A Programmer's Guide to Nektar++

Developer's Guide

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

January 23, 2019

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

Contents

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 to allow extensions 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 to 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. Dave 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 Research Fellow in the Department of Aeronautics at Imperial College London (UK).

David Moxey: Dr. David Moxey is a Lecturer in Mechanical 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

The current effort, *Nektar++* [13], 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 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. The software supports a variety of discretization techniques and implementation strategies [53, 15, 14, 11], 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.

Nektar++ is a cross-platform spectral/*hp* element framework which aims to make high-order finite element methods accessible to the broader community. This is achieved by providing a structured hierarchy of C++ components, encapsulating the complexities of these methods, which can be readily applied to a range of application areas. These components are distributed in the form of cross-platform software libraries, enabling 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 2048-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 early on spent a lot of time considering what things we wanted to be distinctive about *Nektar++* and also what guiding principles would we use to help set 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. The first three boldface items below denote the three major themes of our work (i.e. respecting reason (1) above) and the subsequent items denotes the guardrails (i.e. respecting reason (2) above) that we put in place to help guide our efforts.

Principles: The following are our three guiding principles for *Nektar++*:

- **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" (pseodspectral) orders of 16^{th} degree. As part of our early work [53], 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 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 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, 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 what is best in terms of discretization, polynomial order, algorithm choice, etc. We want to 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 or 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 build around or on top of our basic spectral/*hp*framework. For others, the heart of *Nektar++* are its solver – 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 with *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 part of the library contains all the basic mathematical and computer science building blocks of the *Nektar++* code.

StdRegions: This part of the 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 regions over which we support differentiation and integration.

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

LocalRegions: This part of the library contains objects that express "local region" data and operations. 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.

Collections: This part of the library we amalgamate the action of key operators on multiple (standard region and 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 part of the library holds the mapping data structures used to assemble local regions into multi-regions.

MultiRegions: This part of the 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. Once you then define how these elements interact, you implicitly are defining the function space they represent and/or the approximation method. It is at this point in the library that a collection of (local region) elements can be thought of as representing a dG or cG field.

NekMeshUtils: This part of the library contains various utilities are beneficial to our NekMesh efforts.

SolverUtils: This part of the library contains various utilities that aid in the pre- and post-processing tasks related to our solvers.

Timings: This part of the library contains various timing utilities.

UnitTests: This part of the library contains unit tests that allow us to test basic functionality within *Nektar++*. These are useful for verifying that new additions or modifications to the code to 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 directory, in which we define our canonical standard regions. There are seven fundamental regions: 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 StdQuad is a region defined on $[-1, 1] \times [-1, 1]$ over which we can build approximations $\Psi(\xi_1, \xi_2)$. Typically Ψ 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 Ψ lives on $E_S = [-1, 1] \times [-1, 1]$ (i.e. $\Psi : E_S \to \mathbb{R}$) 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 is contained within the StdRegions directory (and will be discussed in Chapter

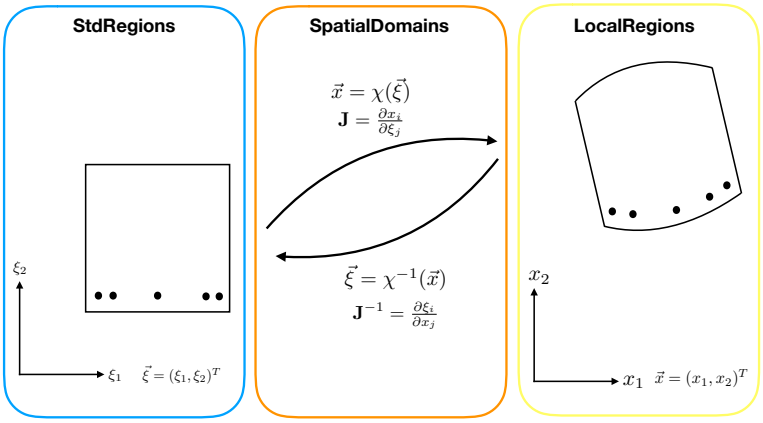


Figure 2.1: Diagram showing the LocalRegion (yellow) *is-a* StdRegion (blue) which *has-a* SpatialDomain (orange) in terms of coordinate systems and mapping functions. This diagram highlights how points (positions in space) are mapped between different (geometric) regions.

2). A local region expansion, such as a QuadExpansion, 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 4). Using the inheritance language of C++, we would say that a local region *is-a* standard region and *has-a* spatial domain object. The SpatialDomains directory contains information that expresses the mapping function $\chi(\cdot)$ from the standard region to a local region. SpatialDomains is explored in Chapter 3. In the case of our quad example, the SpatialDomain object held by a QuadExpansion 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 lived). If E denotes our geometric region is world space and if $F : E \to \mathbb{R}$ build upon polynomials over its standard region, then we obtain $F(x_1, x_2) = \Psi(\chi^{-1}(x_1, x_2))$. Note that even though Ψ is polynomial and χ is polynomial, the composition using the inverse of χ 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 6) 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.

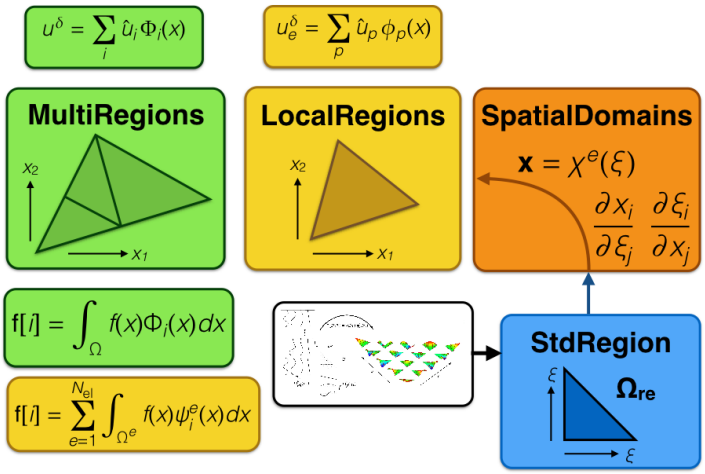


Figure 2.2: Diagram showing how a MultiRegion (green) contains a collection of LocalRegions (yellow), where a LocalRegion *is-a* StdRegion (blue) which *has-a* SpatialDomain (orange) in terms of coordinate systems and mapping functions. This diagram highlights the expansions of the solution formed over 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) [36, 47, 8] and more generally the mathematical ideas surrounding continuous Galerkin (cG) methods. This includes basic calculus

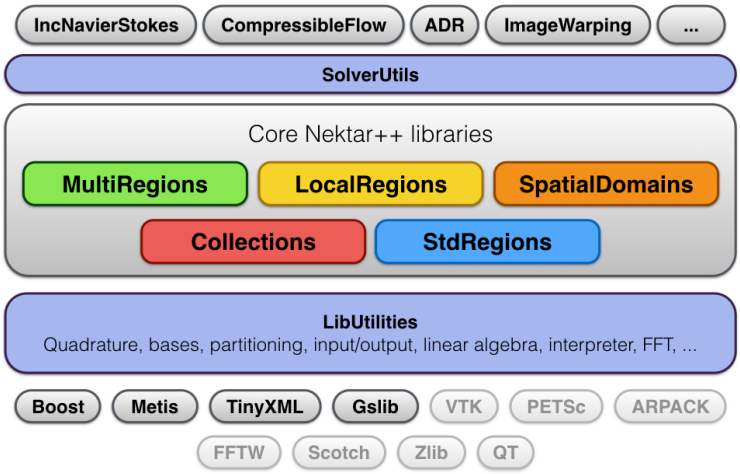


Figure 2.3: Diagram showing the top-down perspective on library components. Various solvers (along the top) can capitalize on generic SolverUtils and are built upon the core *Nektar++* libraries consisting of MultiRegions (green), Collections (red), LocalRegions (yellow), SpatialDomains (orange) and 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.

of variation concepts, basic partial differential equation knowledge, and general forms of discretization and approximation.

2. Polynomial Methods [17, 28, 32], and in particular concepts surrounding polynomial spaces, basis functions and numerical differentiation and quadrature.

3. Spectral Element Methods (SEM) [22, 39].

4. Discontinuous Galerkin Methods [18, 35].

5. Scientific Computing [31, 38]. 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 [51, 21]. 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 [10] 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 [25] 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) [29] 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 [52] 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 [20] and deal.II [9]. FEniCS [41] 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 [33], and PyFR [54], 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

3.1.1 General Resources

Mailing List

Please make sure that you sign up for the *Nektar++* mailing list:

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

While it is labeled as a *users* list, 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.

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 host of other *Nektar++* users.

3.1.2 Version Control: GIT

Anonymous Access

Nektar++ uses the *git* distributed version control system and is available for anonymous download using this command line (assuming you have the *git* program installed on your machine):

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

Until you need to contribute code back to the project, the only other *git* command you need to know is how to update your local code with the latest version of the trunk code:

```
> git pull
```

Collaborative Access

However, if you plan to work with the *Nektar++* community, you will need to have a reasonable understanding of the *git* version control 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`).

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 create an account on the *Nektar++ gitlab* site: `https://gitlab.nektar.info`.

Code contribution follows three basic steps: 1) Create an *issue* to describe the code updates you are making, 2) Branch the *Nektar++* code trunk and make your changes on that branch, and 3) Submit a request that your updates be merged into the trunk. A summary of these steps is found below. (For more information you may also refer to:

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 creat an *issue* statement that describes the proposed additions, changes, updates,

etc. This can be done here:

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

Please 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.

- Branches - The second step is to create a *git* branch in which to do the actual code developement. This can be done from the Nektar gitlab website: https://gitlab.nektar.info/nektar/nektar/branches Press the "New Branch" button. Please see the branch naming conventions found here: https://gitlab.nektar.info/nektar/nektar/blob/master/CONTRIBUTING.md Once you have created the branch (using the Nektar gitlab site), you should check it out to your local machine:

```
> git clone https://gitlab.nektar.info/nektar/nektar.git my_branch_id
```

Note, in the above command, we have named the directory in which *git* will place the code as "my_branch_id". However, at this point, that directory is still pointing at the repository master.

```
> git status
On branch master
```

To begin development, you must first switch to the appropriate branch:

```
> git checkout my_branch_id
> git status
On branch my_branch_id
```

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 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 # Display any modifcations to the file.
```

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".

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

Commits the file to your local repository.

```
> git push
```

Sends the file to the remote server.

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.

Once all of your modifications have been implemented, tested, and committed to your branch, it will be time to move on to the final step.

- Merge Requests - 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`

3.1.3 Managing the Build Process: 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.

3.2 C++11 Features

This section highlights some of the C++ 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 serveral practices that may be only familiar to programmers who have developed code using very advanced C++ features. Below we give a short summary of these features in order to provide a starting point when working with these features. We begin with more well known features and end with some advanced techiniques. Note, it is not the purpose of the following sections to cover in detail each of these concepts, but instead to give a brief overview of these important concepts such that the developer may look to other, in depth, sources if they are not familiar with the concepts.

- Namespaces - Many C++ software projects place their code in a namespace. However, it is important to note that *Nektar++* uses two nested namespaces in most (but not all) cases. The top level namespaces is always "Nektar", with the second level usually corresponding to the name of the subdirectory the code exists in. 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.

- C++ Standard Template Library (STL) - *Nektar++* uses of the C++ STL extensively. This consists of common classes such as map and vector, and also, with the move to C++11, 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 (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 nolonger 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 "weak_ptr"s. Dynamic casting allows for safely converting one type of variable into its base type (or vice versa). For example:

```
std::shared_ptr<FilterCheckpointCellModel> sptr =
          std::dynamic_pointer_cast<FilterCheckpointCellModel>( 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 when they inherit from "std::enable_shared_from_this". This is necessary for any class that is to be managed by C++ smart pointers, and allows the use of the class' (inherited) function "shared_from_this()" which returns a shard pointer to the class object.

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.

- typedefs - Like most other large code basis, *Nektar++* uses "typedefs" to create 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.

  ```
  typedef std::shared\_ptr<Expansion> ExpansionSharedPtr;
  ```

  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).

- Forward Declarations - There are two ways that an existing class type can be used when creating a new class in a header file. The existing class can either be declared, or completely defined by the time that it is needed. In other words, if all that is needed in the new class is a variable of the type of the existing class, then the existing class must only be declared. If functions on that variable must be called

(within the new header file), then the existing class must be fully defined by the point it is used. In order to reference a class in a header file, only the name of the class must be known to the compiler at the point it is to be referenced, but to use the existing class (actually call functions on an object of the class' type) the class must be defined.

The reason to just reference the class is that the header file does not need to #include the entire existing class. This allows for cleaner header files and faster compilations. The method in which one allows for the existing class to be referenced (without #including it) is to "Forward Declare" the class. This is done toward the top of the new class file in this manner:

```
class LinearSystem; \# Forward declare this class.
```

In this example, the compiler is told that there exists a class "LinearSystem", but at this point the compiler does not need further details. In the new class' header file, you can then create variables (or use them as parameters to functions) of this existing class ("LinearSystem") type.

- 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 one or more of the specific versions 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:

```
template<typename Dim, typename DataType> class Array; \\
template<typename DataType> class Array<OneD, const DataType> { \\
  // Explicit coding of class methods and variables specific to \\
  // this version of Array are found here. \\
};
```

In the above example, on the first line the generic templated "Array" class is declared. The second line shows an explicit instantiate of the Array class for a one dimensional (version of) "Array". When explicitly instantiating 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).

- 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 straigtforward 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 of them should actually be instantiated. For example:

```
class Expansion2D : virtual public Expansion, virtual public StdRegions::StdExpansio
```

  In the above example, if parent classes "Expansion" and "StdExpansion2D" both inherit from a shared grand parent, then the class "Expansion2D" (without the "virtual" keyword) would have two copies of the grandparent class, leading to strange and hard to debug issues in the code. Fortunately this is easily avoided by using (as seen above) the "virtual" keyword when specifying the inheritance hierarchy.

- Virtual Functions and Inheritance - Within *Nektar++*, classes that inherit from a parent class and override one of the parent class methods, use the "v_Function()" naming convention. This is a visual reminder that the function overrides a parent class function. Additionally, at the top level parent class you will find "Function()". As an example of this, let us consider a triangle element ("TriExp"). The "TriExp" class (eventually) inherits from the "StdExpansion" class. The "StdExpansion" has the "Integral()" function which is used to provide integration over an element. However, the only thing the "StdExpansion::Integral()" function does is call 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.

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

- Const keyword - While the "const" keyword is known to most C++ developers, it is used (as it should be) liberally in *Nektar++* for 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. If code is carefully written and correctly written, then marking a variable "const" will not be required for correct code execution. However, its use can substantially reduce accidental errors and allow for accelerated debugging. "Const" 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, 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.

- Function pointers and bind - Function pointers ("std::function") are similiar to pointers to data, except that they point to functions - and thus allow a function to be invoked indirectly (in other words, without explicilty 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 provied with a pointer to the function that will (later) be called to generate the required data. 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.

3.2.1 Factory Pattern

Nektar++ makes extensive use of the "Factory Pattern". "Factories" are objects used to create (ie: allocate) other objects (See the short discussion of "NekManager" above). More specifically, a "Factory" will create a new object of some "sub-class" type but return a "base class" pointer for 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 (*Nektar++* uses the factory function "CreateInstance()") is passed some information that details what to build; or 2) The factory may have some intrinsic knowledge detailing what objects to create.

Because *Nektar++* uses many factory types throughout the code base, a templated version has been implemented to provide the required functionality for these various factories. The template code for this class can be found in:

`LibUtilities/BasicUtils/NekFactory.hpp`

An example of the use of a *Nektar++* factory can be found in:

`library/NekMeshUtils/VolumeMeshing/VolumeMesh.cpp`

Specifically, in "VolumeMesh::Process()" the "CreateInstance()" function is used to create quadrilaterals.

3.3 Software Testing Approaches

3.3.1 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 [37]. 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 integrety 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/

3.3.2 Functional (Regression) Tests

Functional testing, which in *Nektar++* we refer to as *regression testing*, is the testing of software components based upon expected behavior. 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 [37].

Nektar++ uses a web-enabled "buildbot" site to show the results of recent test runs on multiple operating systems. The "buildbot" system can be used to manually instigate a *Nektar++* system test on a development branch of code. For more information, go to:

`http://buildbot.nektar.info`

Part I

Building-Blocks of Our Framework (Inside the Library)

Chapter 1

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.

1.1 BasicConst

```
static const NekDouble kNekUnsetDouble = -9999;
static const NekDouble kNekMinResidInit = 1e16;
static const NekDouble kVertexTheSameDouble  = 1.0e-8;
static const NekDouble kGeomFactorsTol = 1.0e-8;
static const NekDouble kNekZeroTol = 1.0e-12;
static const NekDouble kGeomRightAngleTol = 1e-14;
static const NekDouble kNekSqrtTol = 1.0e-16;
static const NekDouble kNekIterativeTol = 1e-09;
static const NekDouble kNekSparseNonZeroTol = 1e-16;

// Tolerances for mesh generation and CAD handling
static const NekDouble GeomTol = 1E-2;
static const NekDouble CoinTol = 1E-6;
```

1.2 BasicUtils

1.3 Communication

1.4 FFT

1.5 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 (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.

1.5.1 Points

- PointsKey
- Points

1.5.2 Basis

- BasisKey
- Basis

1.6 Interpreter

1.7 Kernel

1.8 Linear Algebra

1.9 Memory

1.10 Polylib

1.11 Time Integration

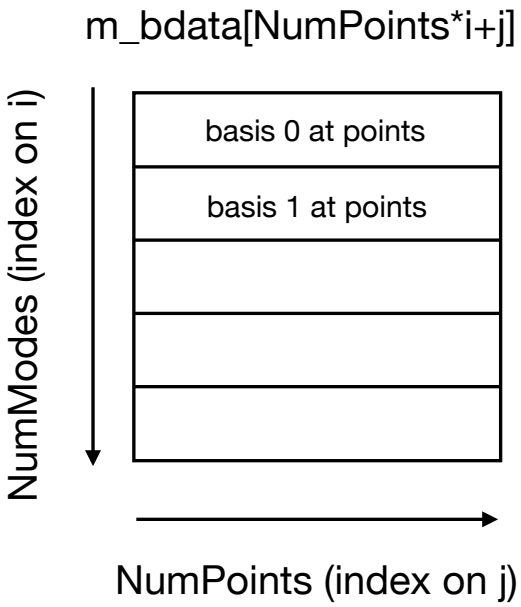


Figure 1.1: Figure 1

Chapter 2

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 [39] 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.

2.1 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 [17]. We show in Figure 2.1 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

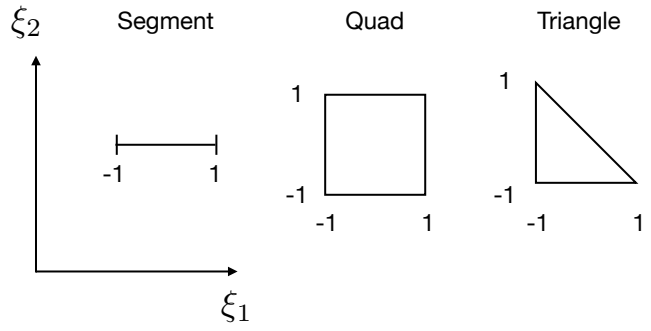


Figure 2.1: Example of the 1D and 2D reference space elements (segment, quad and triangle).

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

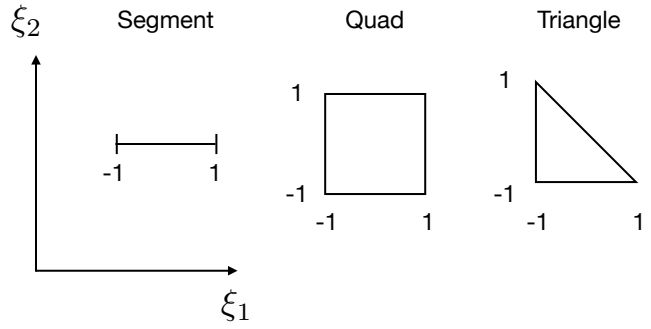


Figure 2.2: FIX THIS IMAGE WITH 3D: Example of the four three dimensional shapes: hexahedra, tetrahedra, prism and pyramid).

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.

2.1.1 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 [24], which was subsequently used by Dubiner [23] 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 2.3. 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.

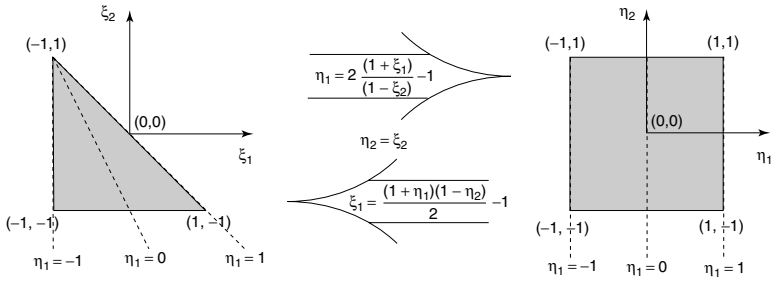


Figure 2.3: Duffy mapping between a right-sided triangle and a square domain.

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 2.4.

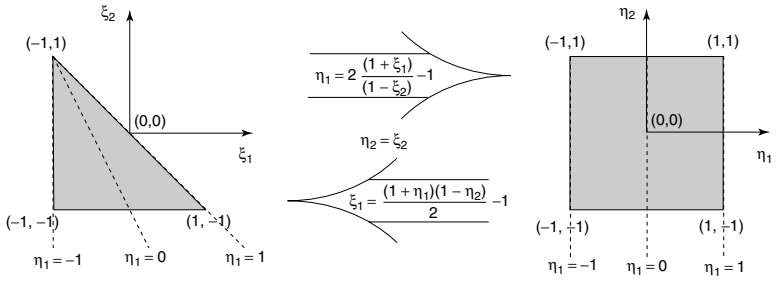


Figure 2.4: FIX THIS IMAGE WITH 3D: Duffy mapping application to generate all the 3D element types.

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 [39] and the references therein.

2.1.2 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 the electrostatic points of Hesthaven [34] and the Fekete points of Taylor and Wingate [49, 50].

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 [40]. 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.

2.2 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: [48, 44]. 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 ??, ?? and ?? [1].

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

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

Table 2.1: 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 2.2: 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 2.3: 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

minimize duplication of code). We present this class hierarchy in Figure 2.5.

As is seen in Figure 2.5, 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 2.6. 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.

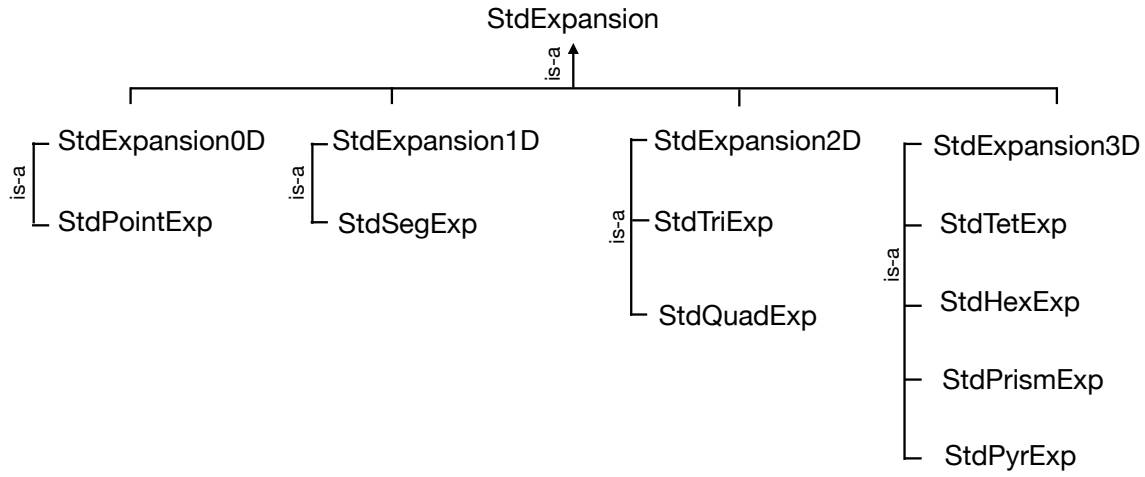


Figure 2.5: Class hierarchy derived from StdExpansion, the base class of the StdRegions Directory.

2.2.1 Variables at the Level of StdExpansion

Private:

Protected:

Public:

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

Private:

Protected:

Public:

2.2.3 Variables at the Level of Shape-Specific StdExpansions

Private:

Protected:

Public:

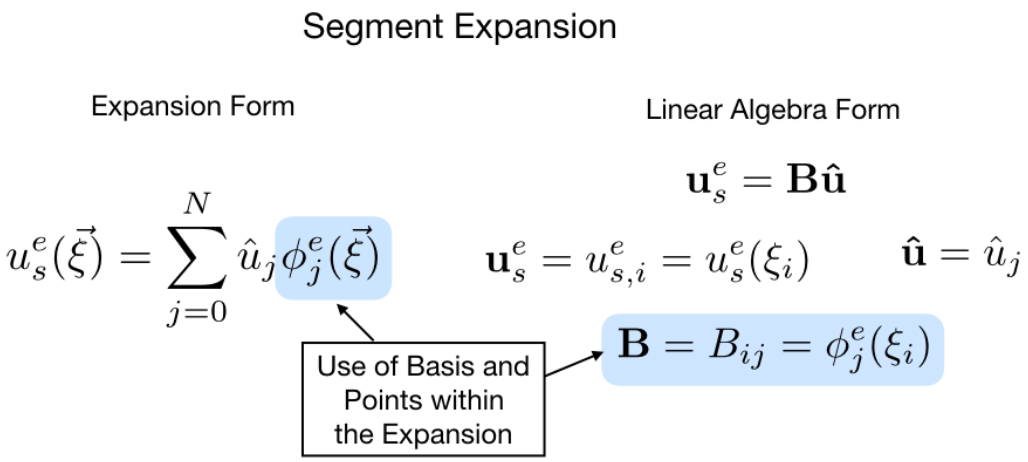


Figure 2.6: Diagram to help understand the various data members (object attributes) contained within StdRegions and how they connect with the mathematical representation presented earlier.

2.3 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 3

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 [39] 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 [39]. 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.

3.1 The Fundamentals Behind SpatialDomains

As mentioned in our discussions of the fundamentals of StdRegions (i.e. Section 2.1), 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 2. We refer to our world space regions as local regions, which we will present in Chapter 4. How these two are connected are via SpatialDomains.

As will be further discussed in Section 3.2, 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 [39] 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 [16] as our guide. The notation therein is slightly different than that of [39] 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 [16] and references therein as needed. For those unfamiliar with covariant and contravariant spaces, we encourage the reader to review [12].

3.1.1 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 3.1, 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 $\chi(\vec{\xi})$.

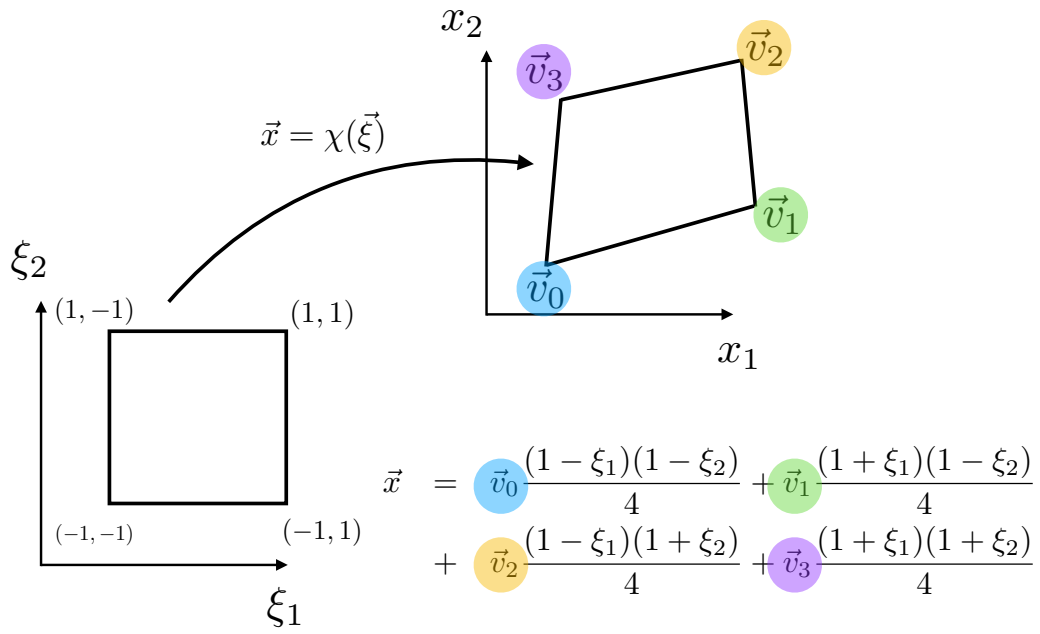


Figure 3.1: Reference space to world space mapping of a 2D quadrilateral to a straight-sided (2D) quadrilateral in the plane via a bi-linear (i.e., $Q(1)$) mapping.

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 [39]. 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 3.1, 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 3.2 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 3.2, 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 [39], 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 we infer (calculate) the modal values and store these values within SpatialDomain data

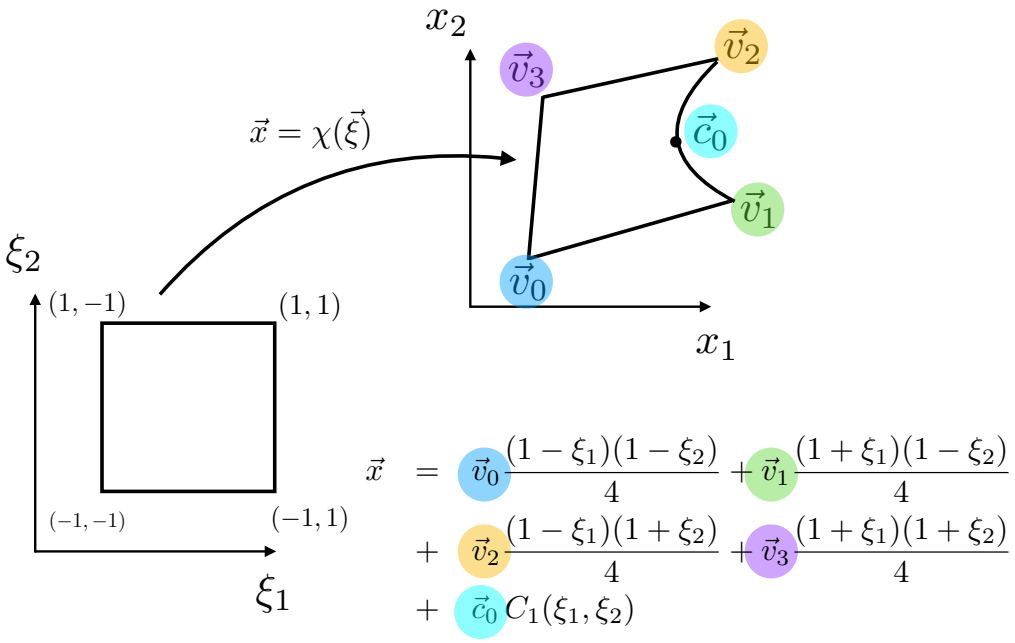


Figure 3.2: Reference space to world space mapping of a 2D quadrilateral to a curve (2D) quadrilateral in the plane via a bi-quadratic (i.e., $Q(2)$) mapping.

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.

3.1.2 Regular Mappings: Geometric Factors

$$\mathbf{J} = \frac{\partial x_i}{\partial \xi_j}$$

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

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

gmat and jac

note that gmat is a tensor defined at each point, and jac is the determinant of the jacobian.

3.1.3 Deformed Mappings: Geometric Factors

3.2 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 2.2. 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 3.3.

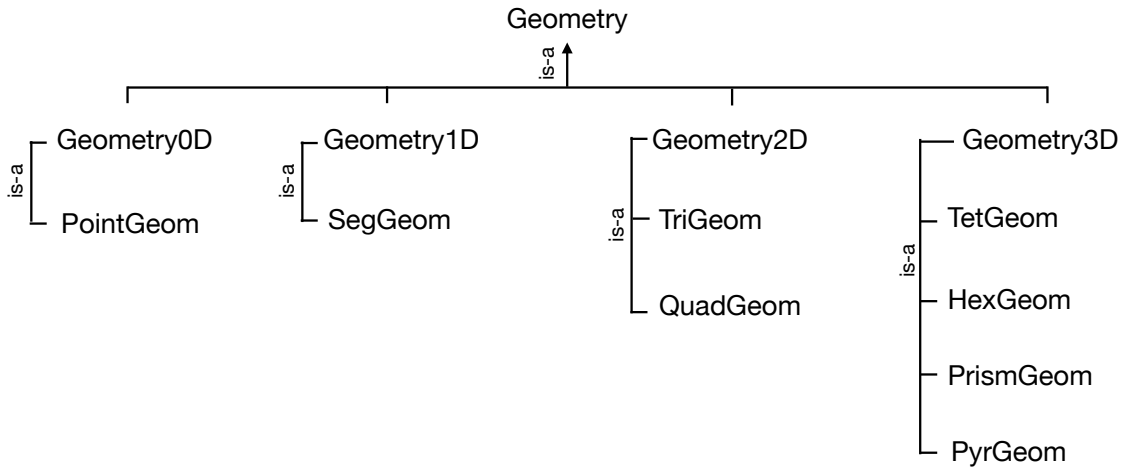


Figure 3.3: Class hierarchy derived from Geometry, the base class of the SpatialDomains Directory.

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.

3.2.1 Variables at the Level of Geometry

Private:

Protected:

Public:

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

Private:

Protected:

Public:

3.2.3 Variables at the Level of Shape-Specific Geometry Information

Private:

Protected:

Public:

3.2.4 Reference to World-Space Mapping

Geometry

GeomFactors

3.2.5 MeshGraph

3.3 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 4

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 [39] 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.

4.1 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.

4.2 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 2.2. 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 4.1.

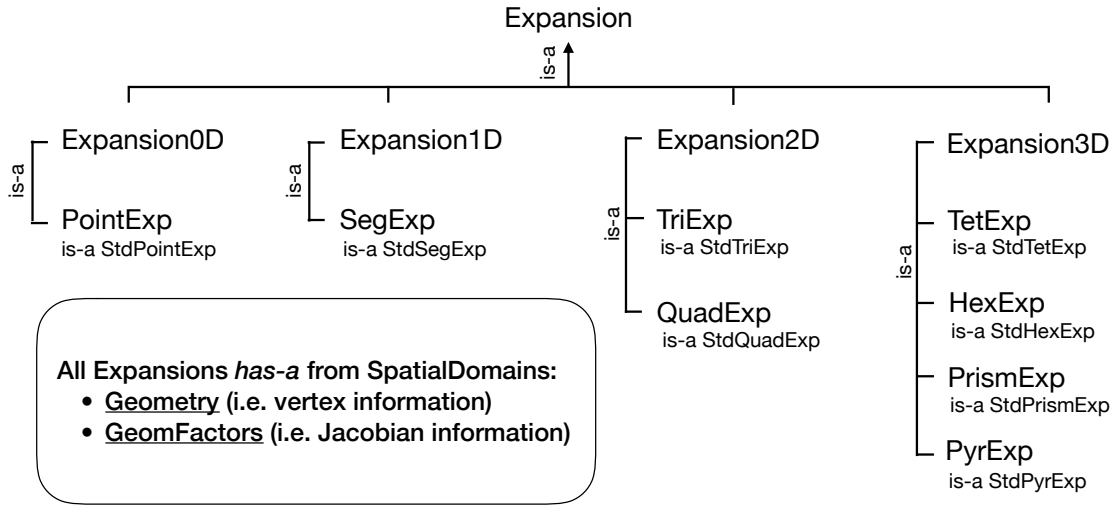


Figure 4.1: Class hierarchy derived from Expansion, the base class of the LocalRegions Directory.

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 2.6. 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.

4.2.1 Variables at the Level of Expansion

Private:

Protected:

Public:

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

Private:

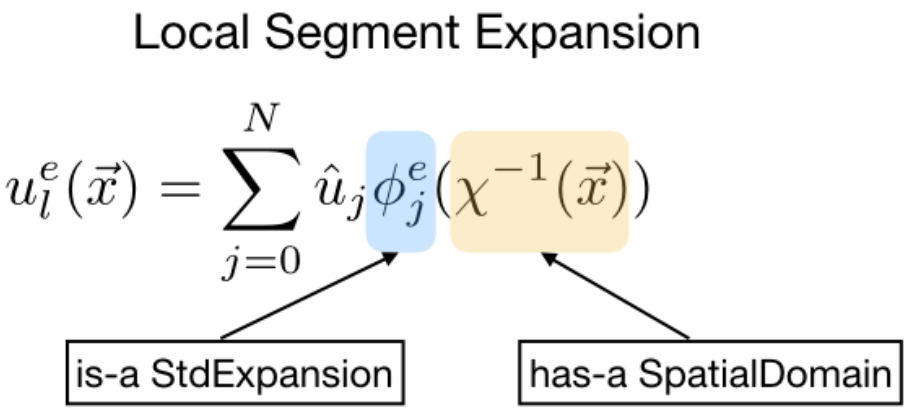


Figure 4.2: Diagram to help understand the various data members (object attributes) contained within LocalRegions and how they connect with the mathematical representation presented earlier. Recall that a LocalRegion *is-a* StdRegion and *has-a* SpatialDomain.

Protected:

Public:

4.2.3 Variables at the Level of Shape-Specific Expansions

Private:

Protected:

Public:

4.3 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 5

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 [39] 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.

5.1 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).

5.2 The Fundamental Data Structures within Collections

5.3 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 6

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 [39] 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.

6.1 The Fundamentals Behind MultiRegions

6.2 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 2.2. 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 6.1.

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

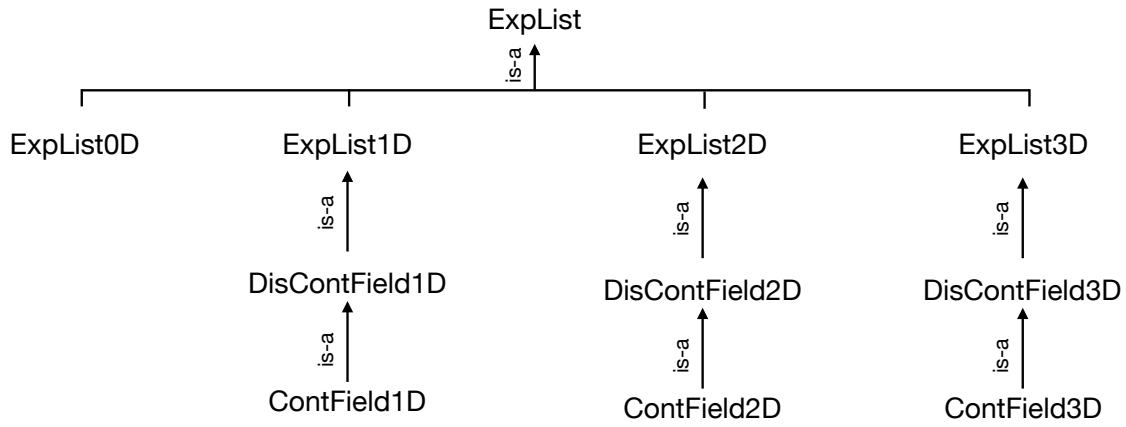


Figure 6.1: Class hierarchy derived from ExpList, the base class of the MultiRegions Directory.

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.

6.2.1 Variables at the Level of ExpList

Private:

Protected:

Public:

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

Private:

Protected:

Public:

6.2.3 Variables at the Level of Discontinuous Field Expansions

Private:

Protected:

Public:

6.2.4 Variables at the Level of Continuous Field Expansions

Private:

Protected:

6.3 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:

Chapter 7

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 the book by Karniadakis and Sherwin [39] as our principle reference. 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.

7.1 The Fundamentals Behind GlobalMapping

7.2 The Fundamental Data Structures within GlobalMapping

7.3 The Fundamental Algorithms within GlobalMapping

Chapter 8

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 [39] 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.

8.1 The Fundamentals Behind FieldUtils

8.2 The Fundamental Data Structures within FieldUtils

8.3 The Fundamental Algorithms within FieldUtils

Chapter 9

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 [39] 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.

9.1 The Fundamentals Behind SolverUtils

9.2 The Fundamental Data Structures within SolverUtils

9.3 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 1

ADRSolver: Solving the Advection-Reaction-Diffusion Equation

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

Chapter 2

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 3

CompressibleFlowSolver: Solving the Compressible Navier-Stokes Equations

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

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 1

FieldConvert

In this chapter, we walk the reader through FieldConvert.

Chapter 2

NekMesh

In this chapter, we walk the reader through NekMesh.

Part IV

NekPy: Python interface to *Nektar++*

Chapter 1

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.

1.1 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 1.1.

A typical snippet could look something like:

Listing 1.1: NekPy sample snippet

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

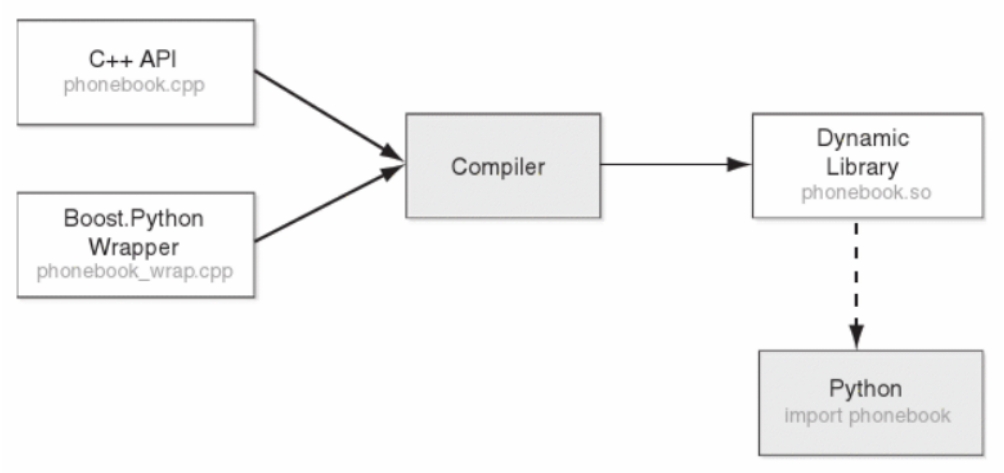


Figure 1.1: A schematic diagram of how C++ code is converted into Python with Boost.Python [43]

```
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 2

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.

2.1 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.

2.1.1 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
```

2.1.2 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
```

2.1.3 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
```

2.2 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 0
    x11005c310>
```

2.2.1 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 3

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 3.1 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`).

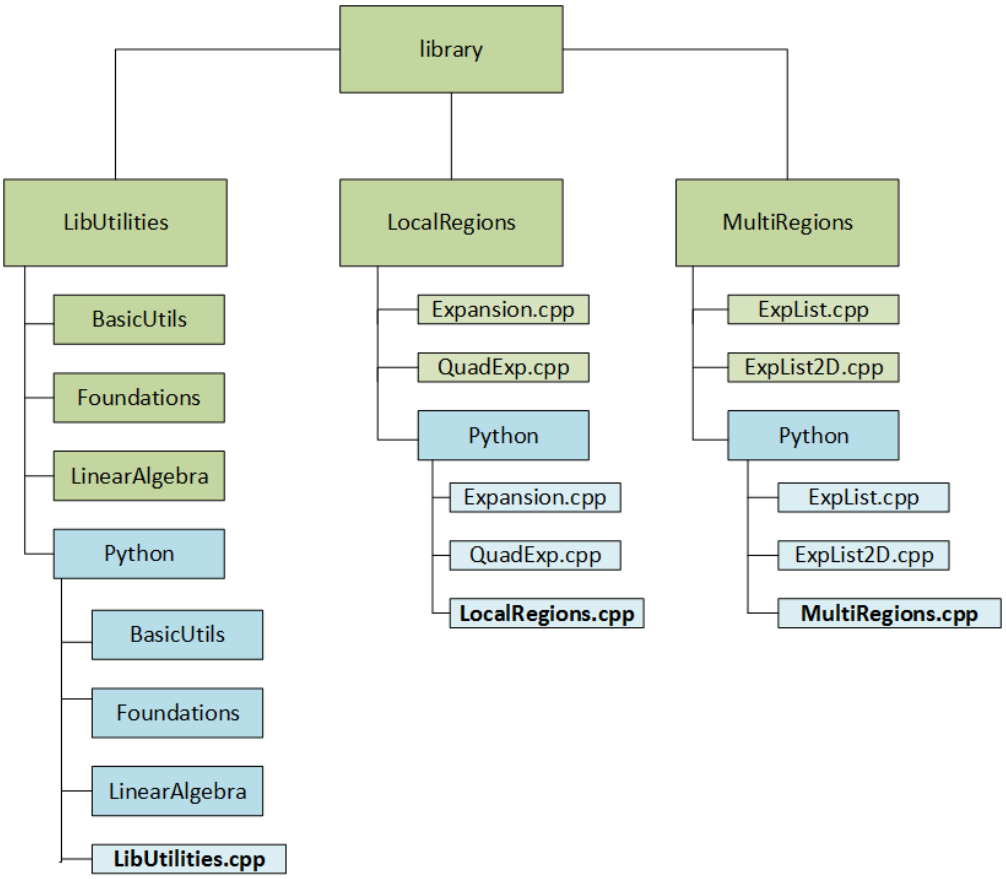


Figure 3.1: The location of Python wrapper files within Nektar++ structure.

Chapter 4

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 [19],
- 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.

4.1 Defining a library

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

Listing 4.1: 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.

4.2 Basic class wrapping

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

Listing 4.2: 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)
```

```
        ;
}
```

4.2.1 `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.

4.2.2 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 4.3: 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:

- NumPy arrays created from Array objects will share their memory, so that changing the C++ array changes the contents of the NumPy array.
- 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.

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

4.2.3 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 4.4: 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.

4.2.4 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 5

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 [30].

Boost.Python provides an easy way to include docstrings in the wrapped methods and classes as shown in Listing 5.1. The included docstrings will appear when Python `help` method is used.

Listing 5.1: Example of class and method documentation in Boost.Python

```
void export_Points()
{
        py::class_<PointsKey>("PointsKey",
        "Create␣a␣PointsKey␣which␣uniquely␣defines␣quadrature␣
            points.\n"
        "\n"
        "Args:\n"
        "\tnQuadPoints␣(integer):␣The␣number␣of␣quadrature␣
            points.\n"
        "\tpointsType␣(PointsType␣object):␣The␣type␣of␣
            quadrature␣points.",
```

```
        py::init<const int, const PointsType&>())

            .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 [45] was used, as shown in Listing 5.2. 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 5.2: 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 [46] 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 6

Memory management in NekPy

One of the biggest technical challenges of the NekPy project was to ensure the appropriate memory management between Python and C++. 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 outlines the basics of memory management in C++ and Python as well as the solutions devised to effectively use the resources when passing data between the different layers of the bindings.

6.1 Memory management in C++ and Python

6.1.1 C++

In C++ the memory used by the application is divided into five containers [42]:

- 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 6.1.

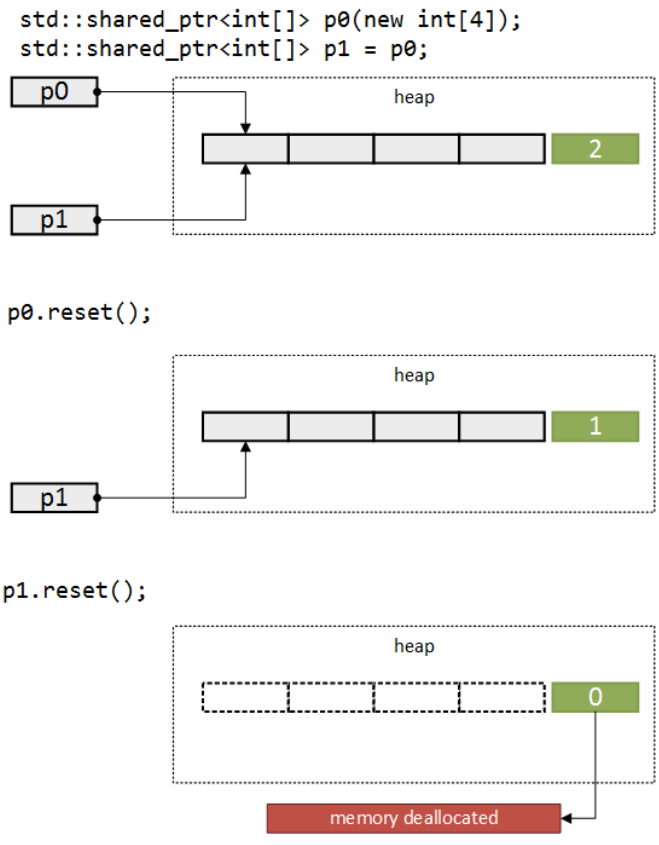


Figure 6.1: Memory management with C++11 `shared_ptr`. The two declared shared pointers reference an array of 4 integers. Reference counter shown in green.

Python

Python manages memory though a private heap containing all Python objects [26]. An internal memory manager is used to ensure the memory is properly allocated and dealocated while the script runs. In contrast to C++, Python 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 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. A disadvantage of this solution is slower execution time, since the garbage collector routines have to be called frequently. Features of Python Memory Manager are schematically shown in Figure 6.2

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, Python enables the programmer to increase or decrease the reference counter of the object using `Py_XINCREF` and `Py_XDECREF` respectively [27].

6.2 Passing C++ arrays to Python [1]

In many situations arrays created by Nektar++ (usually a `shared_ptr` of `NekMatrix<D, StandardMatrixTag>` type) in C++ need to be passed to Python - for instance, when preforming 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. Meanwhile, Python interface might still require the data contained within the array, only to be faced with an error as the data no longer exists in memory. To prevent such situations a solution employing reference counting must be used.

Converter method

Boost.Python provides the methods to do both convert a C++ type element to one recognised by Python as well as to maintain appropriate reference counting. Listing 6.1 shows 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 6.1: Converter method for converting the C++ arrays into Python NumPy arrays.

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

[1] The file implementing the below procedure: `LibUtilitiesPythonLinearAlgebraNekMatrix.cpp`

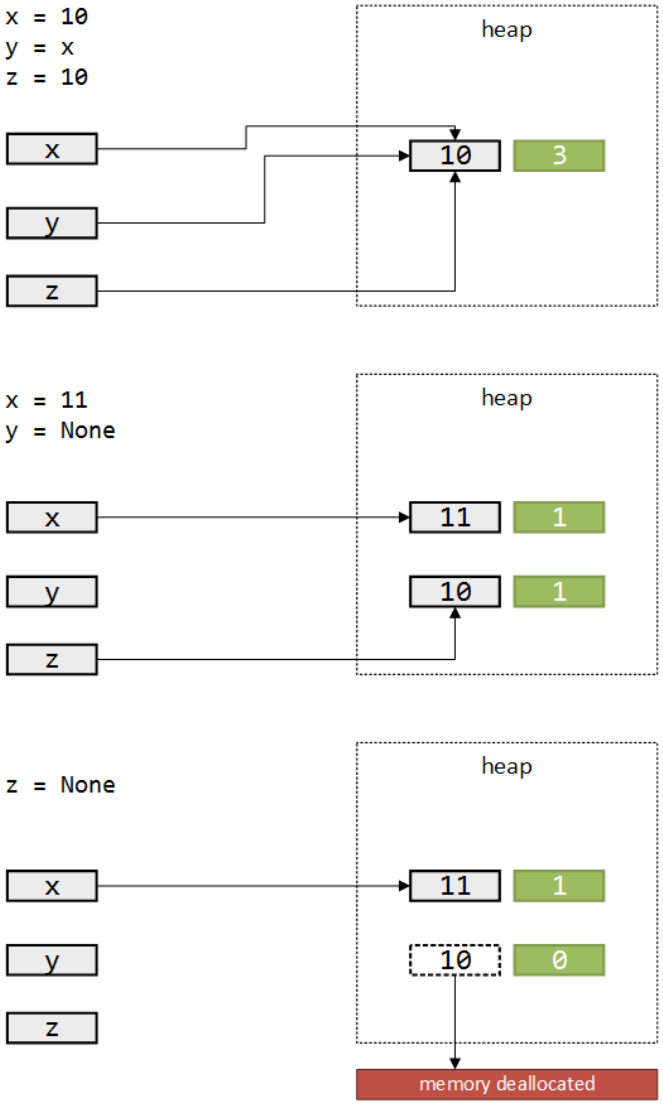
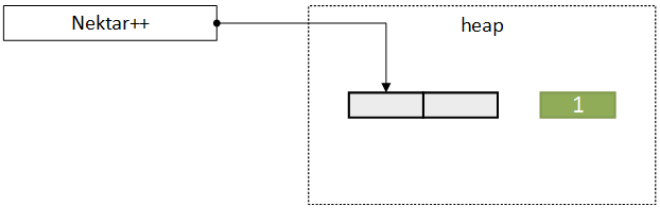


Figure 6.2: Memory management in Python. Memory is optimised as much as possible and garbage collector deallocates the memory if the reference counter (shown in green) reaches 0.

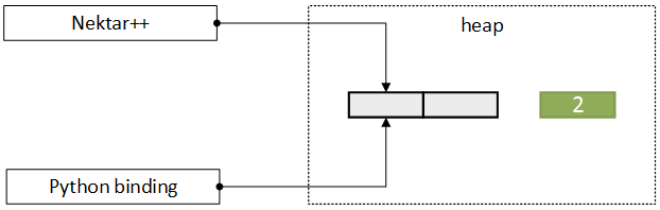
```
#include <LibUtilities/LinearAlgebra/MatrixStorageType.h>
#include <NekPyConfig.hpp>

using namespace Nektar;
using namespace Nektar::LibUtilities;

template<typename T, typename F>
void NekMatrixCapsuleDestructor(void *ptr) // destructor for
    the shared_ptr to data
```

```
{
    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)
    {
    py::object capsule(
            py::handle<>(PyCObject_FromVoidPtr(
                                new std::shared_ptr<NekMatrix<T,
                                    StandardMatrixTag>>(mat), //
                                    new shared_ptr to the data
                                NekMatrixCapsuleDestructor<T,
                                    StandardMatrixTag>))); //
                                    destructor method

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

    return py::incref( // increments Python reference counter
            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)
```

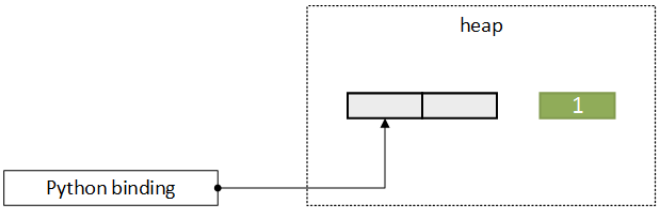
First, a new Python object is created, holding a shared pointer (`PyCObject_FromVoidPtr`) to the array held in memory, increasing its reference counter and preventing it from being prematurely deallocated, as shown in Figure 6.3, as well as destructor which contains instructions on what is to be done when the object is no longer needed (in this case, the `shared_ptr` is to be deleted). 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.



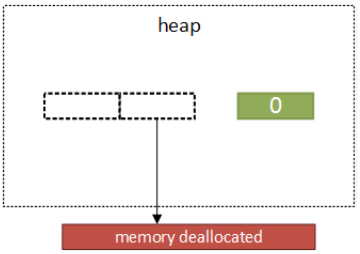
(a) Nektar++ creates a data array referenced by a `shared_ptr`



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

Then, a Boost.Python handle is created around the object, allowing Boost.Python to manage references correctly. Finally, a Boost.Python object named `capsule` is created from the handle.

The converter method returns a NumPy array – this was chosen as it is a well-known datatype used in Python's popular `numpy` package. Additionally, Boost.Python allows

the programmer to create a NumPy array from a generic C++ data container and to specify the owner of said container using the `np::from_data` method.

Care had to be taken to ensure that the array is presented in the correct order, especially as 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, b)` tuple, 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. Hence, the tuple passed was `(sizeof(T), nRows * sizeof(T))`, where `T` is the datatype of the array.

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.

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 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.

6.2.1 Passing Python data to C++ [2]

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_XINCREF` and `Py_XDECREF` methods respectively.

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.

[2] 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:

- `m_memory_pointer` - holding the pointer to the `PyObject` containing the data,
- `m_python_decrement` - holding the pointer to the callback function decrementing the reference counter of the `PyObject`.

If the two new attributes are not set to `nullptr` the array has been constructed throught the Python to C++ converter. The already existing `m_count` attribute was retained in order to keep track of how many C++ references to the array there are.

Adding new attributes to the arrays might cause a significantly increased memory usage. Therefore 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 6.2. `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 6.2: New constructor for initialising arrays created through the Python to C++ converter method.

```
Array(unsigned int dim1Size, DataType* data, void*
    memory_pointer, void (*python_decrement)(void *)) :
                m_size( dim1Size ),
                m_capacity( dim1Size ),
                m_data( data ),
                m_count( nullptr ),
                m_offset( 0 ),
                m_memory_pointer( memory_pointer ),
                m_python_decrement( python_decrement )
            {
                m_count = new unsigned int();
                *m_count = 1;
            }
```

Changes have also been made to the destructor, as shown in Listing 6.3, 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 6.3: The modified destructor for C++ arrays.

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

    *m_count -= 1;
    if( *m_count == 0 )
    {
        #ifdef WITH_PYTHON
        if (m_memory_pointer == nullptr)
        {
            ArrayDestructionPolicy<DataType>::
                Destroy( m_data, m_capacity );
            MemoryManager<DataType>::RawDeallocate(
                m_data, m_capacity );
        }
        else
        {
            m_python_decrement(m_data);
        }
        #else
        ArrayDestructionPolicy<DataType>::Destroy(
            m_data, m_capacity );
        MemoryManager<DataType>::RawDeallocate(
            m_data, m_capacity );
        #endif

        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. A new NumPy array object is created in Python.

2. The NumPy array object is passed as an argument to a C++ method available in the Python wrapper.
3. The converter method is called to convert a Python NumPy array into C++ `Array<OneD, const DataType>` object.
4. The converter creates a new `Array<OneD, const DataType>` object with the following attribute values:
   - `m_data` points to the data contained by the NumPy array,
   - `m_memory_pointer` points to the NumPy array object,
   - `m_python_decrement` points to the function decrementing the reference counter of the `PyObject`,
   - `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 6.4a and 6.4b.

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 6.5 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 6.4c. Similarly, if the C++ array is deleted first, the Python object will still exist as the reference counter will be non-zero (see Figure 6.4d).

Converter method

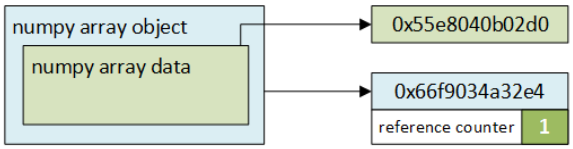
As with conversion from C++ to Python, a converter method was registered to make Ptyhon NumPy arrays available in C++ with Boost.Python. Listing 6.5 shows the converter methods with comments on its contents.

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.

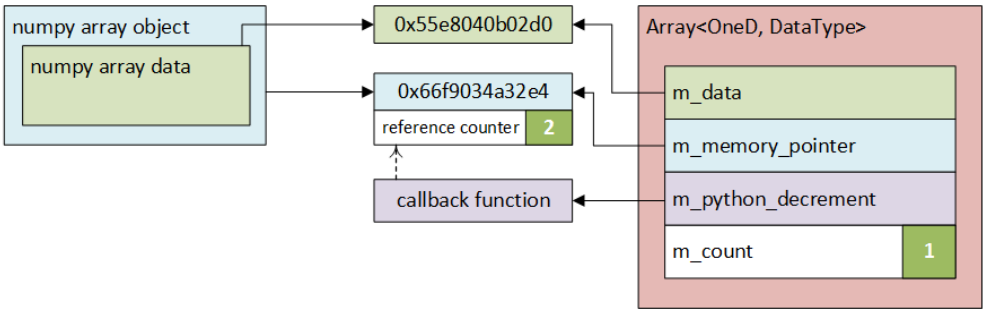
The callback decrement method is shown below in Listing 6.4. When provided with a pointer to PyObject it decrements it reference counter.

Listing 6.4: The decrement method called when the `m_count` of C++ array reaches 0.

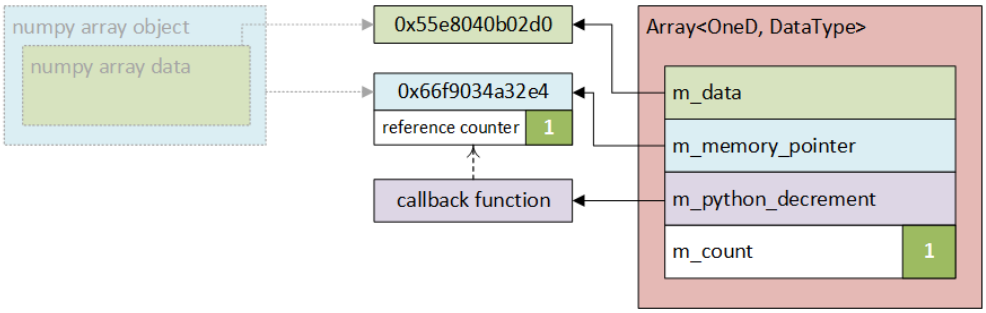
```
static void decrement(void *objPtr)
    {
        PyObject *pyObjPtr = (PyObject *)objPtr;
        Py_XDECREF(pyObjPtr);
    }
```



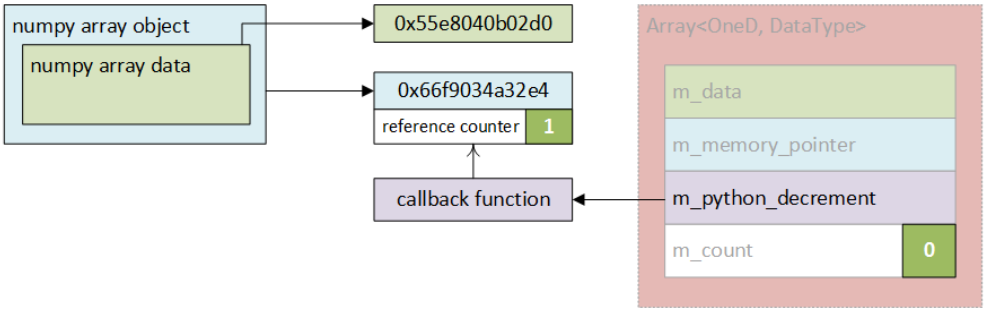
(a) The NumPy array is created in Python. Note that the NumPy object and the data it contains are represented by two separate memory addresses.



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

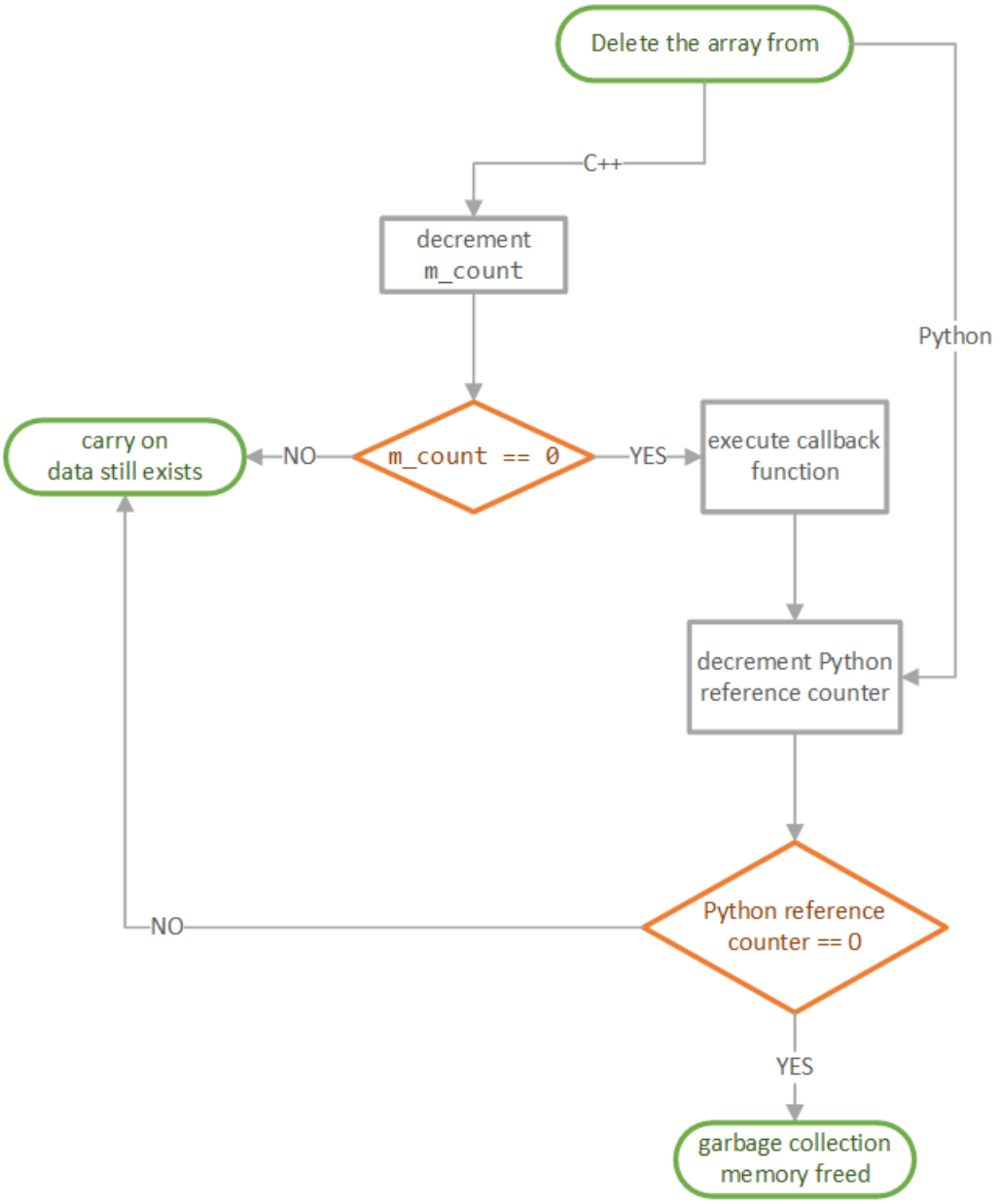


Figure 6.5: Flowchart describing the procedure for array deletion from either side of the code.

Listing 6.5: Converter method for converting the Python NumPy arrays into C++ arrays.

```
static void construct(
        PyObject *objPtr,
        py::converter::rvalue_from_python_stage1_data* data)
    {
        // Extract data from the PyObject.
        // This has to be a _borrowed_ reference, otherwise at
           the end of this
        // scope it seems memory gets deallocated
        py::object obj((py::handle<>(py::borrowed(objPtr))));
        np::ndarray array = py::extract<np::ndarray>(obj);

        // Get pointer to the memory into which the C++ type
           will be constructed
        void *storage = (
            (py::converter::rvalue_from_python_storage<Array<
               OneD, T> >*)data)
            ->storage.bytes;
        data->convertible = storage;

        // Grab the address of PyObject for C++ to use.
        void *memory_pointer = objPtr;

        // Remedies issues with using const datatypes.
        using nonconst_t = typename std::remove_const<T>::type;

        // Construct OneD array from numpy array.
        new (storage) Array<OneD, T>(array.shape(0), // array
           size
            (nonconst_t *)array.get_data(), // pointer to the
               data readable by C++
            memory_pointer, // memory address of the PyObject
            &decrement); // pointer to the callback decrement
               function
        // Increment the Python reference counter.
        Py_XINCREF(objPtr);
    }
```

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 7

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.

7.1 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.

7.2 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.

7.2.1 `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.

7.2.2 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.

7.2.3 Conversion process

7.3 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] 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.

[9] 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.

[10] 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.

[11] 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.

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

[13] 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.

[14] 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.

[15] 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.

[16] 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.

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

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

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

[20] 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.

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

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

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

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

[25] 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.

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

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

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

[29] 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.

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

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

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

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

[34] 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.

[35] 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.

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

[37] Cem Kaner, Jack Falk, and Hung Quoc Nguyen. *Testing Computer Software*. John Wiley & Sons, 2010.

[38] George Em Karniadakis and Robert M. Kirby. *Parallel Scientific Computing in C++ and MPI*. Cambridge University Press, New-York, NY, USA, 2003.

[39] George Em Karniadakis and Spencer J. Sherwin. *Spectral/hp element methods for Computational Fluid Dynamics (Second Edition)*. Oxford University Press, 2005.

[40] Robert M. Kirby and Spencer J. Sherwin. Aliasing errors due to quadratic non-linearities on triangular spectral/hp element discretisations. *Journal of Engineering Mathematics*, 56:273–288, 2006.

[41] 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.

[42] Daconta M. *C++ pointers and dynamic memory management*. New York: Wiley, 1995.

[43] Reddy M. *API Design for C++*. Burlington: Elsevier, 2011.

[44] Scott Meyers. *Effective C++: 55 Specific Ways to Improve Your Programs and Designs (Third Edition)*. Addison-Wesley Professional, 2005.

[45] Jaroszyński P. Piotr jaroszyński's blog: Boost.python: docstrings in enums, 2007. [Accessed 16th May 2018].

[46] Jhong E. et al. Patel A., Picard A. Google python style guide. [Accessed 16th May 2018].

[47] Ch. Schwab. *p– and hp– Finite Element Methods: Theory and Applications in Solid and Fluid Mechanics*. Oxford University Press, 1999.

[48] Bjarne Stroustrup. *The C++ Programming Language (Fourth Edition)*. Addison-Wesley Professional, 2013.

[49] M. Taylor and B.A. Wingate. The fekete collocation points for triangular spectral elements. *Journal on Numerical Analysis*, 1998.

[50] 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.

[51] Lloyd N. Trefethen and III David Bau. *Numerical Linear Algebra*. SIAM, Philadelphia, PA, USA, 1997.

[52] 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.

[53] 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.

[54] 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.