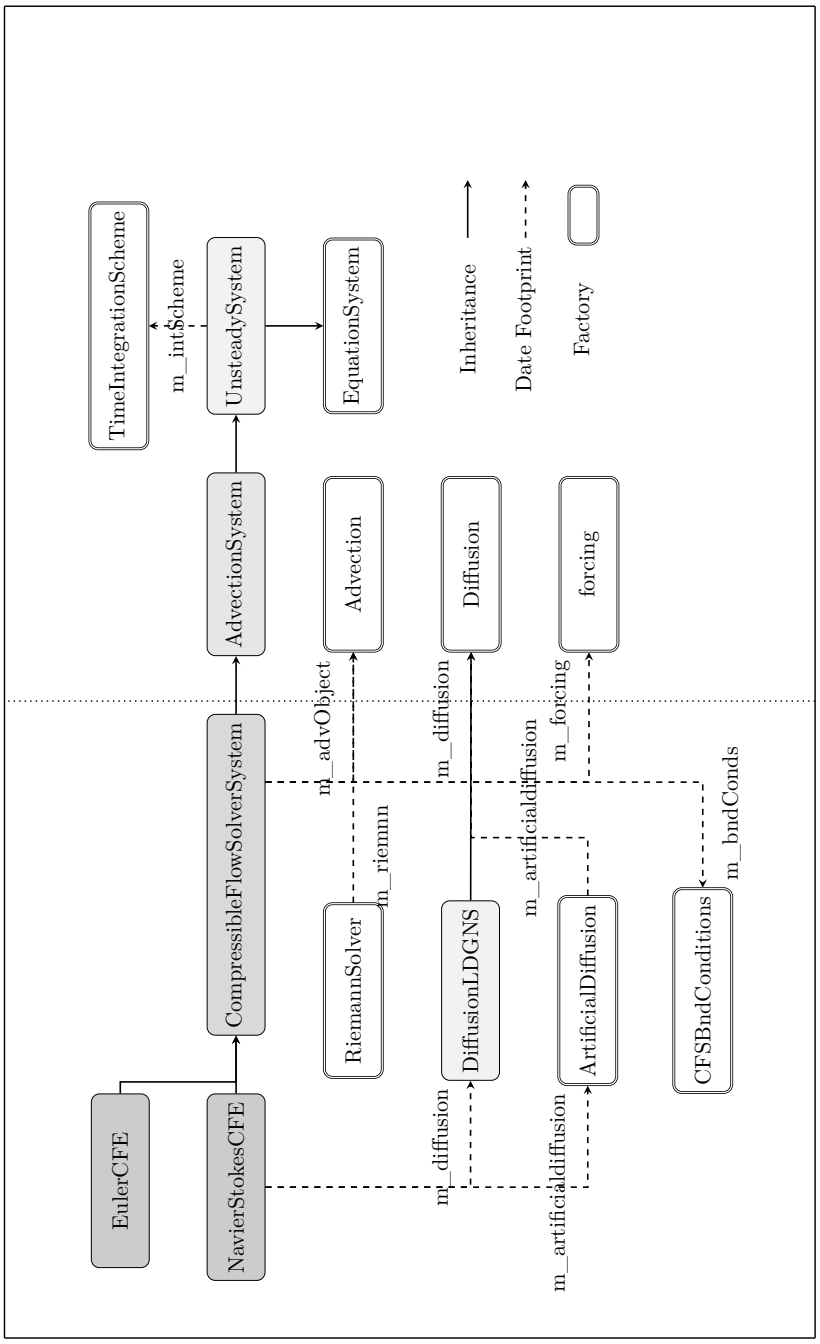
CompressibleFlowSolver: Solving the Compressible Navier-Stokes Equations

In this chapter, we walk the reader through our 2D and 3D compressible Navier-Stokes Solver (CompressibleFlowSolver).

15.1 Fundamental Theories of CompressibleFlowSolver

15.2 Data Structure of CompressibleFlowSolver

Figure 15.1: CompressibleFlowSolver DataStructure



15.3 Flow Chart of CompressibleFlowSolver

Figure 15.2: CompressibleFlowSolver Main Flow Chart

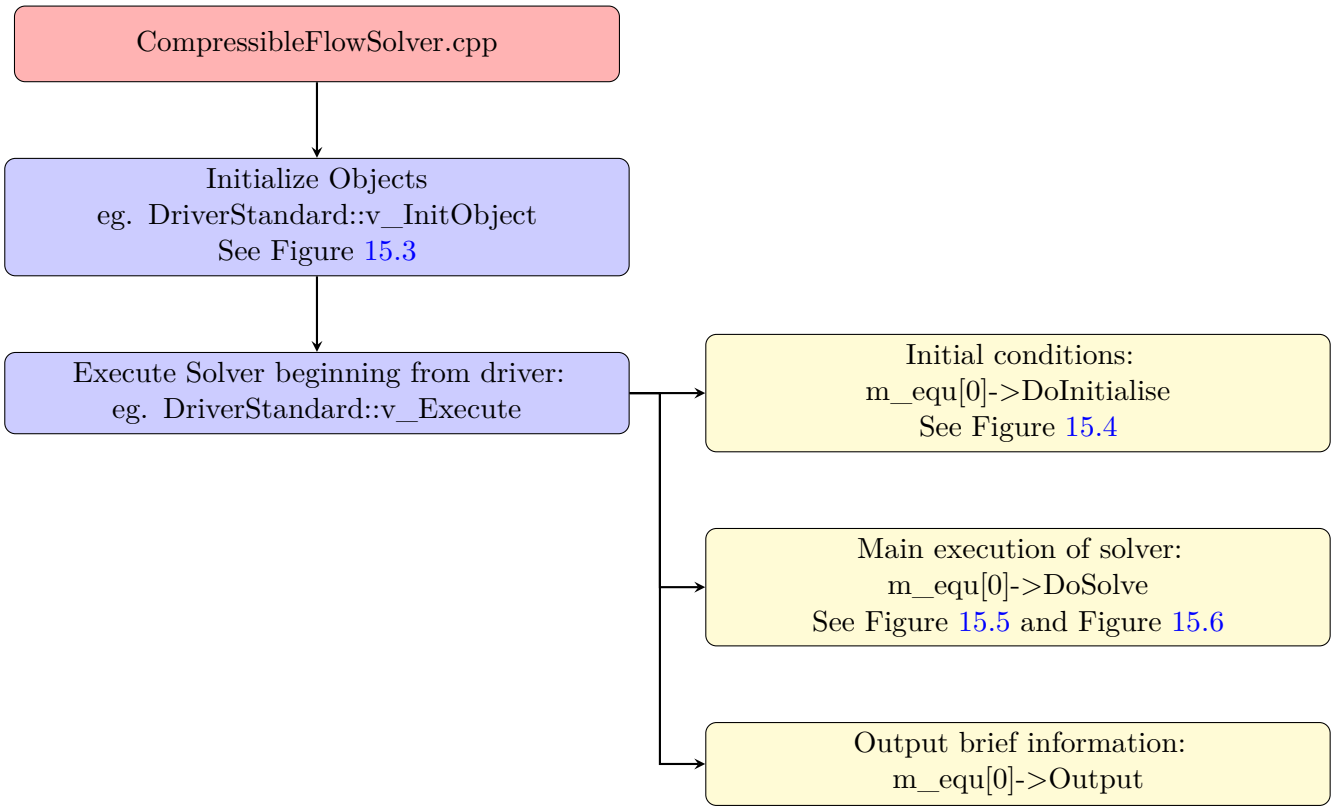


Figure 15.3: CompressibleFlowSolver InitObject

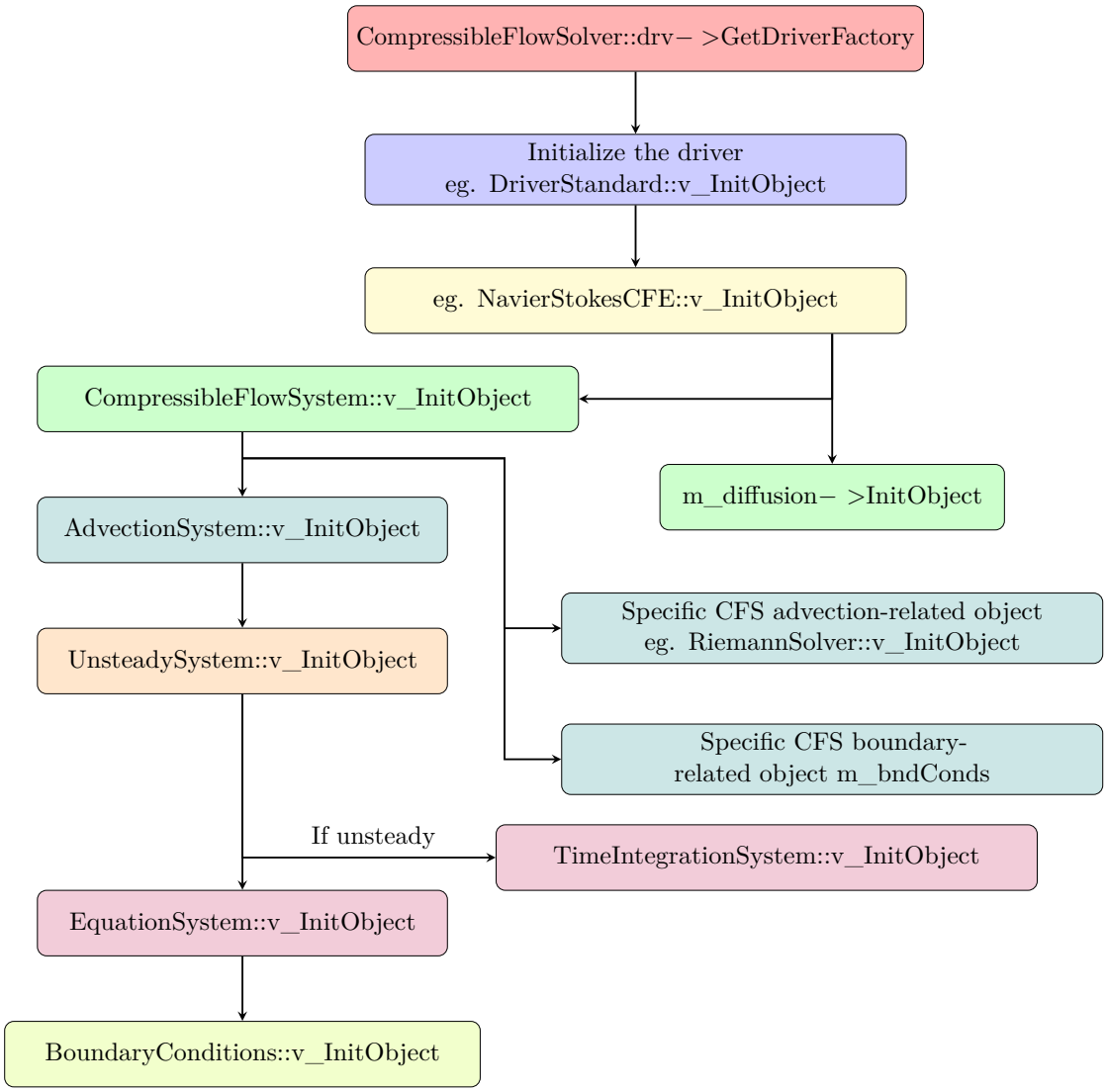


Figure 15.4: CompressibleFlowSolver Initialize Conditions

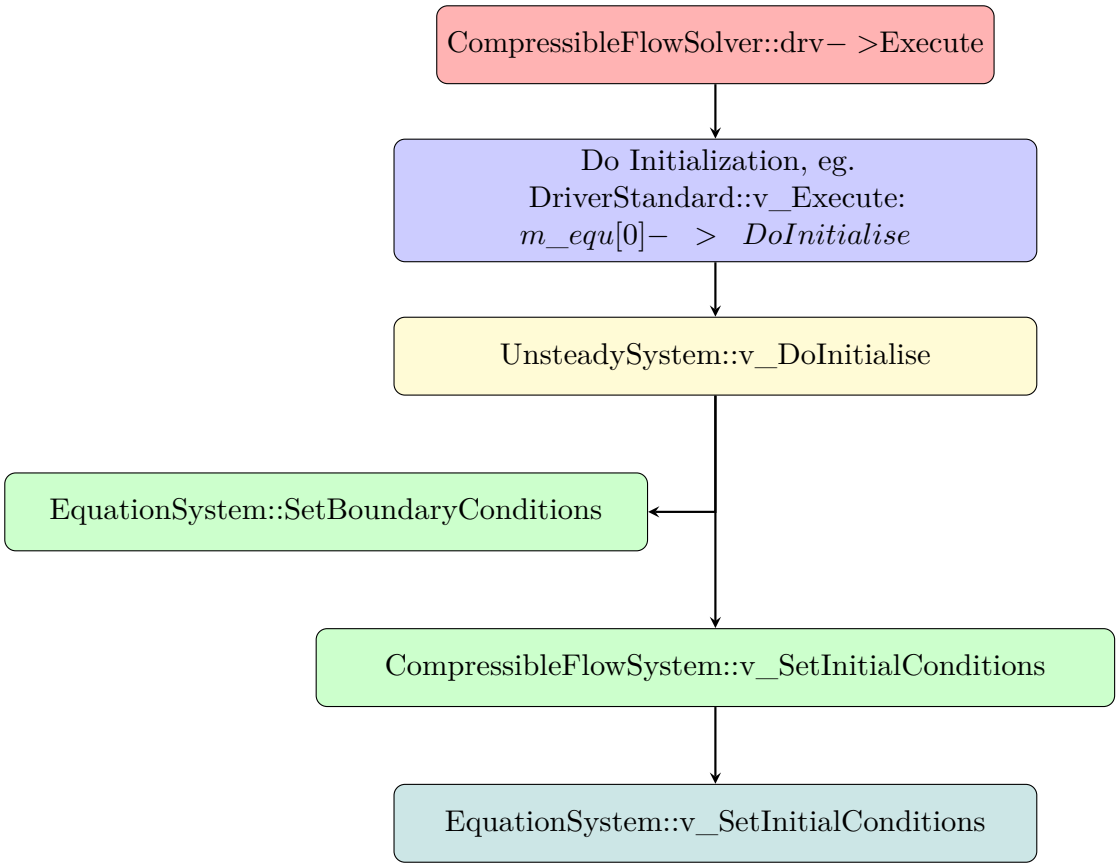


Figure 15.5: CompressibleFlowSolver Execute Advection

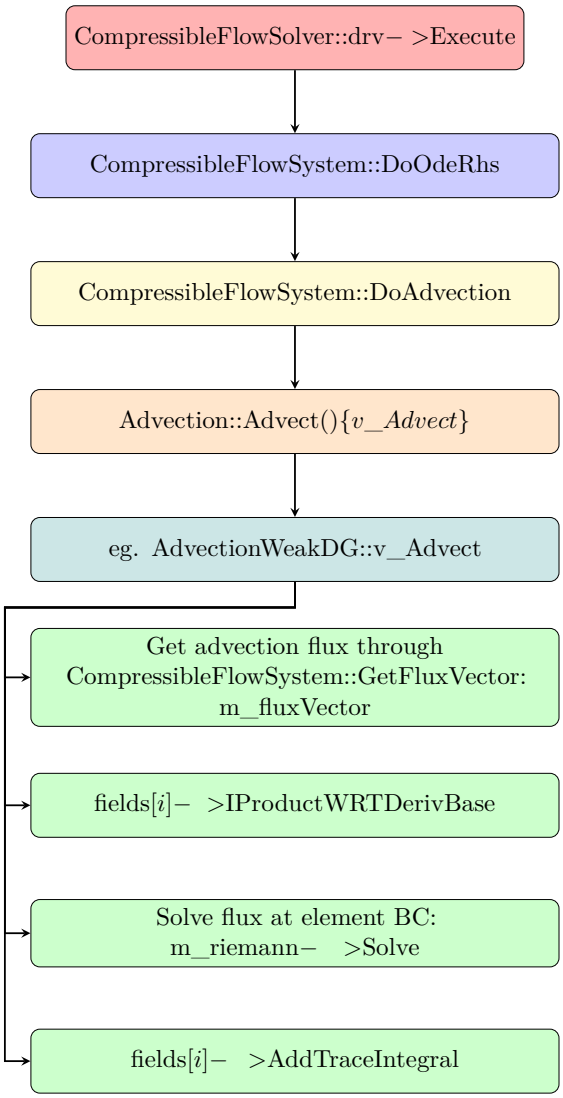
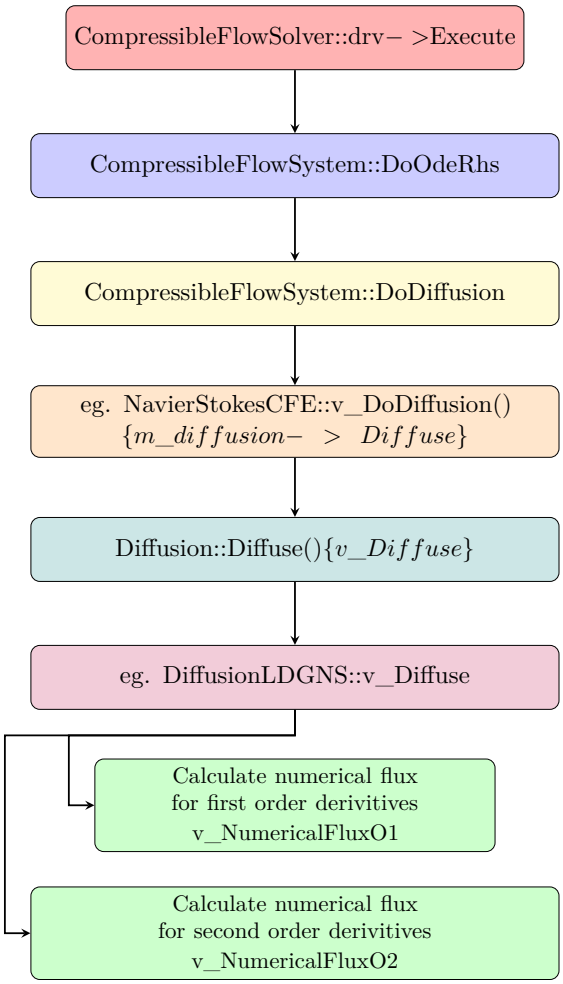


Figure 15.6: CompressibleFlowSolver Execute Diffusion



Part III

Utilities

In this part of the developer's guide, we walk you through development aspects of the various utilities that are available within publicly-available *Nektar++*repository. We encourage you to incorporate these utilities into your science and engineering work flows. If you find that the current utilities do not meet your needs and consequently you end up building a new utility based upon our framework, we encourage you to consider submitting this to us for incorporation in the publicly-available master branch.

Chapter 16

FieldConvert

In this chapter, we walk the reader through FieldConvert.

Chapter 17

NekMesh

In this chapter, we walk the reader through NekMesh.

Part IV

NekPy: Python interface to *Nektar++*

Chapter 18

Introduction

This part of the guide contains the information on using and developing `NekPy` Python wrappers for the *Nektar++* spectral/*hp* element framework. *As a disclaimer, these wrappings are experimental and incomplete.* You should not rely on their current structure and API remaining unchanged.

Currently, representative classes from the `LibUtilities`, `StdRegions`, `SpatialDomains`, `LocalRegions` and `MultiRegions` libraries have been wrapped in order to show the proof-of-concept.

18.1 Features and functionality

`NekPy` uses the `Boost.Python` library to provide a set of high-quality, hand-written Python bindings for selected functions and classes in Nektar++.

It is worth noting that Python (CPython, the standard Python implementation written in C, in particular) includes C API and that everything in Python is strictly speaking a C structure called `PyObject`. Hence, defining a new class, method etc. in Python is in reality creating a new `PyObject` structure.

Boost.Python is essentially a wrapper for Python C API which conveniently exports C++ classes and methods into `PyObjects`. At compilation time a dynamic library is created which is then imported to Python, as shown in Figure 18.1.

A typical snippet could look something like:

Listing 18.1: NekPy sample snippet

```
1 from NekPy.LibUtilities import PointsKey, PointsType, BasisKey, BasisType
2 from NekPy.StdRegions import StdQuadExp
3 import numpy as np
4 numModes = 8
```

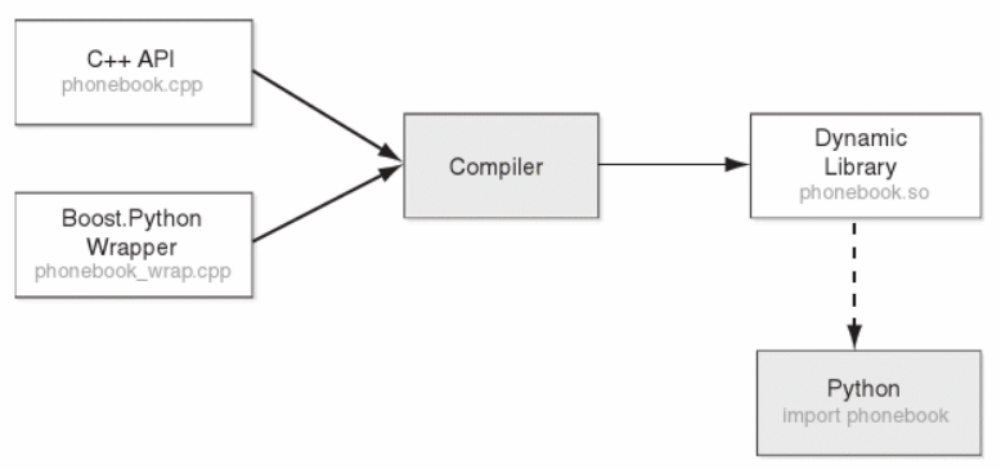


Figure 18.1: A schematic diagram of how C++ code is converted into Python with Boost.Python [49]

```
numPts   = 9
ptsKey   = PointsKey(numPts, PointsType.GaussLobattoLegendre)
basisKey = BasisKey(BasisType.Modified_A, numModes, ptsKey)
quadExp  = StdQuadExp(basisKey, basisKey)
x, y     = quadExp.GetCoords()
fx       = np.sin(x) * np.cos(y)
proj     = quadExp.FwdTrans(fx)
```

`NekPy` uses the `Boost.NumPy` library, contained in Boost 1.63+, to automatically convert C++ `Array<OneD, >` objects to and from the commonly-used `numpy.ndarray` object, which makes the integration more seamless between Python and C++.

Chapter 19

Installing NekPy

NekPy has the following list of requirements:

- Boost with Python support
- Nektar++ `master` branch compiled from source (i.e. not from packages)
- Python 2.7+ (note that examples rely on Python 2.7)
- NumPy

Most of these can be installed using package managers on various operating systems, as we describe below. We also have a requirement on the `Boost.NumPy` package, which is available in Boost 1.63 or later. If this isn't found on your system, it will be automatically downloaded and compiled.

19.1 Compiling and installing Nektar++

Nektar++ should be compiled as per the user guide instructions and installed into a directory which we will refer to as `$NEKDIR`. By default this is the `dist` directory inside the Nektar++ build directory.

Note that Nektar++ must, at a minimum, be compiled with `NEKTAR_BUILD_LIBRARY`, `NEKTAR_BUILD_UTILITIES` , `NEKTAR_BUILD_SOLVERS` and `NEKTAR_BUILD_PYTHON`. This will automatically download and install `Boost.NumPy` if required. Note that all solvers may be disabled as long as the `NEKTAR_BUILD_SOLVERS` option is set.

19.1.1 macOS

Homebrew

Users of Homebrew should make sure their installation is up-to-date with `brew upgrade`. Then run

```
brew install python boost-python
```

To install the NumPy package, use the `pip` package manager:

```
pip install numpy
```

MacPorts

Users of MacPorts should sure their installation is up-to-date with `sudo port selfupdate && sudo port upgrade outdated`. Then run

```
sudo port install python27 py27-numpy
sudo port select --set python python27
```

19.1.2 Linux: Ubuntu/Debian

Users of Debian and Ubuntu Linux systems should sure their installation is up-to-date with `sudo apt-get update && sudo apt-get upgrade`

```
sudo apt-get install libboost-python-dev python-numpy
```

19.1.3 Compiling the wrappers

Run the following command in `$NEKDIR/build` directory to install the Python package for the current user:

```
make nekpy-install-user
```

Alternatively, the following command can be used to install the package for all users:

```
make nekpy-install-system
```

19.2 Using the bindings

By default, the bindings will install into the `dist` directory, along with a number of examples that are stored in the `$NEKDIR/library/Demos/Python` directory. To test your

installation, you can for example run one of these (e.g. `python Basis.py`) or launch an interactive session:

```
$ cd builds
$ python
Python 2.7.13 (default, Apr  4 2017, 08:47:57)
[GCC 4.2.1 Compatible Apple LLVM 8.1.0 (clang-802.0.38)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> from NekPy.LibUtilities import PointsKey, PointsType
>>> PointsKey(10, PointsType.GaussLobattoLegendre)
<NekPy.LibUtilities._LibUtilities.PointsKey object at 0x11005c310>
```

19.2.1 Examples

A number of examples of the wrappers can be found in the `$NEKDIR/library/Demos/Python` directory, along with a sample mesh `newsquare_2x2.xml`:

- `SessionReader.py` is the simplest example and shows how to construct a session reader object. Run it as `python SessionReader.py mesh.xml`.
- `Basis.py` shows functionality of basic `LibUtilities` points and basis classes. Run this as `python Basis.py`.
- `StdProject.py` shows how to use some of the `StdRegions` wrappers and duplicates the functionality of `Basis.py` using the `StdExpansion` class. Run this as `python StdProject.py`.
- `MeshGraph.py` loads a mesh and prints out some basic properties of its quadrilateral elements. Run it as `python MeshGraph.py newsquare_2x2.xml`.

If you want to modify the source files, it's advisable to edit them in the `$NEKDIR/library/Demos/Python` directory and re-run `make install`, otherwise local changes will be overwritten by the next `make install`.

Chapter 20

Package structure

The NekPy wrapper is designed to mimic the library structure of Nektar++, with directories for the `LibUtilities`, `SpatialDomains` and `StdRegions` libraries. This is a deliberate design decision, so that classes and definitions in Nektar++ can be easily located inside NekPy.

There are also some other directories and files:

- `LibUtilities/Python/NekPyConfig.hpp` is a convenience header that all `.cpp` files should import. It sets appropriate namespaces for `boost::python` and `boost::python::numpy`, depending on whether the `Boost.NumPy` library was compiled or is included in Boost,
- `cmake/python` contains templates for `init.py` and `setup.py` files which every Python package should contain,
- `cmake/ThirdPartyPython.cmake` is a CMake configuration file which searches for `Boost.Python` and prepares `make` targets for installing NekPy.

Figure 20.1 shows the location of Python wrapper files within Nektar++ structure. Every sub-module of Nektar++ hosts an additional folder for Python files and the structure of the Python folder mimics the structure of the sub-module itself, as shown on the left of the figure with `LibUtilities` sub-module. Individual `.cpp` files, such as `Expansion.cpp` contain the wrappers for classes and methods from corresponding Nektar++ library files whereas `.cpp` files named after sub-modules (e.g. `LibUtilities.cpp`) contain `BOOST_PYTHON_MODULE` definitions which create package modules (e.g. `NekPy.LibUtilities`).

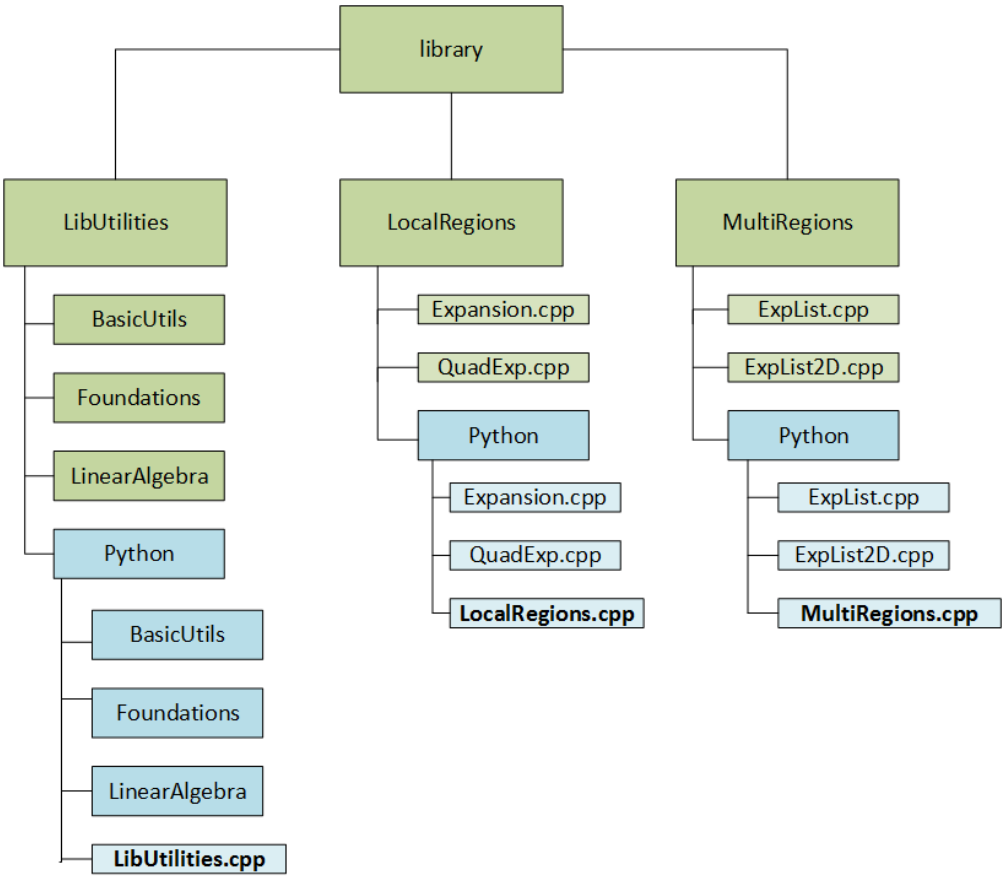


Figure 20.1: The location of Python wrapper files within Nektar++ structure.

Chapter 21

NekPy wrapping guide

This section attempts to outline some of the basic principles of the NekPy wrapper, which relies entirely on the excellent `Boost.Python` library. An extensive documentation is therefore beyond the scope of this document, but we highlight aspects that are important for the NekPy wrappers.

In general, note that when things go wrong with `Boost.Python`, it'll be indicated either by an extensive compiler error, or a runtime error in the Python interpreter when you try to use your wrapper. Judicious use of Google is therefore recommended to track down these issues!

You may also find the following resources useful:

- The `Boost.Python` tutorial [22],
- The `Boost.Python` entry on the Python wiki [2],
- Examples on GitHub [5],
- The OpenStreetGraph cookbook [4] and rationale for using manual wrapping [3] which served as a starting point for this project.

To demonstrate how to wrap classes, we'll refer to a number of existing parts of the code below.

21.1 Defining a library

First consider `LibUtilities`. An abbreviated version of the base file, `LibUtilities.cpp` has the following structure:

Listing 21.1: Defining a library with Boost.Python

```
#include <LibUtilities/Python/NekPyConfig.hpp>

void export_Basis();
void export_SessionReader();

BOOST_PYTHON_MODULE(_LibUtilities)
{
    // Initialise Boost.NumPy.
    np::initialize();

    // Export classes.
    export_Basis();
    export_SessionReader();
}
```

The `BOOST_PYTHON_MODULE(name)` macro allows us to define a Python module inside C++. Note that in this case, the leading underscore in the name (i.e. `_LibUtilities`) is deliberate. To define the contents of the module, we call a number of functions that are prefixed by `export_`, which will define one or more Python classes that live in this module. These Python classes correspond with our Nektar++ classes. We adopt this approach simply so that we can split up the different classes into different files, because it is not possible to call `BOOST_PYTHON_MODULE` more than once. These functions are defined in appropriately named files, for example `export_Basis()` lives in the file `LibUtilities/Python/Foundations/Basis.cpp`. This corresponds to the Nektar++ file `LibUtilities/Foundations/Basis.cpp` and the classes defined therein.

21.2 Basic class wrapping

As a very basic example of wrapping a class, let's consider the `SessionReader` wrapper.

Listing 21.2: Basic class wrapping with Boost.Python

```
void export_SessionReader()
{
    py::class_<SessionReader,
           std::shared_ptr<SessionReader>,
           boost::noncopyable>(
               "SessionReader", py::no_init)

        .def("CreateInstance", SessionReader_CreateInstance)
        .staticmethod("CreateInstance")

        .def("GetSessionName", &SessionReader::GetSessionName,
             py::return_value_policy<py::copy_const_reference>())

        .def("Finalise", &SessionReader::Finalise)
        ;
}
```

21.2.1 `py::class_<>`

This `Boost.Python` object defines a Python class in C++. It is templated, and in this case we have the following template arguments:

- `SessionReader` is the class that will be wrapped
- `std::shared_ptr<SessionReader>` indicates that this object should be stored inside a shared (or smart) pointer, which we frequently use throughout the library, as can be seen by the frequent use of `SessionReaderSharedPtr`
- `boost::noncopyable` indicates that `Boost.Python` shouldn't try to automatically wrap the copy constructor of `SessionReader`. We add this here because of compiler errors due to subclasses used inside `SessionReader`, but generally, this should be used for abstract classes which can't be copied.

We then have two arguments:

- `"SessionReader"` is the name of the class in Python.
- `py::no_init` indicates this object has no publically-accessible initialiser. This is because for `SessionReader`, we define a factory-type function called `CreateInstance` instead.

21.2.2 Wrapping member functions

We then call the `.def` function on the `class_<>`, which allows us to define member functions on our class. This is equivalent to `def`-ing a function in Python. `.def` has two required parameters, and one optional parameter:

- The function name as a string, e.g. `"GetSessionName"`
- A function pointer that defines the C++ function that will be called
- An optional return policy, which we need to define when the C++ function returns a reference.

`Boost.Python` is very smart and can convert many Python objects to their equivalent C++ function arguments, and C++ return types of the function to their respective Python object. Many times therefore, one only needs to define the `.def()` call.

However, there are some instances where we need to do some additional conversion, mask some C++ complexity from the Python interface, or deal with functions that return references. We describe ways to deal with this below.

Thin wrappers

Instead of defining a function pointer to a member of the C++ class, we can define a function pointer to a separate function that defines some extra functionality. This is called a *thin wrapper*.

As an example, consider the `CreateInstance` function. In C++ we pass this function the command line arguments in the usual `argc`, `argv` format. In Python, command line arguments are defined as a list of strings inside `sys.argv`. However, `Boost.Python` does not know how to convert this list to `argc, argv`, so we need some additional code.

Listing 21.3: Thin wrapper example

```
SessionReaderSharedPtr SessionReader_CreateInstance(py::list &ns)
{
    // ... some code here that converts a Python list to the standard
    // c/c++ (int argc, char **argv) format for command line arguments.
    // Then use this to construct a SessionReader and return it.
    SessionReaderSharedPtr sr = SessionReader::CreateInstance(argc, argv);
    return sr;
}
```

In Python, we can then simply call `session = SessionReader.CreateInstance(sys.argv)`.

Dealing with references

When dealing with functions in C++ that return references, e.g. `const NekDouble &GetFactor()` we need to supply an additional argument to `.def()`, since Python immutable types such as strings and integers cannot be passed by reference. For a full list of options, consult the `Boost.Python` guide. However a good rule of thumb is to use `copy_const_reference` as highlighted above, which will create a copy of the const reference and return this.

Dealing with `Array<OneD, >`

The `LibUtilities/Python/BasicUtils/SharedArray.cpp` file contains a number of functions that allow for the automatic conversion of Nektar++ `Array<OneD, >` storage to and from NumPy `ndarray` objects. This means that you can wrap functions that take these as parameters and return arrays very easily. However bear in mind the following caveats:

- Any NumPy `ndarray` created from an `Array<OneD, >` (and vice versa) will share their memory. Although this avoids expensive memory copies, it means that changing the C++ array changes the contents of the NumPy array (and vice versa).
- Many functions in Nektar++ return Arrays through argument parameters. In Python this is a very unnatural way to write functions. For example:

```
# This is good
x, y, z = exp.GetCoords()
# This is bad
x, y, z = np.zeros(10), np.zeros(10), np.zeros(10)
exp.GetCoords(x,y,z)
```

Use thin wrappers to overcome this problem. For examples of how to do this, particularly in returning tuples, consult the `StdRegions/StdExpansion.cpp` wrapper which contains numerous examples.

- `TwoD` and `ThreeD` arrays are not supported.

More information on the memory management and how the memory is shared can be found in Section 23.

21.2.3 Inheritance

Nektar++ makes heavy use of inheritance, which can be translated to Python quite easily using `Boost.Python`. For a good example of how to do this, you can examine the `StdRegions` wrapper for `StdExpansion` and its elements such as `StdQuadExp`. In a cut-down form, these look like the following:

Listing 21.4: Inheritance with Boost.Python

```
void export_StdExpansion()
{
    py::class_<StdExpansion,
               std::shared_ptr<StdExpansion>,
               boost::noncopyable>(
                   "StdExpansion", py::no_init);
}
void export_StdQuadExp()
{
    py::class_<StdQuadExp, py::bases<StdExpansion>,
               std::shared_ptr<StdQuadExp> >(
                   "StdQuadExp", py::init<const LibUtilities::BasisKey&,
                   const LibUtilities::BasisKey&>());
}
```

Note the following:

- `StdExpansion` is an abstract class, so it has no initialiser and is non-copyable.
- We use `py::bases<StdExpansion>` in the definition of `StdQuadExp` to define its parent class. This does not necessarily need to include the full hierarchy of C++ inheritance: in `StdRegions` the inheritance graph for `StdQuadExp` looks like `StdExpansion -> StdExpansion2D -> StdQuadExp`. In the above wrapper, we omit the StdExpansion2D call entirely.

- `py::init<>` is used to show how to wrap a C++ constructor. This can accept any arguments for which you have either written explicit wrappers or `Boost.Python` already knows how to convert.

21.2.4 Wrapping enums

Most Nektar++ enumerators come in the form:

```
enum MyEnum {
    eItemOne,
    eItemTwo,
    SIZE_MyEnum
};
static const char *MyEnumMap[] = {
    "ItemOne"
    "ItemTwo"
    "ItemThree"
};
```

To wrap this, you can use the `NEKPY_WRAP_ENUM` macro defined in `NekPyConfig.hpp`, which in this case can be used as `NEKPY_WRAP_ENUM(MyEnum, MyEnumMap)`. Note that if instead of `const char *` the map is defined as a `const std::string`, you can use the `NEKPY_WRAP_ENUM_STRING` macro.

Chapter 22

Documentation

The NekPy package certainly had to be documented in order to provide an easily accessible information about the wrapped classes to both users and developers. Ideally, the documentation should be:

- easily readable by humans,
- accessible using Python's inbuilt `help` method,
- compatible with the existing Nektar++ doxygen-based documentation.

Traditionally, Python classes and functions are documented using a docstring – a string occurring as the very first statement after the function or class is defined. This string is then accessible as the `__doc__` attribute of the function or class. The conventions associated with Python docstrings are described in PEP 257 document [34].

Boost.Python provides an easy way to include docstrings in the wrapped methods and classes as shown in Listing 22.1. The included docstrings will appear when Python `help` method is used.

Listing 22.1: Example of class and method documentation in Boost.Python

```
1 void export_Points()
2 {
3         py::class_<PointsKey>("PointsKey",
4         "Create a PointsKey which uniquely defines quadrature points.\n"
5         "\n"
6         "Args:\n"
7         "\tnQuadPoints (integer): The number of quadrature points.\n"
8         "\tpointsType (PointsType object): The type of quadrature points.",
9         py::init<const int, const PointsType&>())
10
```

```
            .def("GetNumPoints", &PointsKey::GetNumPoints,
            "Get the number of quadrature points in PointsKey.\n"
            "\n"
            "Args:\n"
            "\tNone\n"
            "Returns:\n"
            "\tInteger defining the number of quadrature points in
    PointsKey.")
}
```

In order to fully document the existing bindings a number of enumeration type classes such as `PointsType` had to have docstrings included which proved to be a challenge since Boost.Python does not provide a way to do this. Instead a direct call to Python C API has to be made and the method adapted from [52] was used, as shown in Listing 22.2. A downside of this solution is that it does requires the developer to manually update the Python documentation if the enumeration type is ever changed (e.g. adding a new type of point) as the code does not automatically gather information from the C++ class. In theory it could be possible to create a Python script which would generate Python docstrings based on the existing C++ documentation using regular expressions; however it would be difficult to integrate this solution into the existing framework.

Listing 22.2: Code used to include dosctings in enumetation type classes - part of `NekPyConfig.hpp`

```
#define NEKPY_WRAP_ENUM_STRING_DOCS(ENUMNAME,MAPNAME,DOCSTRING)    \
    {                                                              \
        py::enum_<ENUMNAME> tmp(#ENUMNAME);                        \
        for (int a = 0; a < (int)SIZENAME(ENUMNAME); ++a)          \
        {                                                          \
            tmp.value(MAPNAME[a].c_str(), (ENUMNAME)a);            \
        }                                                          \
        tmp.export_values();                                       \
        PyTypeObject * pto =                                       \
            reinterpret_cast<PyTypeObject*>(tmp.ptr());            \
        PyDict_SetItemString(pto->tp_dict, "__doc__",              \
            PyString_FromString(DOCSTRING));                       \
    }
```

There are many docstrings conventions that are popular in Python such as Epytext, reST and Google therefore a choice had to be made as to which docstring style to use. After considering the criteria which the documentation had to fulfill it was decided to use Google Python Style [53] as it is highly readable by humans (and hence an excellent choice for documentation which will be primarily accessible by Python `help` method) and can be used to generate automated documentation pages with Sphinx (a tool for creating Python documentation).

Unfortunately, it proved to be difficult to include the documentation of NumPy package in the existing doxygen-based documentation due to the fact that the docstrings are

generated by Boost.Python. It was decided that if the time constraints of the project permit this problem could be resolved at a later date and the possibility of accessing the documentation though inbuilt `help` method was deemed sufficient.

Chapter 23

Memory management in NekPy

A significant amount of effort has been invested into developing efficient memory management techniques, to enable the sharing of memory between Python and C++ for the ubiquitous *Nektar++* `Array` structure which is used throughout the code. The computations carried out in *Nektar++* are usually highly demanding on the machines they are run on, therefore any unnecessary data duplication or memory leaks would be detrimental to software performance.

This section first outlines some precusor information relating to the basics of memory management in C++ and Python, before explaining the strategy used in NekPy to effectively use the resources when passing data between the different layers of the bindings.

23.1 Memory management in C++ and Python

23.1.1 C++

In C++ the memory used by the application is divided into five containers [48]:

- text segment – containing the set of instructions for the program to execute;
- data segment – containing static and global variables initialised with values, e.g. `float pi = 3.14;` – the size of the data segment is pre-determined at the time of the compilation of the program and depends on the size of the variables in the source code;
- bss (block started by symbol) segment – containing static and global variables not explicitly initialised with any value, e.g. `int n;` – the size of the data segment is pre-determined at the time of the compilation of the program and depends on the size of the variables in the source code;
- stack – a LIFO (last in, first out) structure containing the function calls and variables intialised in the functions – the size of the stack is pre-determined at

the time of compilation of the program and depends on the operating system and development environment;

- heap – containing all dynamically allocated memory (e.g. using instruction `new` in C++).

The access to data on the heap is maintained by pointers which store the memory address of said data. If the pointer ceases to exist (e.g. because the function which contained it finished running and hence was taken off the stack) the programmer has no way of referencing the data on the heap again. The unused, inaccessible data will stay in the memory, consuming valuable resources if not properly deallocated using `delete` instruction. Issues may arise if the same memory address is held by two different pointers – the deallocation of memory can render one of the pointers empty and possibly lead to errors.

In order to facilitate memory management, C++11 standard introduced shared pointers (`shared_ptr`) [1]. Shared pointers maintain a reference counter which is increased when another shared pointer refers to the same memory address. The memory will only be deallocated when all shared pointers referencing the memory are destroyed, as shown in Figure 23.1.

Python

Python manages memory though a private heap containing all Python objects [30]. An internal memory manager is used to ensure the memory is properly allocated and deallocated while the script runs. In contrast to C++, Python's memory manager tries to optimise the memory usage as much as possible. For example, the same object reference is allocated to a new variable if the object already exists in memory. Another important feature of Python's memory manager is the use of garbage collector based on reference counting. When the object is no longer referenced by any variable the reference counter is set to zero, which triggers the garbage collector to free the memory (possibly at some later time). A disadvantage of this solution is slower execution time, since the garbage collector routines have to be called periodically. Features of the Python memory manager are schematically shown in Figure 23.2

It is unusual for the Python programmer to manually modify the way Python uses memory resources – however it is sometimes necessary, as is the case with this project. In particular, the Python C API exposes several macros to handle reference counting, and developers can increase or decrease the reference counter of the object using `Py_INCREF` and `Py_DECREF` respectively [31]. In `Boost.Python`, these calls are wrapped in the `py::incref` and `py::decref` functions.

23.2 Passing C++ memory to Python

To highlight the technique for passing C++ memory natively into Python's `ndarray`, we consider first the case of the native *Nektar++* matrix structure. In many situations, matri-

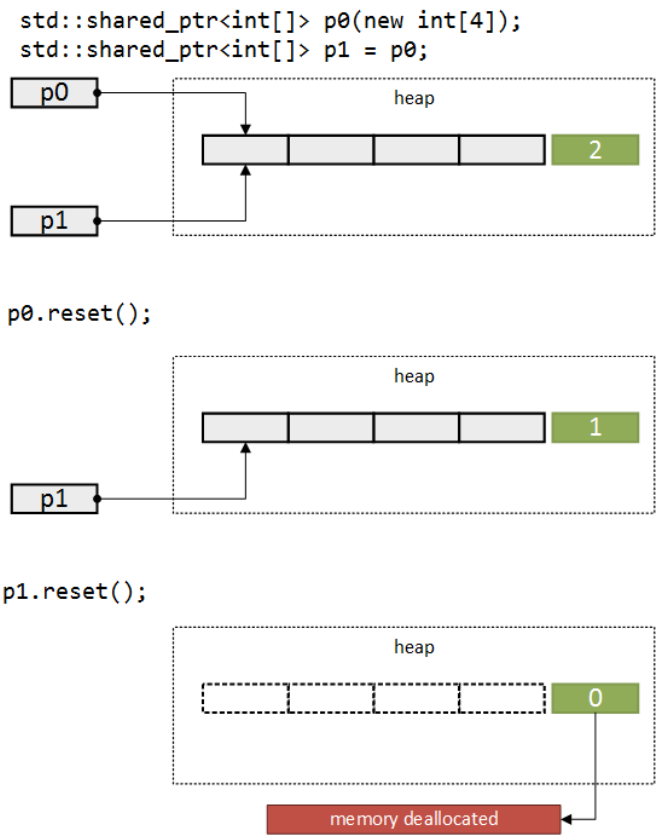


Figure 23.1: Memory management with C++11 `shared_ptr`. The two declared shared pointers reference an array of 4 integers. Reference counter shown in green.

ces created by *Nektar++* (usually a `shared_ptr` of `NekMatrix<D, StandardMatrixTag>` type) need to be passed to Python - for instance, when performing differentiation using e.g. Gauss quadrature rules a differentiation matrix must be obtained. In order to keep the program memory efficient, the data should not be copied into a NumPy array but rather be referenced by the Python interface. This, however, complicates the issue of memory management.

Consider a situation where C++ program no longer needs to work with the generated array and the memory dedicated to it is deallocated. If this memory has already been shares with Python, the Python interface may still require the data contained within the array. However since the memory has already been deallocated from the C++ side, this will typically cause an out-of-bounds memory exception. To prevent such situations a solution employing reference counting must be used.

Converter method

Boost.Python provides the methods to convert a C++ type element to one recognised by Python as well as to maintain appropriate reference counting. Listing 23.1 shows

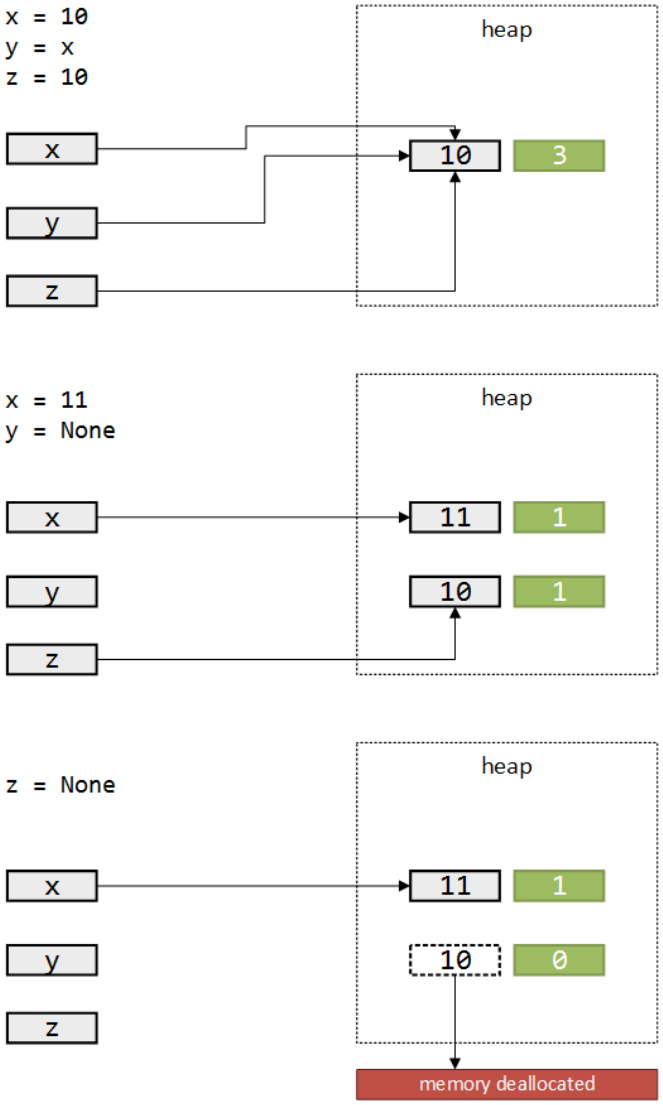


Figure 23.2: Memory management in Python. Memory is optimised as much as possible and garbage collector deallocates the memory if the reference counter (shown in green) reaches 0.

an abridged version of the converter method (for Python 2 only) with comments on individual parameters. The object requiring conversion is a `shared_ptr` of `NekMatrix<D, StandardMatrixTag>` type (named `mat`).

Listing 23.1: Converter method for converting the C++ arrays into Python NumPy arrays.

```
1 #include <LibUtilities/LinearAlgebra/NekMatrix.hpp>
```

```
#include <LibUtilities/LinearAlgebra/MatrixStorageType.h>
#include <NekPyConfig.hpp>

using namespace Nektar;
using namespace Nektar::LibUtilities;

template<typename T, typename F>
void NekMatrixCapsuleDestructor(void *ptr)
{
    // Destructor for shared_ptr when the capsule is deallocated.
    std::shared_ptr<NekMatrix<T, F>> *mat =
        (std::shared_ptr<NekMatrix<T, F>> *)ptr;
    delete mat;
}

template<typename T>
struct NekMatrixToPython
{
    static PyObject *convert(
        std::shared_ptr<NekMatrix<T, StandardMatrixTag>> const &mat)
    {
        // Create a PyCObject, which will hold a shared_ptr to the matrix.
        // When capsule is deallocated on the Python side, it will call the
        // destructor function above and release the shared_ptr reference.
        py::object capsule(py::handle<>(PyCObject_FromVoidPtr(
            new std::shared_ptr<NekMatrix<T, StandardMatrixTag>>(mat),
            NekMatrixCapsuleDestructor<T, StandardMatrixTag>)));

        int nRows = mat->GetRows(), nCols = mat->GetColumns();

        // increments Python reference counter to avoid immediate
    deallocation
        return py::incref(
            np::from_data(
                mat->GetRawPtr(), // pointer to data
                np::dtype::get_builtin<T>(), // data type
                py::make_tuple(nRows, nCols), // shape of the array
                py::make_tuple(sizeof(T), nRows * sizeof(T)), // stride of
    the array
                capsule).ptr()); // capsule - contains the object owning
    the data (preventing it from being prematurely deallocated)
    }
}

template<typename T>
void export_NekMatrix()
{
    py::to_python_converter<std::shared_ptr<NekMatrix<T, StandardMatrixTag
    >>,
                            NekMatrixToPython<T>>();
}
```

Firstly we give a brief overview of the general process undertaken during type conversion. Boost.Python maintains a registry of known C++ to Python conversion types, which by

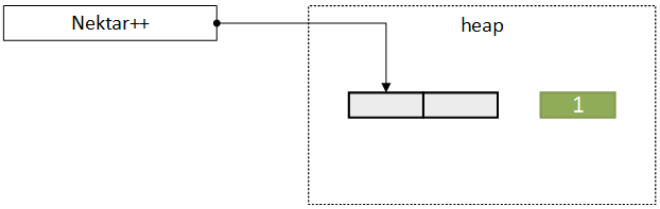
default allows for fundamental data type conversions such as `double` and `float`. In this manner, many C++ functions can be automatically converted, for example when they are used in `.def` calls when registering a Python class. Clearly the `convert` function here contains much of the functionality. In order to perform automatic conversion between the `NekMatrix` and a 2D `ndarray`, we register the conversion function inside Boost.Python's registry so that it is aware of the datatypes. We also note that throughout the conversion code (and elsewhere in NekPy), we make use of the Boost.NumPy bindings. These are a lightweight wrapper around the NumPy C API, which simplifies the syntax somewhat and avoids direct use of the API.

In terms of the conversion function itself, we first create a new Python capsule object. The capsule is designed to hold a C pointer and a callback function that is called when the Python object is deallocated. Since there is no Boost.Python wrapper around this, we use a `handle<>` to wrap it in a generic Boost.Python object. The strategy we therefore employ is to create a `shared_ptr`, which increases the reference counter of the `NekMatrix`. This will prevent it being destroyed if it is in use in Python, even if on the C++ side the memory is deallocated. The callback function simply deletes the `shared_ptr` when it is no longer required, cleaning up the memory appropriately. This process is shown in Figure 23.3. It is worth noting that the steps (c) and (d) can be reversed and the `shared_ptr` created by the Python binding can be removed first. In this case the memory will be deallocated only when the `shared_ptr` created by C++ is also removed.

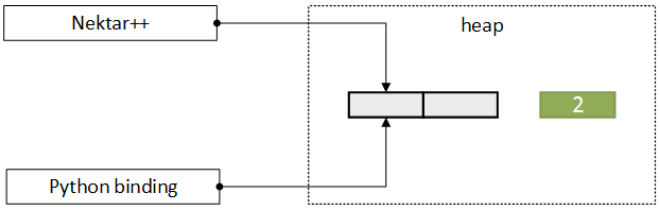
Finally, the converter method returns a NumPy array using the `np::from_data` method. Note that the `capsule` object is passed as the `ndarray` base argument – i.e. the object that owns the data. In this case, when all `ndarray` objects that refer to the data become deallocated, NumPy will not directly deallocate the data but instead release its reference to the capsule. The capsule will then be deallocated, which will decrement the counter in the `shared_ptr`. We also note that data ordering is important for matrix storage; *Nektar++* stores data in column-major order, whereas NumPy arrays are traditionally row-major. The stride of the array has to be passed into the `np::from_data` in a form of a tuple `(a, b)`, where `a` is the number of bytes needed to skip to get to the same position in the next row and `b` is the number of bytes needed to skip to get to the same position in the next column. In order to stop Python from immediately destroying the resulting NumPy array, its reference counter is manually increased before the array is passed on to Boost.Python and eventually returned to the user's code.

Testing

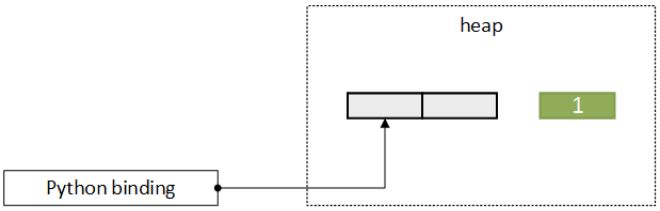
The process outlined above requires little manual intervention from the programmer. There are no almost no explicit calls to Python C API (aside from creating a `PyObject` – `PyCObject_FromVoidPtr`) as all operations are carried out by Boost.Python. Therefore, the testing focused mostly on correctness of returned data, in particular the order of the array. To this end, the *Differentiation* tutorials were used as tests. In order to correctly run the tutorials the Python wrapper needs to retrieve the differentiation matrix which, as mentioned before, has to be converted to a datatype Python recognises. The test runs



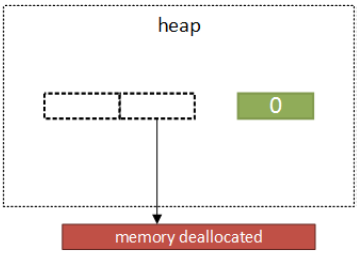
(a) Nektar++ creates a data array referenced by a `shared_ptr`



(b) Array is passed to Python binding which creates a new `shared_ptr` to the data



(c) Nektar++ no longer needs the data - its `shared_ptr` is removed but the memory is not deallocated



(d) When the data is no longer needed in the Python interface the destructor is called and `shared_ptr` is removed.

Figure 23.3: Memory management of data created in C++ using `shared_ptr` and passed to Python. Reference counter shown in green.

the differentiation tutorials and compares the final result to the fixed expected value. The test is automatically run as a part of `ctest` command if both the Python wrapper and the tutorials have been built.

23.2.1 Passing Python data to C++

Conversely, a similar problem exists when data is created in Python and has to be passed to the C++ program. In this case, as the data is managed by Python, the main reference counter should be maintained by the Python object and incremented or decremented as appropriate using `py::incref` and `py::decref` methods respectively. Although we do not support this process for the `NekMatrix` as described above, we do use this process for the `Array<OneD, >` structure. When the array is no longer needed by the C++ program the reference counter on the Python side should be decremented in order for Python garbage collection to work appropriately – however this should only happen when the array was created by Python in the first place.

The files implementing the below procedure are:

- `LibUtilities/Python/BasicUtils/SharedArray.cpp` and
- `LibUtilities/BasicUtils/SharedArray.hpp`

Modifications to `Array<OneD, const DataType>` class template

In order to perform the operations described above, the C++ array structure should contain information on whether or not it was created from data managed by Python. To this end, two new attributes were added to C++ `Array<OneD, const DataType>` class template in the form of a `struct`:

```
1 struct PythonInfo {
2     void *m_pyObject; // Underlying PyObject pointer
3     void (*m_callback)(void *); // Callback
4 };
```

where:

- `m_pyObject` is a pointer to the `PyObject` containing the data, which should be an `ndarray`;
- `m_callback` is a function pointer to the callback function which will decrement the reference counter of the `PyObject`.

Inside `Array<OneD, >`, this struct is held as a double pointer, i.e.:

```
1 PythonInfo **m_pythonInfo;
```

This is done because if the `ndarray` was created originally from a C++ to Python conversion (as outlined in the previous section), we need to convert the Array – and any

other shared arrays that hold the same C++ memory – to reference the Python array. If we did not do this, then it is possible that the C++ array could be destroyed whilst it is still used on the Python side, leading to an out-of-bounds exception. By storing this as a double pointer, in a similar fashion to the underlying reference counter `m_count`, we can ensure that all C++ arrays are updated when necessary. We can keep track of Python arrays by checking `*m_pythonInfo`; if this is not set to `nullptr` then the array has been constructed throught the Python to C++ converter.

Adding new attributes to the arrays might cause a significantly increased memory usage or additional unforeseen overheads, although this was not seen in benchmarking. However to avoid all possibility of this, a preprocessor directive has been added to only include the additional arguments if `NekPy` had been built (using the option `NEKTAR_BUILD_PYTHON`).

A new constructor has been added to the class template, as seen in Listing 23.2. `m_memory_pointer` and `m_python_decrement` have been set to `nullptr` in the pre-existing constructors. A similar constructor was added for `const` arrays to ensure that these can also be passed between the languages. Note that no calls to Nektar++ array initialisation policies are made in this constructor, unlike in the pre-existing ones, as there is no need for the new array to copy the data.

Listing 23.2: New constructor for initialising arrays created through the Python to C++ converter method.

```
1  Array(unsigned int dim1Size, DataType* data, void* memory_pointer, void (*
       python_decrement)(void *)) :
2      m_size( dim1Size ),
3      m_capacity( dim1Size ),
4      m_data( data ),
5      m_count( nullptr ),
6      m_offset( 0 )
7  {
8      m_count = new unsigned int();
9      *m_count = 1;
10
11     m_pythonInfo = new PythonInfo *();
12     *m_pythonInfo = new PythonInfo();
13     (*m_pythonInfo)->m_callback = python_decrement;
14     (*m_pythonInfo)->m_pyObject = memory_pointer;
15 }
```

Changes have also been made to the destructor, as shown in Listing 23.3, in order to ensure that if the data was initially created in Python the callback function would decrement the reference counter of the NekPy array object. The detailed procedure for deleting arrays is described further in this section.

Listing 23.3: The modified destructor for C++ arrays.

```
~Array()
{
    if( m_count == nullptr )
    {
        return;
    }

    *m_count -= 1;
    if( *m_count == 0 )
    {
        if (*m_pythonInfo == nullptr)
        {
            ArrayDestructionPolicy<DataType>::Destroy( m_data, m_capacity )
    ;
            MemoryManager<DataType>::RawDeallocate( m_data, m_capacity );
        }
        else
        {
            (*m_pythonInfo)->m_callback((*m_pythonInfo)->m_pyObject);
            delete *m_pythonInfo;
        }

        delete m_pythonInfo;
        delete m_count; // Clean up the memory used for the reference count.

    }
}
```

Creation of new arrays

The following algorithm has been proposed to create new arrays in Python and allow the C++ code to access their contents:

1. The NumPy array object to be converted is passed as an argument to a C++ method available in the Python wrapper.

2. The converter method is called to convert a Python NumPy array into C++ `Array<OneD, const DataType>` object.

3. If the NumPy array was created through the C++-to-Python process (which can be determined by checking the `base` of the NumPy array), then:

    - extract the *Nektar++* Array from the capsule;
    - convert the Array (and all of its other references) to a Python array so that any C++ arrays that share this memory also know to call the appropriate decrement function;
    - set the NumPy array's `base` to an empty object to destroy the original capsule.

4. Otherwise, the converter creates a new `Array<OneD, const DataType>` object with the following attribute values:
   - `data` points to the data contained by the NumPy array,
   - `memory_pointer` points to the NumPy array object,
   - `python_decrement` points to the function decrementing the reference counter of the `PyObject`,
   - subsequently, `m_count` is equal to 1.
5. The Python reference counter of the NumPy array object is increased.
6. If any new references to the array are created in C++ the `m_count` attribute is increased accordingly. Likewise, if new references to NumPy array object are made in Python the reference counter increases.

The process is schematically shown in Figure 23.4a and 23.4b.

Array deletion

The array deletion process relies on decrementing two reference counters: one on the Python side of the program (Python reference counter) and the other one on C++ side of the program. The former registers how many Python references to the data there are and if there is a C++ reference to the array. The latter (represented by `m_count` attribute) counts only the number of references on the C++ side and as soon as it reaches zero the callback function is triggered to decrement the Python reference counter so that registers that the data is no longer referred to in C++. Figure 23.5 presents the overview of the procedure used to delete the data.

In short, the fact that C++ uses the array is represented to Python as just an increment to the object reference counter. Even if the Python object goes out of scope or is explicitly deleted, the reference counter will always be non-zero until the callback function to decrement it is executed, as shown in Figure 23.4c. Similarly, if the C++ array is deleted first, the Python object will still exist as the reference counter will be non-zero (see Figure 23.4d).

Converter method

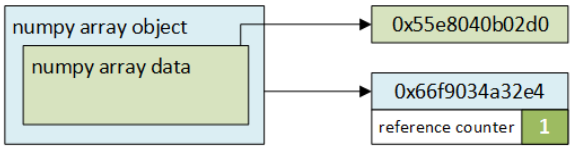
As with conversion from C++ to Python, a converter method was registered to make Python NumPy arrays available in C++ with Boost.Python, which can be found in the `SharedArray.cpp` bindings file. In essence, Boost.Python provides the used with a memory segment (all expressions containing `rvalue_from_python` are to do with doing that). The data has to be extracted from PyObject in order to be presented in a format C++ knows how to read – the `get_data` method allows the programmer to do it for NumPy arrays. Finally, care must be taken to manage memory correctly, thus the use

of borrowed references when creating Boost.Python object and the incrementation of PyObject reference counter at the end of the method.

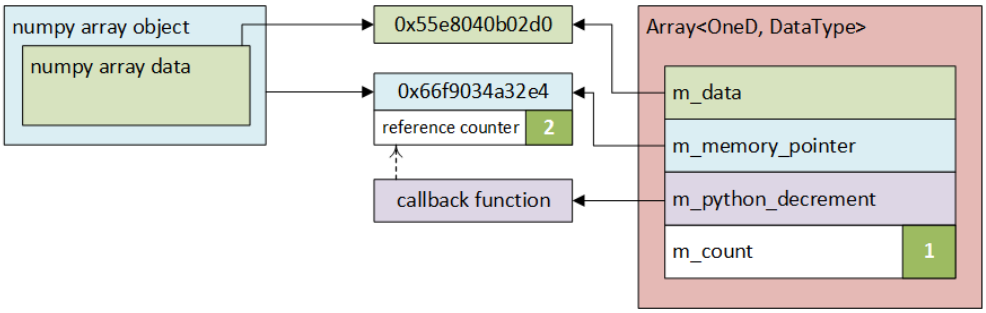
The callback decrement method is shown below in Listing 23.4. When provided with a pointer to PyObject it decrements it reference counter.

Listing 23.4: The decrement method called when the `m_count` of C++ array reaches 0.

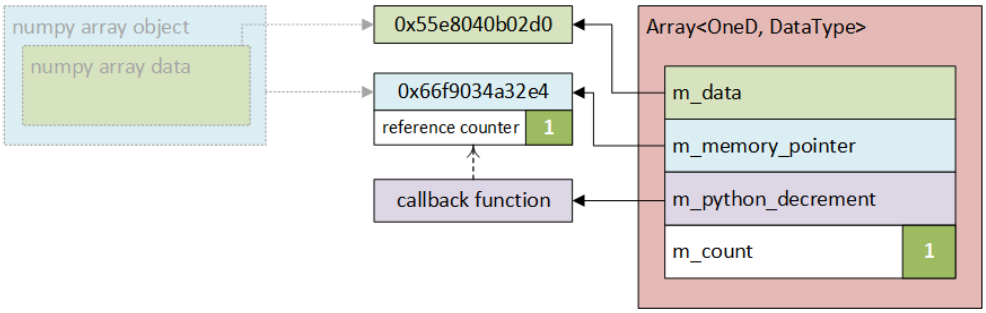
```
1 static void decrement(void *objPtr)
2     {
3         PyObject *pyObjPtr = (PyObject *)objPtr;
4         Py_XDECREF(pyObjPtr);
5     }
```



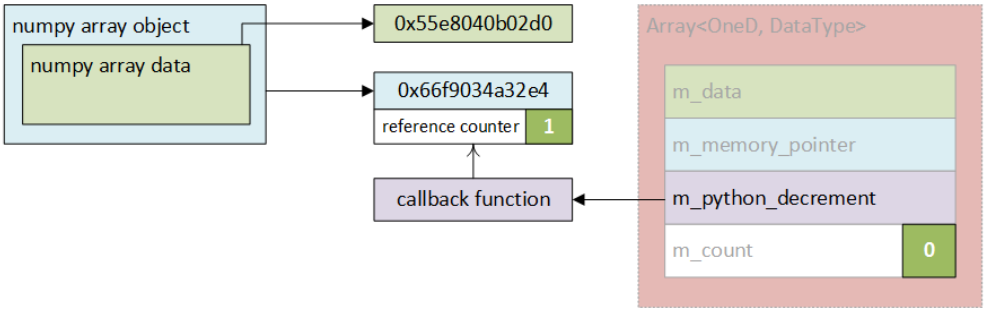
(a) The NumPy array is created in Python. Note that the NumPy object and the data it contains are represented by two separate memory addresses.



(b) The C++ array is created through the converter method: its attributes point to the appropriate memory addresses and the reference counter of the memory address of NumPy array object is incremented.



(c) If the NumPy object is deleted first, the reference counter is decremented, but the data still exists in memory.



(d) If the C++ array is deleted first, the callback function decrements the reference counter of the NumPy object but the data still exists in memory.

Figure 23.4: Diagram showing the process of creation and deletion of arrays. Reference counters shown in green.

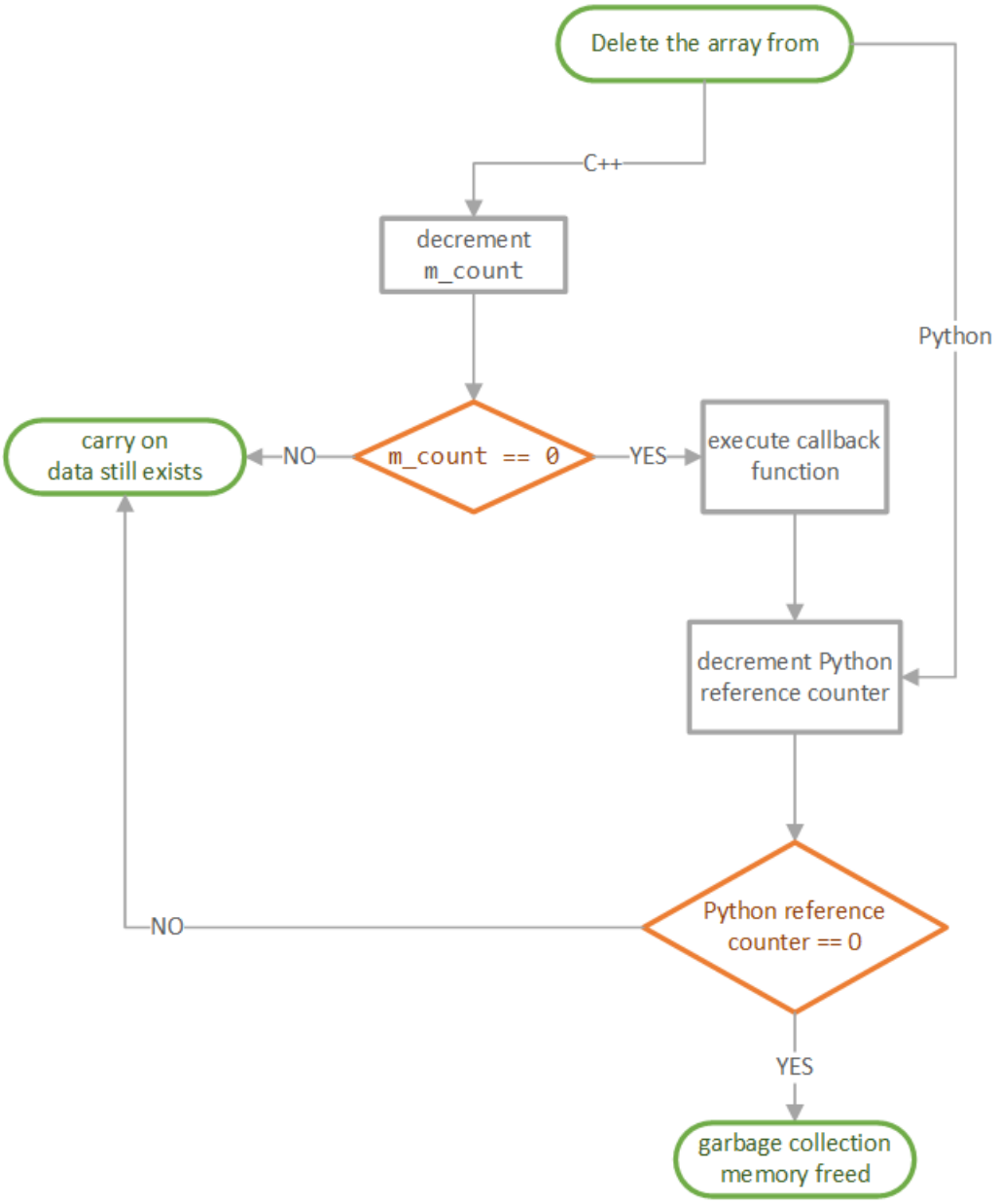


Figure 23.5: Flowchart describing the procedure for array deletion from either side of the code.

Testing

As the process of converting arrays from Python to C++ required making direct calls to C API and relying on custom-written methods, more detailed testing was deemed necessary. In order to thoroughly check if the conversion works as expected, three tests were conducted to determine whether the array is:

1. referenced (not copied) between the C++ program and the Python wrapper,
2. accessible to C++ program after being deleted in Python code,
3. accessible in Python script after being deleted from C++ objects.

Python files containing test scripts are currently located in `library\Demos\Python\tests`. They should be converted into unit tests that should be run when Python components are built.

Chapter 24

FieldConvert in NekPy

This chapter will describe the idea behind the `FieldConvert` utility in NekPy and discuss how the process is implemented through a Python `FieldConverter` class. As this part of the project has not been fully completed yet, this chapter also outlines the tasks that need to be done in order to show the proof-of-concept, as well as some ideas for further improving the tool.

24.1 Idea and motivation

The idea behind porting `FieldConvert` utility to Python is to allow the user to execute a seamless workflow, from converting the mesh and running calculations to preparing data visualisations. This solution is a potential improvement over the original `FieldConvert` tool, as the Python-based workflow can potentially be executed from a single Python script.

Another advantage of the Python version of the tool is the possibility of interacting with other software featuring Python interface, e.g. Paraview. This could make the process of visualising the results of computations done with *Nektar++* even easier.

24.2 Design and implementation

`FieldConvert` utility in Python was designed to provide the user with an easy workflow. Hence, a minimum effort is required to set up the conversion and the majority of work is done behind the scenes.

This section will discuss the design of the utility, including the description of work yet to be done, as well as the user workflow required to execute the basic conversions.

24.2.1 `FieldConverter` class

The entire procedure outlined in the original `FieldConvert` utility is managed by the `FieldConverter` class located in `utilities/FieldConvert/Python/FieldConvert.py`.

The class has the following attributes:

- `fieldSharedPtr`: a shared pointer to the field, necessary to initialise a module;
  - *TO-DO*: `FieldSharedPtr` type remains to be wrapped.
  - *TO-DO*: Whether a separate `FieldSharedPtr` is needed for each module initialisation needs to be discussed.
- `moduleFactory`: would be equivalent to `GetModuleFactory()` in the original utility;
  - *TO-DO*: Whether this is possible needs to be discussed.
  - *TO-DO*: It would be best to wrap the general template `NekFactory` and initialise a module factory this way.
- `availableModuleList`: holds a list of available modules in order to assert that the modules requested by the user are available;
  - *TO-DO*: `PrintAvailableClasses` method needs to be wrapped.
  - *TO-DO*: It would definitely be neater if the available modules existed as an enum rather than just strings containing names.
- `sessionFile`, `inputFile`, `outputFile`: hold the necessary filenames;
- `moduleList`: holds a list of `Module` objects, created as requested by the user;
- `variableMap`: equivalent of `vm` variable from the original utility.
  - *TO-DO*: This variable map needs to be constructed from user input as it needs to be passed into `Process` method of `Module` class.

24.2.2 User workflow

The currently suggested user workflow is as follows:

1. Initialise an instance of `FieldConverter` class.
2. Add a session file as well as an input and an output file.
   - The converter will assert that the session file and the input file exist, assert that the file extensions are supported and create appropriate modules.
3. Add any modules, e.g. `vorticity`.
   - Currently the argument passed into `addProcessModule` is a string containing the module name.

- *TO-DO*: As mentioned before, it would be neater if said argument was an enum.
- *TO-DO*: Some more thought is required about how to support modules requiring or permitting parameters. One solution would be to pass a tuple containing module name and parameters. Care needs to be taking in asserting the validity of parameters.
- As before, the converter will assert that the module exists and create an appropriate `Module` class object.

4. Run the conversion.

The code showcasing the suggested workflow can be found in the main function of the `FieldConvert.py` file.

24.2.3 Conversion process

24.3 Further development and improvement

Bibliography

[1] std::shared_ptr - cppreference.com. [Accessed 21 March 2018].

[2] boost.python/howto - python wiki, 2015. [Accessed 1st May 2018].

[3] osgboostpython/manualwrappingrationale.md at wiki · skylark13/osgboostpython · github, 2015. [Accessed 7th May 2018].

[4] osgboostpython/wrappingcookbook.md at wiki · skylark13/osgboostpython · github, 2015. [Accessed 7th May 2018].

[5] Github - tng/boost-python-examples: Some examples for the use of boost::python, 2016. [Accessed 7th May 2018].

[6] Mark Ainsworth. Pyramid algorithms for bernstein-bézier finite elements of high, nonuniform order in any dimension. *SIAM Journal of Scientific Computing*, 36:A543–A569, 2014.

[7] Mark Ainsworth, Gaelle Andriamaro, and Oleg Davydov. Bernstein-bézier finite elements of arbitrary order and optimal assembly procedures. *SIAM Journal of Scientific Computing*, 33:3087–3109, 2011.

[8] R. Aris. *Vectors, tensors, and the basic equations of fluid mechanics*. Dover Pubns, 1989.

[9] Ivo Babuska, Barna A Szabo, and I Norman Katz. The p-version of the finite element method. *SIAM journal on numerical analysis*, 18(3):515–545, 1981.

[10] Wolfgang Bangerth, Ralf Hartmann, and Guido Kanschat. deal.II–a general-purpose object-oriented finite element library. *ACM Transactions on Mathematical Software (TOMS)*, 33(4):24, 2007.

[11] Hugh M Blackburn and SJ Sherwin. Formulation of a galerkin spectral element–fourier method for three-dimensional incompressible flows in cylindrical geometries. *Journal of Computational Physics*, 197(2):759–778, 2004.

[12] A. Bolis, C.D. Cantwell, R.M. Kirby, and S.J. Sherwin. h-to-p efficiently: Optimal implementation strategies for explicit time-dependent problems using the spectral/hp element method. *International Journal for Numerical Methods in Fluids*, 75:591–607, 2014.

[13] A.I. Borisenko, I.E. Tarapov, and R.A. Silverman (Translator). *Vector and Tensor Analysis with Applications*. Dover Books on Mathematics, 2012.

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

[15] C.D. Cantwell, D. Moxey, A. Comerford, A. Bolis, G. Rocco, G. Mengaldo, D. de Grazia, S. Yakovlev, J-E Lombard, D. Ekelschot, B. Jordi, H. Xu, Y. Mohamied, C. Eskilsson, B. Nelson, P. Vos, C. Biotto, R.M. Kirby, and S.J. Sherwin. Nektar++: An open-source spectral/hp element framework. *Computer Physics Communications*, 192:205–219, 2015.

[16] C.D. Cantwell, S.J. Sherwin, R.M. Kirby, and P.H. Kelly. From h-to-p efficiently: Selecting the optimal spectral/hp discretisation in three dimensions. *Math. Model. Nat. Phenom.*, 6(3):84–96, 2011.

[17] C.D. Cantwell, S.J. Sherwin, R.M. Kirby, and P.H.J. Kelly. From h-to-p efficiently: Strategy selection for operator evaluation on hexahedral and tetrahedral elements. *Computers and Fluids*, 43:23–28, 2011.

[18] C.D. Cantwell, S. Yakovlev, R.M. Kirby, N.S. Peters, and S.J. Sherwin. High-order continuous spectral/hp element discretisation for reaction-diffusion problems on a surface. *Journal of Computational Physics*, 257:813–829, 2014.

[19] C. Canuto, M.Y. Hussaini, A. Quarteroni, and T.A. Zang. *Spectral Methods in Fluid Mechanics*. Springer-Verlag, New York, 1987.

[20] Bernardo Cockburn, George Karniadakis, and Chi-Wang Shu. *Discontinuous Galerkin Methods: Theory, Computation and Applications*. Springer-Verlag, 2000.

[21] J. Austin Cottrell, Thomas J. R. Hughes, and Yuri Bazilevs. *Isogeometric Analysis: Toward Integration of CAD and FEA*. John Wiley and Sons, 2009.

[22] Abrahams D. de Guzman J. Boost.python tutorial - 1.67.0, 2018. [Accessed 1st May 2018].

[23] Andreas Dedner, Robert Klöfkorn, Martin Nolte, and Mario Ohlberger. A generic interface for parallel and adaptive discretization schemes: abstraction principles and the DUNE-FEM module. *Computing*, 90(3-4):165–196, 2010.

[24] James W. Demmel. *Applied Numerical Linear Algebra*. SIAM, Philadelphia, PA, USA, 1997.

[25] M.O. Deville, P.F. Fisher, and E.H. Mund. *High-Order Methods for Incompressible Fluid Flow*. Cambridge University Press, 2002.

[26] Julia Docampo-Sánchez, Jennifer K Ryan, Mahsa Mirzargar, and Robert M Kirby. Multi-dimensional filtering: Reducing the dimension through rotation. *SIAM Journal on Scientific Computing*, 39(5):A2179–A2200, 2017.

[27] M. Dubiner. Spectral methods on triangles and other domains. *J. Sci. Comp.*, 6:345, 1991.

[28] M.G. Duffy. Quadrature over a pyramid or cube of integrands with a singularity at a vertex. *SIAM J. Numer. Anal.*, 19:1260, 1982.

[29] Paul Fischer, James Lottes, Stefan Kerkemeier, Oana Marin, Katherine Heisey, Aleks Obabko, Elia Merzari, and Yulia Peet. *Nek5000 User Manual*. ANL/MCS-TM-351, 2014.

[30] Python Software Foundation. Memory management - python 2.7.14 documentation. [Accessed 21 March 2018].

[31] Python Software Foundation. Reference counting - python 2.7.14 documentation. [Accessed 21 March 2018].

[32] D. Funaro. *Polynomial Approximations of Differential Equations: Lecture Notes in Physics, Volume 8*. Springer-Verlag, New York, 1992.

[33] F.X. Giraldo, J.F. Kelly, and E.M. Constantinescu. Implicit explicit formulations of a three dimensional non-hydrostatic unified model of the atmosphere (NUMA). *SIAM Journal of Scientific Computing*, 35:1162–1194, 2013.

[34] van Rossum G. Goodger D. Pep 257 – docstring conventions, 2001. [Accessed 16th May 2018].

[35] Michael T. Heath. *Scientific Computing: An Introductory Survey*. McGraw-Hill Companies, 2002.

[36] Jan Hesthaven, Sigal Gottlieb, and David Gottlieb. *Spectral Methods for Time-Dependent Problems*. Cambridge University Press, 2007.

[37] Jan S Hesthaven and Tim Warburton. *Nodal discontinuous Galerkin methods: algorithms, analysis, and applications*, volume 54. Springer, 2007.

[38] J.S. Hesthaven. From electrostatics to almost optimal nodal sets for polynomial interpolation in a simplex. *SIAM J. Numer. Anal.*, 35(2):655–676, 1998.

[39] J.S. Hesthaven and T.C. Warburton. *Nodal Discontinuous Galerkin Methods: Algorithms, Analysis, and Applications*. Springer Texts in Applied Mathematics 54. Springer Verlag: New York, 2008.

[40] T. J. R. Hughes. *The Finite Element Method*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1987.

[41] Ashok Jallepalli, Julia Docampo-Sánchez, Jennifer K Ryan, Robert Haimes, and Robert M Kirby. On the treatment of field quantities and elemental continuity in FEM solutions. *IEEE Transactions on Visualization and Computer Graphics*, 24(1):903–912, 2017.

[42] Cem Kaner, Jack Falk, and Hung Quoc Nguyen. *Testing Computer Software*. John Wiley & Sons, 2010.

[43] George Em Karniadakis and Robert M. Kirby. *Parallel Scientific Computing in C++ and MPI*. Cambridge University Press, New-York, NY, USA, 2003.

[44] George Em Karniadakis and Spencer J. Sherwin. *Spectral/hp element methods for Computational Fluid Dynamics (Second Edition)*. Oxford University Press, 2005.

[45] Robert M. Kirby and Spencer J. Sherwin. Aliasing errors due to quadratic nonlinearities on triangular spectral/hp element discretisations. *Journal of Engineering Mathematics*, 56:273–288, 2006.

[46] David A. Kopriva. *Implementing Spectral Methods for Partial Differential Equations: Algorithms for Scientists and Engineers*. Springer, 2009.

[47] Anders Logg, Kent-Andre Mardal, and Garth Wells (editors). *Automated Solution of Differential Equations by the Finite Element Method*. Springer Lecture Notes in Computational Science and Engineering, Volume 84, 2012.

[48] Daconta M. *C++ pointers and dynamic memory management*. New York: Wiley, 1995.

[49] Reddy M. *API Design for C++*. Burlington: Elsevier, 2011.

[50] A.T.T. McRae, G.-T. Bercea, L. Mitchell, D.A. Ham, and C.J. Cotter. Automated generation and symbolic manipulation of tensor product finite elements. *SIAM Journal on Scientific Computing*, 38(5):S25–S47, 2016.

[51] Scott Meyers. *Effective C++: 55 Specific Ways to Improve Your Programs and Designs (Third Edition)*. Addison-Wesley Professional, 2005.

[52] Jaroszyński P. Piotr jaroszyński's blog: Boost.python: docstrings in enums, 2007. [Accessed 16th May 2018].

[53] Jhong E. et al. Patel A., Picard A. Google python style guide. [Accessed 16th May 2018].

[54] Ch. Schwab. *p– and hp– Finite Element Methods: Theory and Applications in Solid and Fluid Mechanics*. Oxford University Press, 1999.

[55] Bjarne Stroustrup. *The C++ Programming Language (Fourth Edition)*. Addison-Wesley Professional, 2013.

[56] M. Taylor and B.A. Wingate. The fekete collocation points for triangular spectral elements. *Journal on Numerical Analysis*, 1998.

[57] M. Taylor, B.A. Wingate, and R.E. Vincent. An algorithm for computing Fekete points in the triangle. *SIAM J. Num. Anal.*, 38:1707–1720, 2000.

[58] Lloyd N. Trefethen. Is gauss quadrature better than clenshaw-curtis? *SIAM Review*, 50:67–87, 2008.

[59] Lloyd N. Trefethen and III David Bau. *Numerical Linear Algebra*. SIAM, Philadelphia, PA, USA, 1997.

[60] Tomáš Vejchodskỳ, Pavel Šolín, and Martin Zítka. Modular *hp*-FEM system HERMES and its application to Maxwell's equations. *Mathematics and Computers in Simulation*, 76(1):223–228, 2007.

[61] Peter E. J. Vos, Spencer J. Sherwin, and Robert M. Kirby. h-to-p efficiently: Implementing finite and spectral/hp element methods to achieve optimal performance for low- and high-order discretisations. *Journal of Computational Physics*, 229:5161–5181, 2010.

[62] Peter E.J. Vos, Sehun Chun, Alessandro Bolis, Claes Eskilsson, Robert M. Kirby, and Spencer J. Sherwin. A generic framework for time-stepping pdes: General linear methods, object-oriented implementations and applications to fluid problems. *International Journal of Computational Fluid Dynamics*, 25:107–125, 2011.

[63] F.D. Witherden, P.E. Vincent, and A. Jameson. Chapter 10 – high-order flux reconstruction schemes. *Handbook of Numerical Analysis*, 17:227–263, 2016.

[64] FR Witherden, AM Farrington, and PE Vincent. PyFR: An open source framework for solving advection-diffusion type problems on streaming architectures using the flux reconstruction approach. *Computer Physics Communications*, 185:3028–3040, 2014.