UNIVERSITY OF EXETER

Final Report

Expansion of Python Framework in Nektar++
Noah Yewman

3rd Year Individual Project

I certify that all material in this thesis that is not my own work has been identified and that no material has been included for which a degree has previously been conferred on me.

Signed.. [illegible]

College of Engineering, Mathematics, and Physical Sciences
University of Exeter

Final Report

ECM3175

Expansion of Python Framework in Nektar++

30/04/2020

Noah Yewman
BEng Mechanical Engineering
Student number: 670041990
Candidate number: 133819

Supervisor: Dr David Moxey

Abstract

Nektar++ is an open-source software which allows the user to generate and solve high-order computational fluid dynamics problems by making use of Spectral/hp element methods. Spectral/hp element methods provide significantly higher accuracy of results when running CFD simulations, with a reduced computational load due to the necessity of fewer elements. Nektar++ has a significant barrier to entry due to being written in C++, a language which has become a language which is complicated to understand, with complex syntax. To attempt to lower the barrier to entry of the software, and therefore Spectra/hp element methods, the Nektar++ Python library was expanded. Two programs were made which allow for users to use at different parts in a pipeline. Documentation was completed to provide information on the workings of the programs for both the user and developer.

Keywords: Nektar++, Python, High-Order Mesh Generation

Acknowledgements

I would like to extend my gratitude towards Dr David Moxey for his invaluable help in completing this project. Without his immense patience, answering my programming and Nektar++ related questions, completion of this project would have never been possible.

I would like to further thank all of my friends who have helped and encouraged me throughout my time at Exeter, answering countless stupid questions at all hours of the day and making my time at Exeter enjoyable.

Without the unwavering support and sacrifice of both of my parents throughout the entirety of my education I would never be in the position that I am today.

Finally, I would like to thank my beautiful girlfriend Katie who has supported me throughout my time at Exeter, staying up late nights with me to study and encouraging me through various episodes of me claiming coursework has been ‘impossible’. I will always be indebted for your support throughout university.

Table of Contents

1. Introduction and Background

Computational Fluid Dynamics (CFD) are invaluable within modern day engineering, allowing for engineers to efficiently and cheaply run simulations to gain valuable information about the characteristics and performance of a design. CFD is used widely in many different industries, from medicine to aerospace. With first and second order methods rapidly hitting a wall in terms of how much more can be done to develop them, more and more focus is turning to high-order methods, to gain more accurate simulation results at a faster speed. Currently high-order methods are difficult to use are not very accessible to somebody without advanced knowledge on the subject. Nektar++ attempts to break down this barrier to entry, with the aim of generating a high-order mesh on the command line using one executable to take a CAD file and turn it into a high-order mesh. The branch of Nektar++ which generates high order meshes is called NekMesh. Within this paper, the framework allowing for mesh generation to be performed simply will be expanded to further lower the barrier for entry of the program.

1.1. The Needs for a High-Order Mesh Generator

The majority of CFD programs make use of linear control volumes, resulting in elements with only straight edges. This means that the mesh which is on the geometry is not true to the geometry, it is an approximation. In second order codes, it is assumed that there will be a linear solution within each control volume which is discontinuous across control volume boundaries. Despite the approximation in the geometry, so long as the mesh is fine enough around the complex geometry (by using the h method) the results will be sufficiently accurate. However, by increasing the number of elements, the computational load increases exponentially, so the more logical method of increasing accuracy is by having higher order meshes, which would allow for fewer elements around complex geometry which would also allow for the mesh to map almost identically to the geometry with a much more coarse mesh. Figure 1.1 shows the difference between the error against degrees of freedom for both h and p methods.

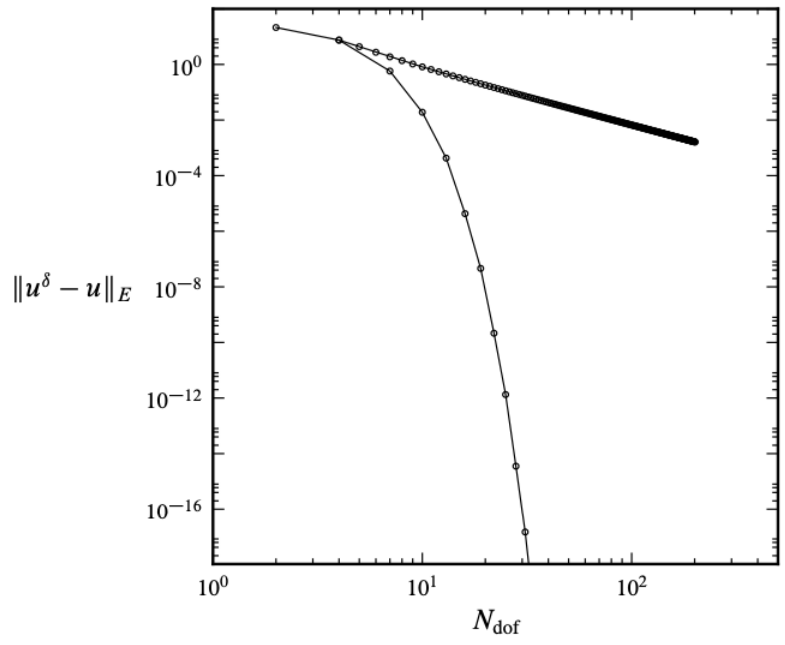


Figure 1.1 – Logarithmic plot of error against the number of degrees of freedom for h- (top line) and p-type (bottom line) convergence [1]

1.2. *What is Nektar++?*

Despite the somewhat obvious benefits of the spectral/hp element method, it is more complicated and difficult to understand, which has proven to be quite a wall preventing widespread use of the method. To make the spectral/hp element method more accessible for academics and industry professionals in various fields, Nektar++ was created.

Nektar++ is an open-source framework that provides a flexible, high-performance and scalable platform for the development of solvers for partial differential equations using the high-order spectral/hp element method [2]. Nektar++ is written in C++, making use of the object orientated programming with the aim of making it easy to use the complicated spectral/hp method, containing partial differential equation solvers within the package, which can be adapted to the need of the user. The need for such a software has already been outlined, to increase the accuracy of computational fluid problems whilst reducing the computational load by making use of the spectral/hp element method rather than using more traditional methods such as the finite element method or finite volume method. The common issue which is associated with the spectral/hp element method is that it is often perceived to be quite fragile, due to difficulties with elements self-intersecting, although Nektar++ uses several techniques to address this problem [2].

The framework currently has the following capabilities [3]:

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

The figure below shows the individual libraries within Nektar++ and what they are used for.

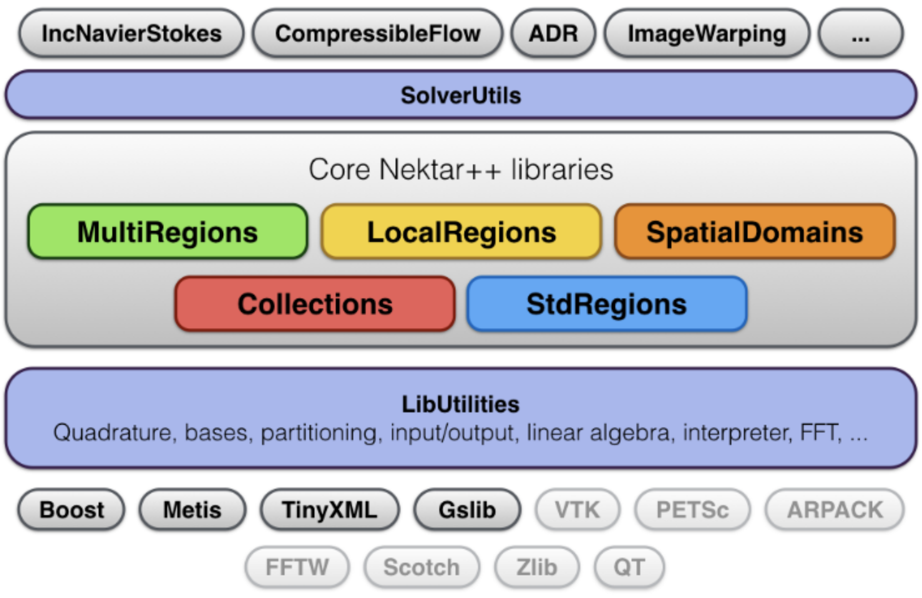


Figure 1.2 – Structure of Nektar++ Libraries.[46]

Within this project, the focus will be on NekMesh, which generates a high-order mesh by only using one executable rather than doing it by doing it in several other steps, which is common in other high-order mesh generators. The pipeline for generating a mesh is shown below in Figure 1.3, with the dotted line representing the goal for Nektar++ to produce the high-order mesh.

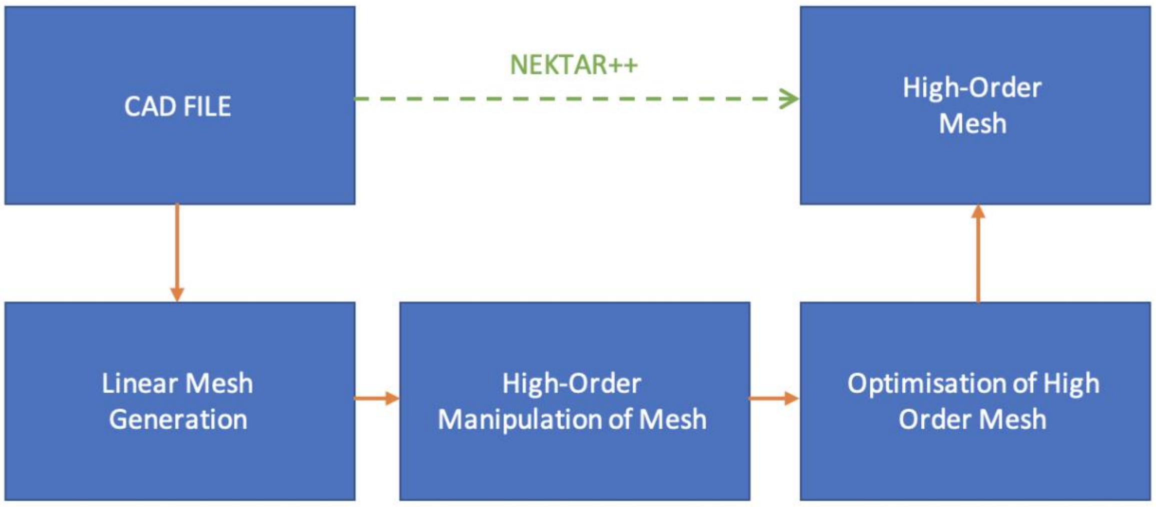

CAD FILE
NEKTAR++
High-Order Mesh
Linear Mesh Generation
High-Order Manipulation of Mesh
Optimisation of High Order Mesh

Figure 1.3 – Target pathway of generating a high-order mesh through Nektar++ (inspired by [4])

1.3. *Other High-Order Mesh Generators*

High-order mesh generation is something which is significantly more commonplace due to the undeniable benefits over more traditional methods of mesh generation. There are other companies which are generating high order meshes such as Pointwise. Pointwise was founded in 1994 'to meet the mesh generation needs of the CFD community' [5], written primarily in the Tcl programming language using the scripting language Glyph to extend the language, allowing the users to make use of commands and entities of the program. Pointwise has many big customers from around the world, including Lockheed Martin, Boeing and even NASA [6]! Despite the success of Pointwise, the most glaring issue with it is that it is written in a fairly unknown language which creates a fairly large barrier to entry and can make the software unattractive as users might not want to learn a niche language with limited use. This is where Nektar++ becomes a particularly attractive proposition. Being written in the much more popular C++, Nektar has a much larger audience, making high-order mesh generation much more accessible [4].

The design philosophies behind Nektar++ are to primarily be as unobtrusive as possible and as easy to use for the end user as possible, whilst maximising the functionality and configurability of the software. Nektar++ makes the software accessible and easy to understand by ensuring that all of the software is modular, where each module does one thing, where the software automatically passes a common mesh between modules within a pipeline [7]. Nektar also tries to maintain the CAD information from the time it enters to the time it exits the software, and to try to prioritise high order throughout the pipeline, from when the initial mesh is made until the final mesh optimisation has been completed. The primary benefit of Nektar++ over other high order mesh generation software to be the modularity of the code, which gives the user the freedom to use the software how they like. Whether the user wants to just use Nektar++ to generate a linear surface mesh, or to use the software to optimise a mesh, or solve CFD problems using the software, it is possible which is not common throughout all mesh generation software. This allows for the user to decide how they want to use the software in addition to the use of it with other software, reducing the intrusiveness of the software.

1.4. The Need for an Interface

Nektar++ has found a variety of different uses, solving computational fluid dynamic problems, across a wide range of fields, varying from Formula One [8] to the biomedical industry (cardiac electrophysiology) [3]! Despite the numerous uses for the software, the relatively complex syntax of C++ can be perceived to be somewhat off-putting for a new user of the code or somebody with little coding experience, with complex hierarchies of classes and inheritance within Nektar++ itself [2].

With the objective of the framework to be to make the spectral/hp element method more accessible to the widest possible audience, having a high barrier to entry seems nonsensical. Therefore, an interface only seems necessary to make Nektar++ more accessible.

1.5. Objectives

The primary objective of this project was to expand the framework of the Nektar++ libraries, taking the existing Boost.Python Python bindings, expanding on them to be able to use NekMesh to lower the barrier to entry. To achieve the primary objective of the project, the goals were extended to ensure that any development must be able to interact with other libraries within Nektar++ and can be used in conjunction with completely separate software to make use of any programs which were developed to be used within a pipeline. By making use of a pipeline, it makes Nektar++ much more futureproof and to allow for a much wider range of applications of the software.

Once any programs are developed, it will be important to document any changes, informing how they work and can be used (for the users to be able to use the software) and documenting how they work (for the developers to be able to further development, and to make new programs interact with the existing programs).

2. Literature review

In recent years, the ability to run simulations using Computational Fluid Dynamics or Finite Element Analysis have become extremely commonplace and are available to most engineers. Since the use of CFD and FEA are becoming more accessible, there is continual research into the development of the methods to complete tasks more efficiently, such as development on the spectral/hp-element methods.

The term ‘Finite Element Method’ was coined by R.W. Clough, in his 1960s paper titled ‘The Finite Element Method in Plane Stress Analysis’, developed by Dr O. C. Zienkiewicz in his book titled ‘The Finite Element Method: Its Basis and Fundamentals’ in 1966 which remains to be the standard text for the Finite Element Method. From there, there have been significant developments throughout the CFD world, with almost every paper relating back to Zienkiewicz’s original book.

When considering the Spectral/hp-element method, there is one main text, which is referred to answer almost all questions about the inner workings of the method. The book titled ‘Spectral/hp

Element Methods for Computational Fluid Dynamics' by George Karniadakis and Spencer Sherwin [9]. The book takes the reader from the basic ideas of the how the Spectral/hp element method works in one-dimension, progressing to more complex, real world examples. The first two chapters of the book were extensively used within the report to demonstrate the need for such methods, and to give numerical evidence of how the method works within one-dimension. The book outlines the differences between linear mesh generation and a high-order mesh and the benefits between the two. In its simplest form, a linear mesh is one which is formed of elements with straight lines, whereas within a high-order mesh, the elements are formed of curvilinear entities allowing for better mapping around complex geometry. High-order mesh generation is largely better than linear generation, because of the increased accuracy when solving the partial differential equations and the decrease in computational time despite the overall increase in computational time per element. Due to high-order methods generating a global solution, fewer elements need to be created to obtain a more accurate result which overall lowers the computational time. With computational methods becoming more accessible and commonplace it seems almost nonsensical that the spectral/hp element method is not equally as commonplace. However, because of the robustness of linear mesh generation, compared to high-order mesh generation, makes high-order mesh generation significantly more difficult due to self-intersecting elements (discussed further in Section 3.2). In this sense, robustness refers to the code working in a wide range of cases with little knowledge of how mesh generation works.

Despite the difficulties in creating a stable high-order mesh generator, there are a few pieces of software which are being developed in an effort to tackle this problem. One such example of software is Nektar++, which is an open-source high-order mesh generator which is being continually developed by a team based from the University of Utah, Imperial College London and the University of Exeter. Nektar++ is currently the only open-source high-order mesh generator which exists. Being open-source means that any user of the code can view the source code and adapt it to their individual needs. The software contains a lot of documentation which has been used extensively throughout the report, in order to ensure that any pieces of software which are developed are able to interact properly with the existing code, dictating many decisions which were made throughout the project.

Efforts from within the Nektar++ development team have already expanded the software to be able to make use of Python, such work has been completed by Emilia Juda, in her paper titled 'Development of a Python Interface to Nektar++' which created Python tutorials for the package and examined the memory usage between Python and C++ [10]. Juda's work develops the NekPy package to be able to efficiently pass arrays between the two languages which has proven useful as it means that it greatly reduces the memory usage when using the Python interface. As a part of the work, Juda gives scope for further work on the Python interface, explaining that the NekMesh and FieldConvert utilities are still lacking an entry point.

A paper [2] published by David Moxey, a Project Leader on the Nektar++ development team outlines the need for the expansion of the Python interface of Nektar++. As explained within the report, Nektar++ was designed with C++, making use of the object orientated programming with complex hierarchies of classes and inheritance. The report then continues to explain the issues with a program being written in C++, namely the significant barrier of entry to using the program, due to the complex syntax of the language. Python is a language which is known for having very simplistic syntax and generally easy to use, an idea explored within Section 3.3.2, making it a very attractive proposition for an interfacing language. As part of the Python interface within the Nektar++ package, it was identified that there were two gaps. One of which being a method of generating a mesh which is developed primarily through Python using NekMesh. Further on within the report, the method in which NekMesh works is explored, using a posteriori approach, creating a linear mesh which then has nodes added along edges faces and volumes to achieve a high-order polynomial discretisation of a mesh. The issues which high-order mesh generation are also explored, with some elements becoming invalidated after the addition of these high-order nodes due to some overlapping, self-intersecting elements. In addition to Moxey's publication, Michael Turners Thesis 'High-Order Mesh Generation For CFD Solvers' [11] was used extensively to understand the workings of NekMesh.

From Moxey's paper, the need for development of the Python interface is explicit, and when combined with Juda's work leaving scope for development of the NekMesh Python branch, the shape of the project was formed with adequate documentation behind it.

3. Background Theory

3.1. Numerical Methods in Computational Fluid Dynamics

There are numerous methods used to calculate Computational Fluid Dynamics, the most popular methods are outlined below. For the sake of simplicity of understanding the methods below, they are only considered in one dimension.

3.1.1. Finite Difference Methods

Dating back to the 17th century, Finite Difference Methods are one of the most simple and oldest methods which can be used to solve differential equations. Finite Difference Methods began to get used within the early 1950s with the invention of computers being able to deal with the complex problems [12]. The main concept behind differential methods is related to the derivative of the first differential of the function u(x):

$$\frac{\partial u}{\partial x}(x) = u_x(x) = \lim_{\Delta x \to 0} \frac{u(x + \Delta x) - u(x)}{\Delta x} \tag{1.1}$$

As Δx approaches 0 (without reaching 0), the right-hand side of the equation provides a good approximation to of the derivative of x.

The functions for the first derivative of a finite difference formula are the forward difference (Equation 1.2), central difference (Equation 1.3) and backwards difference (Equation 1.4). These formulae are derived by expanding the Taylor series.

$$u'(x) \approx \frac{u(x + \Delta x) - u(x)}{\Delta x} \tag{1.2}$$

$$u'(x) \approx \frac{u(x + \Delta x) - u(x - \Delta x)}{2\Delta x} \tag{1.3}$$

$$u'(x) \approx \frac{u(x) - u(x - \Delta x)}{\Delta x} \tag{1.4}$$

Both the forward and backwards difference equations have an error of O(Δx), whilst the central difference has an error of O(Δx)2. This error is due to the truncation error when expanding the Taylor series.

Finite Difference Methods are less efficient where complex geometry is involved [9] and prefer simple geometries. As example use of Finite Difference Methods is a simple heat conduction problem on a grid.

3.1.2. Finite Volume Method

In the Finite Volume Method, the region of flow of the fluid is divided into small cells, which when combined form a mesh. The equations of the flow in and out of each cell can be represented by the changing of the volume of the fluid. This is done by integrating the equations over the volume of each cell with the results being differential equations which can be solved.

When considering a linear differential equation in a domain Ω denoted by:

$$\mathbb{L}\big(u(x)\big) = q(x) \tag{1.5}$$

Where 𝕃 is a linear differential operator acting on the function u, which results in a function of q.

When computing $u^{\delta}(x)$, the result will most likely not be exact to the answer u(x), therefore a non-zero residual is added to the equation, this yields the following equation:

$$\mathbb{L}\big(u^{\delta}\big) - q(x) = R\big(u^{\delta}\big) \tag{1.6}$$

Where $u^{\delta}(x) = u(x)$ and R(u) = 0

Using the method of weighted residuals, the equation is multiplied by a weight set to be w(x) and integrated over the domain.

$$\int_{\Omega} w(x)\mathbb{L}\left(u^{\delta}(x)\right)dx - \int_{\Omega} w(x)q(x)dx \ = \ \int_{\Omega} w(x)R\left(u^{\delta}(x)\right)dx \tag{1.7}$$

When the residual R is multiplied by the weight function, it will be equal to zero (it is defined to be like this), this results in the weak differential equation shown below.

$$\int_{\Omega} w(x)\mathbb{L}\left(u^{\delta}(x)\right)dx = \int_{\Omega} w(x)q(x)dx \tag{1.8}$$

In the Finite Volume method, the weight function used is:

$$w(x) = \begin{cases} 1, & inside\ \Omega^{j} \\ 0, & outside\ \Omega^{j} \end{cases} \tag{1.9}$$

Where Ω_j represents the domain.

The finite volume method is used often within the aerodynamics industry as it is a conservative method, meaning that the fluid entering a cell is equal to the fluid leaving the adjacent cell [9].

3.1.3. Finite Element Method

Finite Element Methods developments began in the 1950's, with papers being written by Turner, et all, and progressed into 'The Finite Element Method' book which was written in 1966 by Dr O.C. Zienkiewicz which remains to be the standard reference text for the basis of the finite element method [13]. Finite Element Methods are primarily used in structural mechanics, but these methods are equally as useful in fluid dynamics despite the original objectives of the developers not targeting the fluid dynamic field [9].

The basic principle of the finite element method is to divide the domain which is being analysed into small parts called elements (creating a mesh) and then solving the partial differential equations. It is assumed that the solution of u(x,t) can be accurately represented by the approximate solution below [9]:

$$u(x,t) = u^{\delta}(x,t) = \sum_{i=1}^{N_{dof}} \hat{u}_i(t)\Phi_i(x) \tag{1.10}$$

Where $\Phi_i(x)$ are trail (or expansion) functions and $\hat{\mathrm{u}}_{\mathrm{i}}(\mathrm{t})$ is the unknown values.

The majority of finite element method solvers use low-order polynomials in order to solve the partial differential equations, for the most part they are either linear or quadratic! However, it is possible to increase the order much higher, which produces more accurate results, these higher order methods are often referred to as p-FEM. In p-FEM the order of the elements is much higher, meaning that the accuracy of the results from the analysis will be much greater.

The other method for reducing the error of a finite element analysis is to have variable sizes of elements, often increasing the number of elements around awkward bits of geometry. This method is commonly referred to as h-FEM. Obviously by combining both methods of increasing accuracy will increase the accuracy of the analysis further. This is referred to as hp-FEM.

Many FEM solvers use the Galerkin Method, due to the power of the method for deriving partial differential equations to be solved. The Galerkin Method works by using the weight function which has been chosen to equal to the expansion function, such that [1]:

$$w(x) = \sum_{j=1}^{N_{dof}} w_j\, \Phi_j(x) \tag{1.11}$$

Substituting the weight function into Equation 1.8 yields:

$$\int_{\Omega} \sum_j w_j \Phi_j \sum_i \hat{u}(t)\Phi_i \, dx = \int_{\Omega} \sum_j w_j \Phi_j q(x)\, dx \tag{1.12}$$

Which can then be simplified to:

$$\sum_j \sum_i w_j \int_{\Omega} \Phi_i \Phi_j dx\, \hat{u}(t) = \sum_j w_j \int_{\Omega} \Phi_j q(x) dx \tag{1.13}$$

Expanding Equation 1.13 into its matrix form gives:

$$[w_1 \quad w_2 \quad \cdots \quad w_j]\begin{bmatrix} \int_\Omega \Phi_1\Phi_1 dx & \int_\Omega \Phi_1\Phi_2\, dx & \cdots & \int_\Omega \Phi_1\Phi_j\, dx \\ \int_\Omega \Phi_2\Phi_1\, dx & \int_\Omega \Phi_2\Phi_2\, dx & \cdots & \int_\Omega \Phi_2\Phi_j\, dx \\ \vdots & \vdots & \ddots & \vdots \\ \int_\Omega \Phi_i\Phi_1\, dx & \int_\Omega \Phi_i\Phi_2\, dx & \dots & \int_\Omega \Phi_i\Phi_j\, dx \end{bmatrix}\begin{bmatrix} \hat{u}_1 \\ \hat{u}_2 \\ \vdots \\ \hat{u}_i \end{bmatrix} \tag{1.14}$$

$$= [w_1 \quad w_2 \quad \cdots \quad w_j]\begin{bmatrix} \int_\Omega \Phi_1 q(x)\, dx \\ \int_\Omega \Phi_2 q(x)\, dx \\ \vdots \\ \int_\Omega \Phi_j q(x)\, dx \end{bmatrix}$$

Equation 1.14 can then be solved using linear algebraic methods as it is of the form Au = b.

3.1.4. Spectral Method

Spectral Methods started being developed by Gottlieb and Orszag in the 1970’s, being published in their 1977 paper, Numerical Analysis of Spectral Methods: Theory and Applications [14]. Spectral methods use basic functions over the whole domain, where finite element methods use nonzero basic functions in local regions. This means that the derivative of a certain point in space depends on all other points within the domain, whereas in finite element methods it only depends on neighbouring points, leading the error to reduce exponentially as opposed to algebraically in the way that it would in finite element methods [15]. Despite the increased accuracy from spectral methods, they are not without their drawbacks, they are more difficult to implement when compared to finite element methods.

3.1.5. Spectral Element Method

The first paper on the spectral element method was released in 1984 by A. T. Patera, combining the generality of the finite element method with the accuracy from spectral techniques to solve Navier-Stokes equations [16]. The general idea behind the spectral element method is using the geometric flexibility of the finite element method with the high accuracy from the spectral method. The advantages of the spectral element method when compared to the finite element method is that with fewer elements and fewer degrees of freedom per node, the accuracy of the results is much higher, reducing computational load and therefore time to compute.

3.1.6. Spectral/hp Element Method

The spectral/hp element method is the most advanced of the spectral methods, using the geometric flexibility of the h-finite element method (increasing the number of elements around a complex section of geometry) and the numerical properties of the spectral element methods (high order polynomial basis functions on a coarse finite element type mesh, with fast convergence and high accuracy) [17].

Similarly, to in the finite element method, the domain is divided into many non-overlapping elements, represented by Ω, in which polynomial expansion is used. These elements are then converted into standard elements, represented by Ω_{st}. These standard elements can then be transformed to local elements which reduces the overall computational load as global modes are, at a maximum, nonzero on two elemental regions. Whereas local elements have a unit value at one end, which decays linearly across the neighbouring elements to zero [9]. The standard element for one-dimension is:

$$\Omega_{st} = \{\xi | -1 \leq \xi \leq 1\} \tag{1.15}$$

Where ξ represents the local coordinate.

The standard element can then be then transformed ($\chi^e(\xi)$) in terms of the local coordinate, producing a curvilinear element:

$$x = \chi^e(\xi) = \frac{1-\xi}{2} x_{e-1} + \frac{1+\xi}{2} x_e, \xi \in \Omega_{st} \tag{1.16}$$

Once the local solutions have been found, the global solution is then found, using either the continuous or discontinuous method. Being continuous means that there is continuous solution between the interfacing elements, whereas the discontinuous method does not require this continuous solution, rather requires a continuous flux between the interfacing elements [10].

To see a more complete explanation on the Spectral/hp element method, please refer to 'Spectral/hp Element Methods for Computational Fluid Dynamics' by George Karniadakis and Spencer Sherwin [9].

3.1.7. Gaussian Quadrature

Quadrature is the process of determining the area of a plane geometric figure by dividing it into a collection of shapes of known area and finding the limit of the sum of these areas [18]. Gaussian Quadrature is a method of gaining an accurate numerical estimate of an integral by picking an optimal abscissa (the x-coordinate of a point in a two-dimensional coordinate system) at which to evaluate a function [19].

To have the greatest level of accuracy in the approximation $\int_a^b f(x)dx$, the nodes $x_1, x_2, \ldots, x_n$ in [a,b] should not be equally spaced. To perform the quadrature function, the limits must always be between [-1,1]. To do this a weight added to ensure that the limits will be [1,-1]. The formula for the gaussian quadrature method is:

$$\int_{-1}^{1} f(x)dt = \sum_{i=1}^{n} w_i f(x_i) \tag{1.17}$$

Where w_i is the weight function, n is the number of points and x is the points.

The quadrature formula has a much greater accuracy when compared to the trapezium rule, and for a polynomial of an order n, the accuracy is 2n-1. Therefore, if a polynomial has an order which is less than, or equal to 2, the approximation will be exact. This makes Gaussian quadrature useful as you can approximate complex integrals with relatively little computing power.

3.2. NekMesh

NekMesh is the utility within the Nektar++ which is used to generate high order meshes, using only one executable command. NekMesh can be found within the MultiRegions library of Nektar++. The process in which NekMesh works by initially generating a linear mesh on the geometry, then adding nodes along edges, faces and volumes to then create a curvilinear element from the standard element. This approach is known as the posteriori approach. There is however a large problem with high order meshing which is that it can often produce elements which self-intersect. In the event that an element self-intersects, the solution from within that element cannot be calculated, and therefore the mesh is no longer valid. Examples of a valid and invalid element can be seen in Figure 3.1.

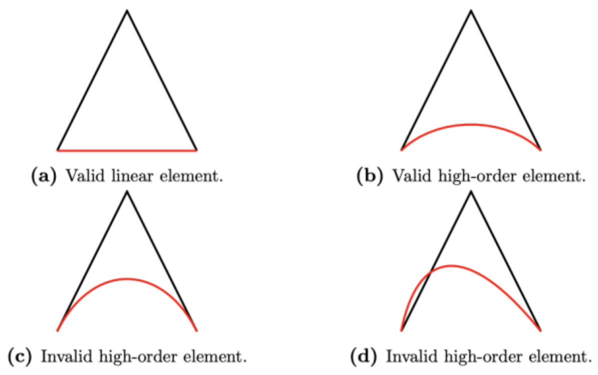

(a) Valid linear element.
(b) Valid high-order element.
(c) Invalid high-order element.
(d) Invalid high-order element.

Figure 3.1 – Examples of valid and invalid elements. [11]

To generate a mesh, NekMesh makes uses of the Octree method (a method for dividing cube into eight equal parts by recursively subdividing, for more information on the Octree method please refer to [20]) and a curvature-based refinement system. Once the mesh has been generated, to correct any elements which would be self-intersecting, NekMesh and the further Nektar++ library is capable of using a linear elastic solver within the meshing pipeline. For more information regarding the inner workings of NekMesh, please refer to Michael Turners 2011 paper, High-Order Mesh Generation for CFD Solvers [11].

3.3. Interfacing between C++ and Python

3.3.1. C++

C++ (originally known as 'C with Classes') was first created in 1979 by Bjarbe Stroustrup. Stroustrup wanted a language which would enable him to write 'efficient systems programs in the styles encouraged by Simula67' [21]. Simula67 is considered to be the world's first object orientated programming language, which was widely used throughout the 60's and 70's, initially for 'simulating discrete event systems' [22]. In order to achieve his goal, Stroustrup added 'facilities for better type checking, data abstraction and object-oriented programming to C' with a general aim to be both 'efficient and elegant' which were things you had to choose between before C++ [21]. C was used as a base language for a few key reasons, C is a powerful and efficient programming language which has relatively simple syntax which is fast to compile and was very portable meaning that it could be used on any operating system [23].

C++ saw a huge surge in popularity soon after being created, due to the aforementioned simplicity and efficiency of the language. C++ is the 4th most popular programming language with approximately 6.3 million users (as of April 2019) [24]. Due to the number of users, it is unsurprising how many uses and applications there are for C++, being used in applications from computer operating systems, CAD software all the way to being used in aircraft software and investment banking software [25]!

Some of the most identifying, categorising characteristics of C++ are:

- An Object-Orientated Programming Language (OOP) – a language which organises the design of the software around the objects rather than around functions and logic.
- A mid-level programming language – the source code of the language is easy for a human to read the syntax which is then converted to a low-level language which a computer can understand and read. C++ was formerly known as a high-level programming language but with the addition of newer languages such as Python it is more commonly grouped with low-level languages.
- A general-purpose programming language – a language which is capable of being used for any type of program, (HTML is an example of a domain-specific language, which can only be used in a specific domain).
- Compiler-based – Before source code can be executed within a command line, the code is run through a compiler to 'translate' the code into machine code. A compiled language usually runs faster than an interpreted language.
- Strongly typed – Each variable must be defined as a data type.

C++ has come through multiple different iterations which are standardised by the ISO, for the first time in 1998 (C++98), which has had 5 revisions and are currently working on the next revision, C++20. The different iterations of the language mean that bug fixes within the software and can also be used to push new features.

Some of the most notable disadvantages of C++ include being poor for web development, often being difficult to debug and becoming extremely complicated when being used for very large programs.

3.3.2. Python

Inspired by the ABC programming language emerging from the Netherlands during the 1980s, Dutch computer programmer Guido van Rossum at Stichting Mathematisch Centrum, decided to create Python as a hobby project to overcome the issues created by the ABC programming language [26]. When creating the language, he took the syntax and some of the features he liked from ABC and fixed the issues and flaws. The original design goal when creating Python was to 'serve as a second language for people who were C or C++ programmers, but who had work where writing a C program was just not effective' [27].

Python has increased in users dramatically within the last decade, with a growth rate of 27% year-over year-growth rate [28]. There are numerous reasons for the growth rate of Python, including the incredibly simple syntax which reads and writes much like plain English, with a very active community which are consistently offering support and tutorials to beginners within the language and large corporate sponsors, such as Google, backing the language [29]. As of April 2020, Python is the 3rd most popular programming language in the world, with 8.2 million users worldwide [24].

Some of the defining characteristic features of Python include:

- An Object-Orientated supporting Programming Language – a language which organises the design of the software around the objects rather than around functions and logic.
- A high-level programming language – when programming in Python, system architecture and memory management do not need to be considered, and the syntax of the language is similar to writing in plain English.
- A general-purpose programming language – a language which is capable of being used for any type of program, (HTML is an example of a domain-specific language, which can only be used in a specific domain).
- Interpreted Language – Python is executed line by line at a time and does not require a compiler.
- Weakly typed – The datatype does not need to be specified for each variable.

Since the initial release of Python 1.0 in January 1994, Python has developed through multiple iterations to the most recent release of Python 3.7 released 27th June 2018, going from a language which was designed to serve as a second language to becoming a language with a large array of applications. Despite the continual development of Python, there are currently two versions being widely used, Python 2.7 (released in July 2010 [30])and Python 3.0 (released December 2008 [31]). Python 3.0 was released to fix the issues within 2.7 however, due to the nature of these fixes, Python 3.0 is not backwards compatible making the migration difficult. The result of this is that many organisations who were using Python 2.7 have to expend a lot of time to migrate to Python 3.0. However, since 1st January 2020 all development on Python 2.7 has discontinued [32].

There are almost an endless number of applications to use Python, from uses in science and numerical applications, web development, game design and machine learning [33]. Python is also used as an introductory language to teach people to code due to the simplistic syntax. The number of applications is also largely attributed to the number of libraries which have been developed for Python, within the Python Package Index (PyPI) there have been over 225,000 packages produced [34],not including packages which have been produced and published in other places, such as GitHub.

Python is often referred to as a glue language, a glue language is ‘a programming language that is designed specifically to write and manage program and code, which connects together different software components…using different programming languages and platforms,’ [35]. It is a particularly good glue language as it is able to interface with many different languages, including C/C++ using Boost.Python, SWIG, ctypes or Python C-Extension, Java using Jython, and R using RPy among many others.

Despite all of the benefits of using Python, like any other language it does have its drawbacks. Most notably, Python is renowned for being slow due to it being an interpreted language (the speed slow when comparing it to a compiled language such as C/C++ although for most applications it is fast enough). Another drawback is that Python has a very high memory consumption, due to the flexibility of data types [36] from being a weakly typed language.

3.3.3. *Python C API*

CPython is the reference implementation of Python written in C which includes the C API (Application Programming Interface). Since C++ is largely based on C, the simplest way to interface between C++ and Python is using the CPython API. All objects (a variable, data structure, function or method) within CPython are extensions of a PyObject. As previously discussed, all object types are treated the same way within Python, therefore the same objects are treated the same way at a C level, which is being represented as a PyObject. Each PyObject contains two pieces of data, a type which can determine the object type, and a reference count which counts how many times the object has been accessed.

Wrapping C++ into Python using the Python C API is possible, but it is very longwinded, and there are many steps for even the simplest of scripts to wrap them. Therefore, a wrapper such as SWIG (Simple Wrapper Interface Generator), Pyrex or Boost.Python have been developed to facilitate the interfacing between the two languages.

4. Design of Expansion of Nektar++ Framework

The overall objective of the project was to expand the Python framework (as outline in Section 1.5) whilst making the software as accessible to as many people as possibly by allowing NekMesh to interface with other software packages. Within this section of the report, the design process towards the program to generate a NekMesh will be explained in addition to how the code works.

4.1.1. *Why Python was Chosen*

The rapidly growing popularity of Python in computer science in addition to the steep rise in popularity of Python within the scientific community, due to the simplistic syntax for both the user and developer, with a large amount of documentation to assist the developer it seems logical to wrap Nektar++ into Python.

The continuous development of packages and modules from the wider community allows for Python to be the ideal glue language for completing complex tasks quickly and efficiently. By using the Python C API (discussed further in Section 3.3.3) it is possible to combine the simplistic syntax of Python with the speed, efficiency and performance of the less beginner friendly language, C++, which would not be achieved with Python.

Therefore, the aim of increasing the accessibility of Nektar++ to a wider community could be achieved by interfacing Nektar++ to be used by a Python user, without jeopardising the efficiency and speed of the code written in C++.

4.2. *Boost.Python*

Boost.Python is a framework for interfacing Python and C++ using no special tools, which is unobtrusive on the existing C++ code, not requiring any classes to be renamed or altered (Guzman & Abrahams, 2005).

Boost.Pyton was selected to be the wrapper for Nektar++ for a variety of reasons, including large number of libraries included within Boost which are already being heavily used within Nektar++ already, such as Boost.Numpy which is used for many computational tasks within Python.

Although an automated wrapper might seem more beneficial for wrapping the existing code, such as SWIG, the bindings are often not as robust or of as high of a quality. Avoiding an automated wrapper also allows much greater control over exactly what is being wrapped and how the wrapping is occurring.

The Boost libraries are more diverse than just their wrapping capabilities, they are also used to provide supportive tasks within C++, including linear algebra (uBLAS) and multi-threading (Thread) with over 150 other libraries! The Boost libraries are peer created and peer reviewed and are considered to be, '… one of the most highly regarded and expertly designed C++ library projects in the world,' [37].

Boost.Python works by importing the C++ objects as PyObjects. When the C++ is compiled, it creates a dynamic library (meaning that the library is read/write) in Python.

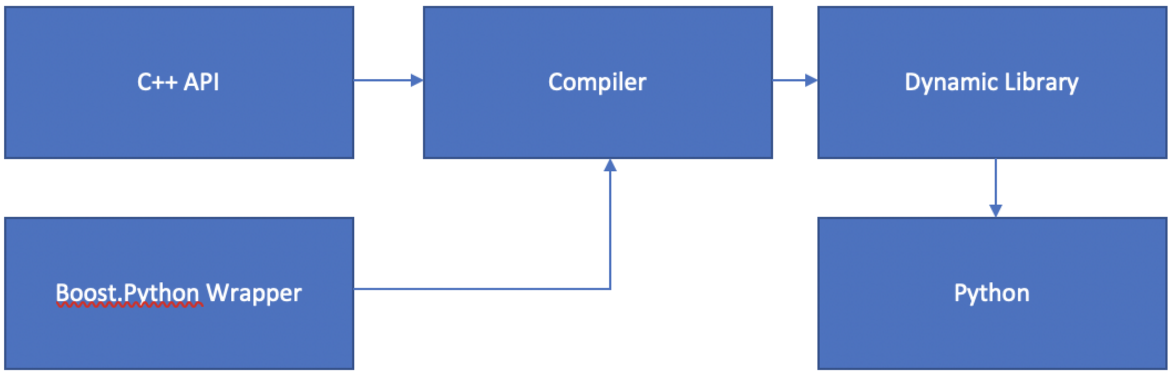


Figure 4.1 – Schematic of how Boost.Python is used to convert C++ Code to a dynamic Python Library (inspired by [38])

Boost.Python can convert many different types of data between C++ and Python, with the option for it to be automated, or for the developer to specify how the conversion should take place if the datatypes are non-standard. Once the converter has been registered by Boost.Python, the conversion will take place. A successful wrapper file will contain 3 things;

1. A method to convert from the C++ datatype to the Python datatype,
2. A method to convert from the Python datatype to the C++ datatype,
3. A method for checking whether the transformation has occurred.

Boost.Python provides the methods to wrap classes easily. An example of a wrapped class is shown below in Listing 4.1.

Listing 4.1 - Example of Boost.Python wrapping syntax of a class

```cpp
void export_Greeting()
{
  py::class_<Greeting>("Greeting")
      .def("greet", &Greeting::greet)
      .def("set", &Greeting::set)
      ;
}
```

The name of the class is defined in angled brackets after the class_ keyword. Within the following brackets contains the name of the class within Python, in the listing above, there are no arguments after the class name, however, in some cases a class would have arguments, which would be the class initialisation arguments. The following lines within Listing 4.1 contain the methods which are found from within the class, with the class name to be in Python within quotation marks, followed by the pointer to the class within the C++ class method.

The arguments of the methods are automatically decided by Boost.Python which can often be convenient although if C++ method has default arguments, the wrapping becomes more complicated and extra steps must be taken. The reason for this is that the C++ function pointers do not contain the default argument. This is where a thin (or lightweight) wrapper is used. Lightweight wrappers will get called rather than the default method from the C++ class when the they are called from Python. When using a lightweight wrapper, Boost.Python will automatically decide which method to call depending on the arguments which are provided when the method is called.

4.3. Wrapping C++ to Python

As previously discussed, before any coding could be done in Python to generate a mesh, Boost.Python had to be used to wrap some of the C++ code. Within the NekMeshUtilities directory, all types of MeshElements can be found. Within this directory, are all the files necessary to make different shaped elements within a mesh, with their .cpp file and their .h file. Within C++, when a class is created, it is created across two files, the .h file (the header file) contains the definitions and functions of the class, where the .cpp file contains the implementation of the class. At first glance, it might seem as though all of the files within the MeshElement directory would need to be wrapped in order to create a mesh using Python, although if a file calls other .cpp files from within it, the other .cpp files are not required to be wrapped in order to call them.

For example, within the elemet.cpp file, other files such as shapetype are called, allowing for just the elemenet.cpp file to be wrapped. The files which were necessary to wrap to be able to form meshes from a Python interface include: Element.cpp, LibUtilities.cpp, Mesh.cpp, Module.cpp, NekMeshUtils.cpp and Node.cpp. An example of a wrapped file is shown below in Listing 4.2 it is the Mesh.cpp file.

Listing 4.2 - Wrapped Mesh.cpp file

```
#include <LibUtilities/Python/NekPyConfig.hpp>
#include <NekMeshUtils/MeshElements/Mesh.h>

using namespace Nektar;
using namespace Nektar::NekMeshUtils;

void export_Mesh()
{
    py::class_<Mesh,
               std::shared_ptr<Mesh>,
               boost::noncopyable>(
                   "Mesh", py::init<>())
        .def_readwrite("node", &Mesh::m_vertexSet)
        .def_readwrite("verbose", &Mesh::m_verbose)
        .def_readwrite("expDim", &Mesh::m_expDim)
        .def_readwrite("spaceDim", &Mesh::m_spaceDim)
        .def_readwrite("nummode", &Mesh::m_nummode)
        .def_readonly("element", &Mesh::m_element)
        ;
}
```

4.4. Expanding Python Framework

4.4.1. Coding Philosophy

When developing the Python framework, it was important to retain the coding philosophies which were laid out by the Nektar++ team (outlined in Section 1.3). This primarily meant that the code was kept modular, meaning that it can be quickly implemented or modified to allow for another user to change it. To make the code a modular as possible, the object orientated programming capabilities of Python were exploited.

The secondary points within the design philosophy include trying to minimise the number of parameters that the user has to input whilst preserving any CAD information as much as possible.

The core objectives of the project were also considered, to ensure that the software is accessible to as big of an audience as possible. To try to ensure this, the software was made with comments which explained each section of code for easier understanding of the processes occurring at each point, in addition to breaking up the structure of the code to look less intimidating.

4.4.2. Core Ideas of Module_Create.py

The process of creating Module_Create.py largely consisted of interfacing between the C++ and Python. This meant that the differences between the two languages had to be considered at all points, and how they would interact. For example, when any variable which was made within the Python side of the code which would interact with the C++ side, it had to be made into an object which C++ could read. As previously discussed, this is because within the C Python API, the object type does not need to be defined, whereas in C++ it must be. To be able to 'translate' the objects, set functions from within Nektar++ must be used, to let the C++ side of the code knows what type of data is being input. This is done by using the functions; GetFloatConfig/GetStringConfig/ GetBoolConfig/GetIntConfig depending on the type of object which is being passed into the C++ code. The C++ code will be expecting a certain type of data, so it is important that they match otherwise errors will be raised.

Within Nektar++, the Factory Method Pattern is used to reduce the amounts of confusing code consisting of many if/elif/else statements and allows the classes to be chosen as the code is being run in addition to making the code look much cleaner. The Factory Method Pattern is used in Module_Create.py to create the input and output modules.

The process modules are pulled directly from the wrapper file, which process the vertices/edges/faces/elements/composites, which generate the mesh to be output.

4.5. Module_Create.py

Module_Create.py was created with the idea that a simple two-dimensional mesh could be made efficiently with very few inputs to allow for a simple mesh to be progressed through a pipeline. Module_Create only requires two coordinates, to create a quadrilateral shape for a mesh to be applied. One the shape has been made, the ability to create the mesh is customisable to create the mesh which the user wants. The user is able to create a mesh from tetrahedrons or triangles, decide whether they would like a boundary layer to be created (using Chebychev nodes) and decide the density of the mesh by deciding the number of nodes in both the x and the y direction.

As the code was going to be written almost entirely by using object orientated programming, a class was created called TestInput which inherits from the module class, meaning that TestInput is a module. The initialisation of the function is then using the initialisation function from mesh. As Module_Create.py is only being used to generate two-dimensional surface to generate a mesh on, the space and expansion dimension are set to be 2.

The following block of code is all adding configuration options from the register configure function which are registered outside of the class. To add a configuration, three arguments are needed (with one optional argument), which are the name of the option, the default value of the option if no value is give, a brief description of the option and an optional argument which must be switched to True if the argument is a Boolean function. This is shown in Listing 4.3 below.

Listing 4.3 - Shows the syntax of adding a configuration option to the input module

```
self.AddConfigOption("nx", "2", "Number of points in x direction")
self.AddConfigOption("ny", "2", "Number of points in y direction")
self.AddConfigOption("coord_1x", "0", "Coordinate 1x")
self.AddConfigOption("coord_1y", "0", "Coordinate 1y")
self.AddConfigOption("coord_2x", "0", "Coordinate 2x")
self.AddConfigOption("coord_2y", "0", "Coordinate 2y")
self.AddConfigOption("comp_ID", "0", "Composite ID")
self.AddConfigOption("shape_type", "Quadrilateral", "Triangular/Quadrilateral Mesh"
)
self.AddConfigOption("chebychev", "False", "Boundary Layer X direction", True)
```

Within the Process method, the different variables are registered to be used within the class, and then the route which the code progresses is decided, and what needs to be executed. It is also where the quadrilateral is made, and the initial nodes are made.

The element shapes are created in either the _create_quadrilateral or _create_triangle methods. From within the Element.cpp file, the arguments are defined for the configuration of the element creation. Therefore, the element configuration requires four arguments. The first argument is the shape type, which in this program is limited to either be triangular or quadrilateral as previously discussed, the second argument is the order of the element, the third is whether face nodes are being added and the final argument is whether volume nodes are needed. Within these methods, the way in which the element is formed is then established, using the Element.Create function, which uses the element configuration, which was just discussed, the location of the nodes and finally the composite ID. Listing 4.4 below shows the _create_quadrilateral and _create_triangle code.

Listing 4.4 - Code extract showing how quadrilateral and triangular elements are created

```
def _create_quadrilateral(self, nodes, nx, ny, comp_ID):
     for y in range(ny-1):
         for x in range(nx-1):
             config = ElmtConfig(ShapeType.Quadrilateral, 1, False, False)
             self.mesh.element[2].append(
                 Element.Create(
                     config, # Element configuration
                     [nodes[y][x], nodes[y][x+1], nodes[y+1][x+1], nodes[y+1][x]],
# node list
                     [comp_ID])) # tag for composite.

```

```python
def _create_triangle(self, nodes, nx, ny, comp_ID):
    for y in range(ny-1):
        for x in range(nx-1):
            config = ElmtConfig(ShapeType.Triangle, 1, False, False)
            self.mesh.element[2].append(
                Element.Create(
                config,
                [nodes[y][x], nodes[y+1][x+1], nodes[y+1][x]],
                [comp_ID]))
            self.mesh.element[2].append(
                Element.Create(
                config,
                [nodes[y][x], nodes[y][x+1], nodes[y+1][x+1]],
                [comp_ID]))
```

For each element, the nodes are numbered in an anticlockwise due to convention. As a part of the node numbering, the vertices of the elements are numbered first, followed by any edge nodes, final to the interior nodes. This is the same convention is followed for any element shape. Figure 4.2 below gives a visual example of how the nodes of an element are numbered.

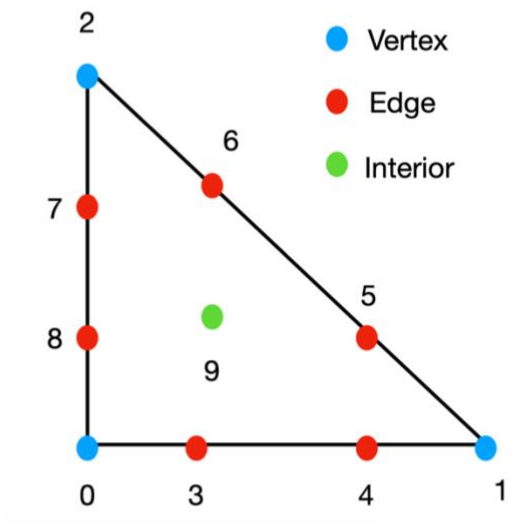


Figure 4.2– Node Numbering of an Element [46]

If the user decides that they would like to have a boundary layer (more dense meshing around the boundary), then they can do that by using Chebychev points. The creation of the Chebychev points occurs before the creation of the elements, as it changes the locations of the nodes. When creating the Chebychev nodes, it was important to ensure that the nodes were mapped between [-1,1]. This is why the return within the method exists, to map the points to the region of [-1,1]. The extract of code to demonstrate how the Chebychev points were generated is shown below in Listing 4.5.

Listing 4.5 - Code extract showing how Chebychev points are generated.

```python
def _create_chebychev(self, a, b, npts):
    k = np.arange(1, npts-1)
    x_points_temp = np.array([-1.0] + np.cos((2*k -1) / (2 * (npts-
2)) * np.pi).tolist()[::-1] + [1.0])
    # transform [-1,1] -> [a,b]
    #     #     [-1, 1] add 1 -> [0, 2]
    #     #     [0, 2] /2 -> [0, 1]
    #     #     [0,1] mult (b-a) [0, b-a]
    #     #     [0, b-a] add a -> [a,b]
    return (b-a) * 0.5 * (x_points_temp + 1.0) + a
```

Finally, within the class, the _call_module method calls the functions which generate the mesh from the generated element configurations.

Outside of the class is the registration of the configuration of the modules which are created using the Factory Method Pattern. An extract showing how the Module.Create was used is shown below in Listing 4.6. The Module.Create function takes 3 arguments [Module.Create(ModuleType.arg1, "arg2", arg3)];

1. The type of the module, whether it is an input/output/process module,
2. The name of the module being used,
3. The object which the process is occurring upon.

Listing 4.6 - Demonstrating the syntax of Module.Create

```
mod1 = Module.Create(ModuleType.Input, "test", mesh)
mod2 = Module.Create(ModuleType.Output, "xml", mesh)
```

Finally, there is a for loop, which sets any unset values to their default values and then executes any of the modules.

Module_Create.py will be available in the next release of Nektar++, following review from the Nektar++ team.

4.6. *LoadCAD.py*

LoadCAD.py was designed to import a step file, generated by CAD software, to generate a mesh to be used within a pipeline, using only Python. When designing loadCAD.py, the same things were important as were important in designing Module_Create.py. Most notably ensuring that the design of the software was modular, to allow for anybody to easily change the code to suit their individual needs. The overall objective of the project was also considered, allowing for the Nektar++ package to be able to interact with other pieces of software, whilst lowering the overall barrier of entry to the package. To make the script easier to understand, it was made to be modular, with all of the processes designed within a class, with each method within the class doing one thing. Comments were also used to explain what was happening at each point within the code in addition to being used to break up a long block of code. Significant effort was also made to ensure that loadCAD.py would adhere to the PEP 8 guidelines, so that the style would be easily readable for anybody who was looking at the script for the first time. More information on the PEP 8 Style Guidelines can be found on the Python.org website [39]. To ensure that PEP 8 guidelines are enforced, the class name (CadToMesh) is written using the CapWords convention, whilst and methods within the class are written in lowercase, with words separated using underscores.

The final script which can be used to generate a mesh from a step file is shown below in Listing 4.7.

Listing 4.7 - LoadCAD.py

```
import sys
from NekPy.NekMeshUtils import Mesh, Module, ModuleType

# Command line format:
# loadCAD.py INPUT.stp OUTPUT.xml order Volumemesh(True/False) mindelta maxdelta ep
s blsurfs blthick bllayers blprog
```

```python
# Must be a .stp file

class CadToMesh(object):
    # initialisation function
    def __init__(self, infile, outfile, nummode, mesh_type, mindel, maxdel, eps): #,
                # blsurfs=None, blthick=None, bllayers=None, blprog=None):
        self.infile = infile
        self.outfile = outfile
        self.mesh_type = bool(mesh_type)
        self.mindel = mindel
        self.maxdel = maxdel
        self.eps = eps
        # self.blsurfs = str(blsurfs)
        # self.blthick = str(blthick)
        # self.bllayers = str(bllayers)
        # self.blprog = str(blprog)

        self.mesh = Mesh()
        self.mesh.verbose = True
        self.mesh.expDim = 3 #set to a default of 3D
        self.mesh.spaceDim = 3 #set to a default of 3D
        self.mesh.nummode = nummode


    def cad_to_mesh(self):
        self.loadcad()
        self.create_octree(self.mindel, self.maxdel, self.eps)
        self.decision()
        self.outcad(self.outfile)

    def decision(self):
        # Method decides whether to create just a surface mesh, or a high
        # order surface mesh. If volume mesh is selected, it will be called
        # within this method.
        if self.mesh.nummode == 1:
            self.surface_mesh()
        else:
            self.surface_mesh()
            self.ho_surface_mesh()

        if self.mesh_type:
            self.volume_mesh()

    def def_and_process(self, mod):
        # Method used to process each module.
        mod.SetDefaults()
        mod.Process()

    def loadcad(self):
        # Method which imports the CAD file to be processed into a mesh.
        # Configs are chosen from the C++ side.
        loadcad = Module.Create(ModuleType.Process, "loadcad", self.mesh)
        loadcad.RegisterConfig("filename", self.infile)
        loadcad.RegisterConfig("verbose", " ")
        self.def_and_process(loadcad)

    def create_octree(self, mindel, maxdel, eps):
        # Creates an octree, which is a way of dividing a cube into equal
        # parts.
        octree = Module.Create(ModuleType.Process, "loadoctree", self.mesh)
        octree.RegisterConfig("mindel", mindel)
        octree.RegisterConfig("maxdel", maxdel)
        octree.RegisterConfig("eps", eps)
        self.def_and_process(octree)

    def surface_mesh(self):
        # Method generates a linear mesh
        surfmesh = Module.Create(ModuleType.Process, "surfacemesh", self.mesh)
        self.def_and_process(surfmesh)
        self.mesh.expDim = 2

    def ho_surface_mesh(self):
        # Method generating a high-order mesh, only called if the order
        # is greater than 1.
        homesh = Module.Create(ModuleType.Process, "hosurface", self.mesh)
        self.def_and_process(homesh)

    def volume_mesh(self):
        volmesh = Module.Create(ModuleType.Process, "volumemesh", self.mesh)
        # volmesh.RegisterConfig("blsurfs", self.blsurfs)
        # volmesh.RegisterConfig("blthick", self.blthick)
        # volmesh.RegisterConfig("bllayers", self.bllayers)
        # volmesh.RegisterConfig("blprog", self.blprog)
        self.mesh.expDim = 3
        self.mesh.spaceDim = 3
        self.def_and_process(volmesh)

    def outcad(self, outfile):
        # Method which saves the file as an .xml file.
        output = Module.Create(ModuleType.Output, "xml", self.mesh)
        output.RegisterConfig("outfile", self.outfile)
        self.def_and_process(output)

# Inputs which the program needs, inputs given within the command line.

        infile = sys.argv[1]
        outfile = sys.argv[2]
        nummode = int(sys.argv[3])
        mesh_type = sys.argv[4]
        mindel = sys.argv[5]
        maxdel = sys.argv[6]
        eps = sys.argv[7]

        # These are the optional variables which only need to be registered if
        # a volume mesh is being created. Therefore, they are defaulted to None.
        # blsurfs = sys.argv[8] if mesh_type[0].lower() == 'v' else None
        # blthick = sys.argv[9] if mesh_type[0].lower() == 'v' else None
        # bllayers = sys.argv[10] if mesh_type[0].lower() == 'v' else None
        # blprog = sys.argv[11] if mesh_type[0].lower() == 'v' else None

        # Calls the CadToMesh class
        c2m = CadToMesh(infile, outfile, nummode, mesh_type, mindel, maxdel, eps)
                  # blsurfs=blsurfs,blthick=blthick, bllayers=bllayers, blprog=blprog)

        # Executes the CadToMesh Class.
        c2m.cad_to_mesh()
```

LoadCAD.py works in a very similar way to Module_Create.py, making use of Object Orientated Programming to form the main basis of the code. LoadCAD.py also makes heavy use of the Factory Pattern Method, in the same way as Module_Create.py to efficiently bring methods from the C++ code into the Python code.

The most notable differences compared to Module_Create.py is the modules being created, with configurations being registered from within the individual methods. The advantage of this is to make the code easier to read, and easier to modify or replicated.

The ‘decision’ method is important as it dictates what type of mesh is to be created, from the user inputs. If the user dictates that the desired mesh order is linear (order 1), the high-order surface mesh method will not be called. This prevents errors from being raised, adding to the robustness of the script.

Lines 103 -109 show the input arguments needed for the script, which are entered on the command line, using the sys.argv[] function which is native to the sys library of Python. Lines 113-116 show the optional inputs which are only required if the user selects a volume mesh with a boundary layer to be generated, if a volume mesh with a boundary layer is not being created, the user will not need to input any values for the arguments, as they are by default set to ‘None’, allowing for a simpler command line.

As a part of Nektar++, it is possible to set boundary layers with different properties, such as the rate at which the boundary layer should progress, and the thickness of the boundary layer, which is why the capacity to add in these have been left in, although they are still in development at the time of this publication.

5. Results

To ensure that both Module_Create.py and loadCAD.py were fully functional, a variety of meshes were formed, using various different inputs.

5.1. *Module_Create.py*

Below is a collection of meshes created using the Module_Create.py script, demonstrating the robustness of the program.

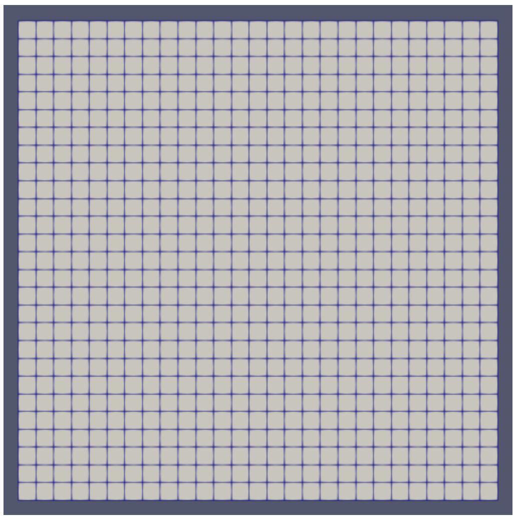


Figure 5.1 – Quadrilateral mesh generated without Chebychev nodes

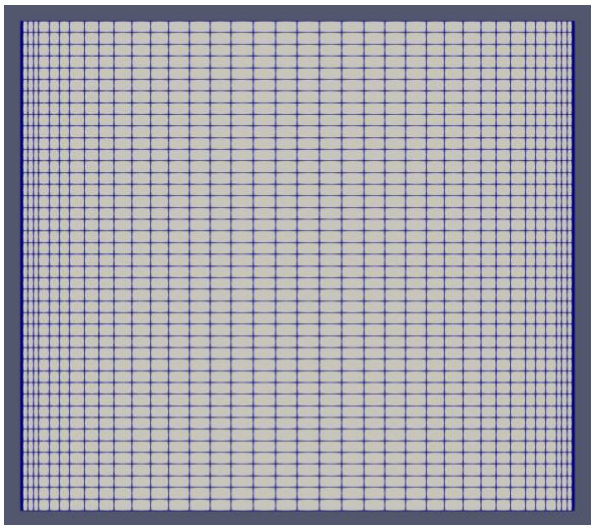


Figure 5.2 – Quadrilateral mesh generated with Chebychev Nodes

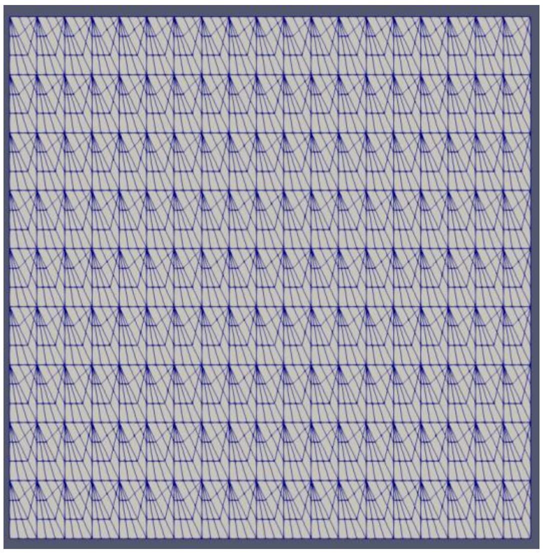


Figure 5.3 – Triangular mesh generated without Chebychev nodes

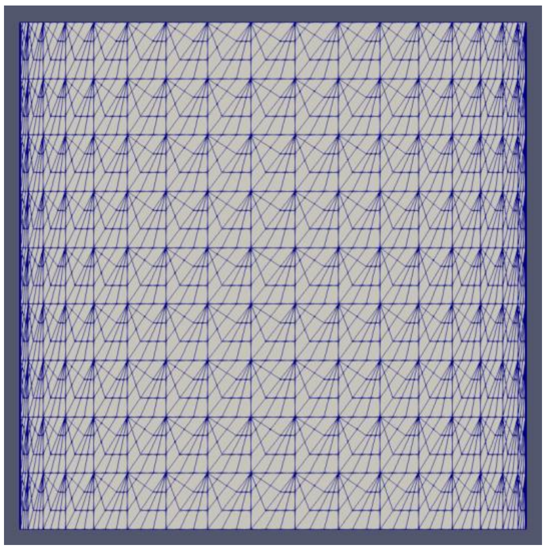


Figure 5.5 – Triangular mesh generated with Chebychev nodes

5.2. LoadCAD.py

LoadCAD.py can be used for a wider variety of things and is a more important piece for within a pipeline within Nektar++. The example below shows two meshes which are generated from a 3-dimensional sphere which can be located within the Test directory within the Nektar++ files. The first figure (Figure 5.5) shows a linear mesh which has been generated using loadCAD.py, as you can see, the sphere has many flat sections which are joined together. Whereas in the second figure (Figure 5.6), you can see that the geometry of the sphere is much more spherical, as it has an order of 15. This is achieved by using the high-order meshing capabilities of NekMesh.

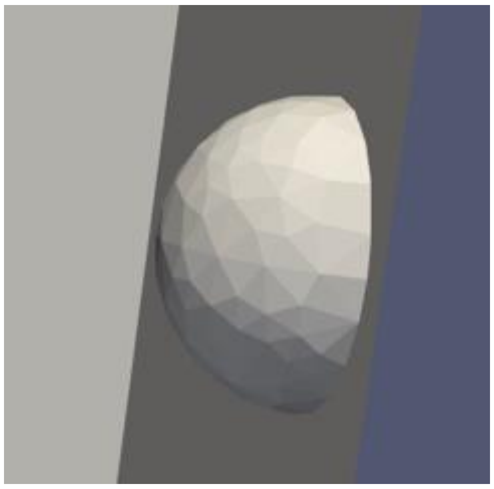


Figure 5.5 – Linear mesh generated on 3d_sphere.stp

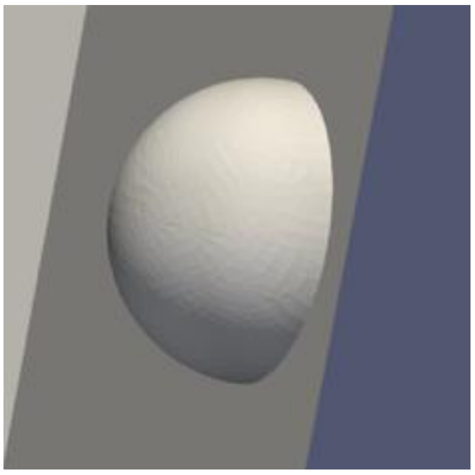


Figure 5.6 – 15th order mesh generated on 3d_sphere.stp

To view the mesh on the mesh once the .xml file had been created, the FieldConvert utility from within Nektar++ was used. This converted the .xml file into a file which Paraview could open, a .vtu.

Once the mesh was created, to ensure that loadCAD.py was creating a mesh which would allow for an accurate solution to be found, a simple solver was run on the 15th order 3d_sphere file. The results are shown below in Figure 5.7.

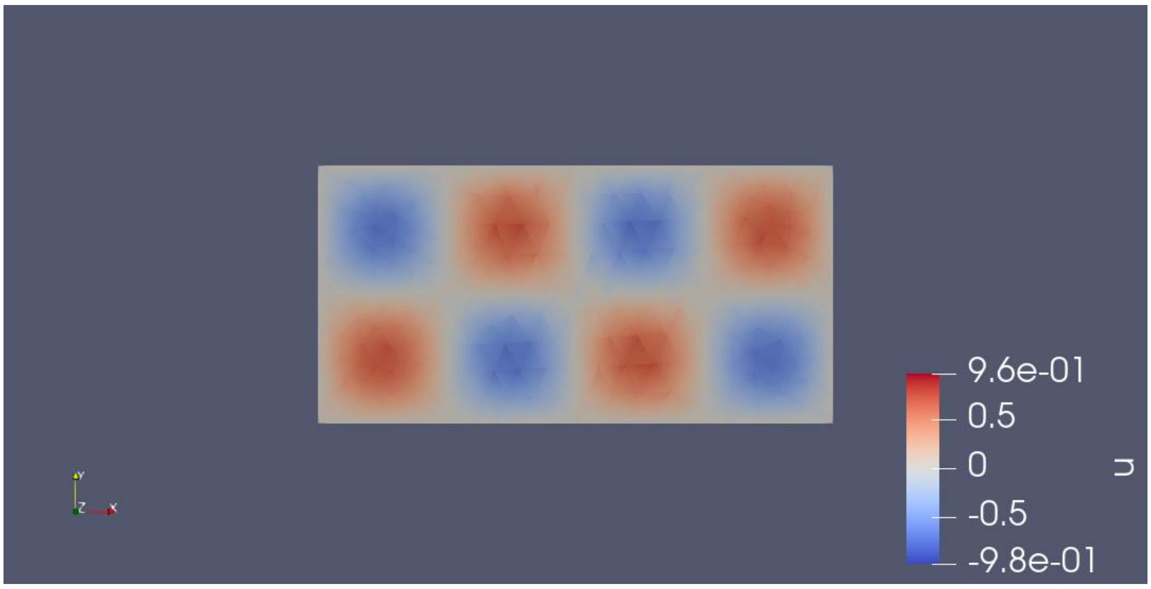


Figure 5.7 – Simulation results using LoadCAD.py

The simulation was run again, without using LoadCAD.py, and only using software which was developed using C++. The results were found to be identical, which resulted in the conclusion that the results can be assumed to be reliable, and LoadCAD.py works. The solution for the problem when it was solved without using LoadCAD.py is shown below in Figure 5.8.

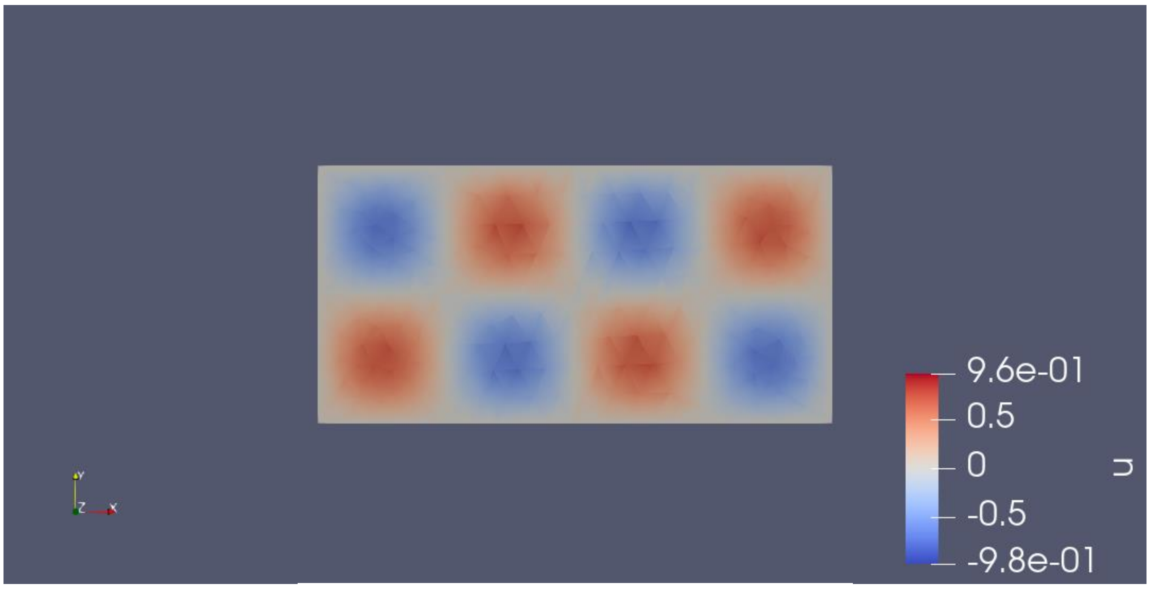


Figure 5.8 -Simulation results without using LoadCAD.py

6. Conclusion

To conclude, the objectives (outlined in Section 1.5) of the project were fulfilled, with an expansion to allow for users of the Nektar++ platform to gain easier access to the NekMesh branch of the software.

With two programs being developed to allow for users to generate either a two-dimensional mesh from scratch or be able to use Nektar++ as part of a pipeline to generate a high-order mesh from a CAD file. Following the approval of the developer team of Nektar++, both programs will be added to the next release of Nektar++, allowing for users to make use of the extended Python interface of the software.

Documentation on the developed software, for both the users and developers alike, has been written and submitted to a Project Leader of the Nektar++ team using LaTeX. The user guide has been updated with instructions on how to use the developed software, including commands and arguments which are needed to run them. The developer guide has been updated to state how the scripts are used, and states further development goals of the programs.

7. Project management, consideration of sustainability and health and safety

7.1. GANTT Chart and Project Management

Objectives were made at the start of the project, with a rough estimate of how long each individual task would take. These tasks were then put into a GANTT chart, with regular project meetings scheduled, and key milestones which had to be met (i.e. deadlines). The majority of the project was not affected by COVID-19, as it is was primarily performed on a personal computer, although regular meetings had to be adapted so that they could be carried out virtually. There was an attempt for the GANTT chart to be followed exactly. The GANTT chart can be found in Figure 7.1.

7.2. Risk Assessment

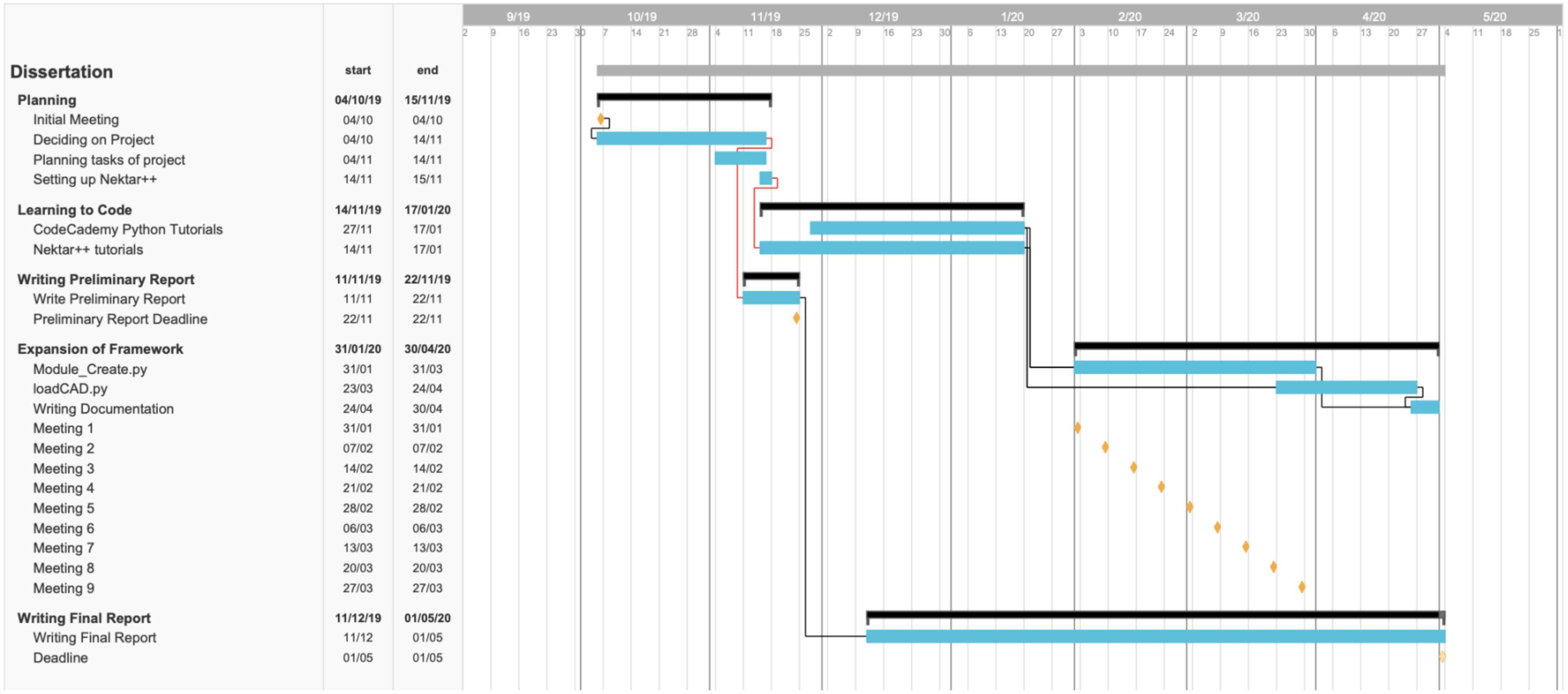
As the entire project was completed using a personal computer, there was no need to consider any real accidental risks. The only risks which were considered were the effects on long stints whilst using a computer, ensuring that breaks were taken at regular intervals to avoid eye strain and avoid adverse effects on the body. Any simulations which were performed were also done under supervision, to reduce the risk of a fire starting from a computer overheating.

Towards the end of the project, the effects of COVID-19 were also considered, and work was performed without contact to reduce the chances of catching the virus.

As the project could be used to perform simulations on projects which could be eventually made, or used for research, the risk of the simulations not being accurate must be considered. However, this is more of a risk which would be considered by somebody using the software, not a risk which must be over analysed for the development of the software. This risk was still minimalised by comparing the results from the developed software, which returned identical results to established software.

7.3. Environmental Impact

There was relatively little thought towards the environmental impact of the project, as nothing was being made. The only things which were considered were being cautious to turn off the computer when it was not being used, to reduce the use of electricity and overall lower the carbon footprint. Travel to and from meeting was done exclusively on foot to further reduce carbon footprint over the course of the project.



Dissertation	start	end
Planning	**04/10/19**	**15/11/19**
Initial Meeting	04/10	04/10
Deciding on Project	04/10	14/11
Planning tasks of project	04/11	14/11
Setting up Nektar++	14/11	15/11
Learning to Code	**14/11/19**	**17/01/20**
CodeCademy Python Tutorials	27/11	17/01
Nektar++ tutorials	14/11	17/01
Writing Preliminary Report	**11/11/19**	**22/11/19**
Write Preliminary Report	11/11	22/11
Preliminary Report Deadline	22/11	22/11
Expansion of Framework	**31/01/20**	**30/04/20**
Module_Create.py	31/01	31/03
loadCAD.py	23/03	24/04
Writing Documentation	24/04	30/04
Meeting 1	31/01	31/01
Meeting 2	07/02	07/02
Meeting 3	14/02	14/02
Meeting 4	21/02	21/02
Meeting 5	28/02	28/02
Meeting 6	06/03	06/03
Meeting 7	13/03	13/03
Meeting 8	20/03	20/03
Meeting 9	27/03	27/03
Writing Final Report	**11/12/19**	**01/05/20**
Writing Final Report	11/12	01/05
Deadline	01/05	01/05

Figure 7.1 – GANTT Chart of the Project

References

[1] D. Moxey, “Spatio-Temporal Dynamics in Pipe Flow,” Univeristy of Warwick, Warwick, 2011.

[2] D. Moxey and e. all, “Nektar++: enhancing the capability and application of high-fidelity spectral/hp element methods,” Cornell University, New York, 2019.

[3] Nektar ++, “User Guide (latest),” 9 December 2019. [Online]. Available: http://doc.nektar.info/userguide/latest. [Accessed 19 April 2020].

[4] M. Turner, “High-order mesh generation; current and future developments within Nektar++,” Nektar++, 16 October 2015. [Online]. Available: https://www.nektar.info/wp-content/uploads/2015/10/16-michael-turner.pdf. [Accessed 28 April 2020].

[5] Pointwise, “Another Fine Mesh - About Pointwise,” 2020. [Online]. Available: https://blog.pointwise.com/about/. [Accessed 21 April 2020].

[6] Pointwise, “About,” 2020. [Online]. Available: https://www.pointwise.com/about/. [Accessed 21 April 2020].

[7] D. Moxey, “NekMesh: an open-source high-order mesh generator,” 1 November 2016. [Online]. Available: https://davidmoxey.uk/assets/talks/2016-11-dipart.pdf. [Accessed 21 April 2020].

[8] F. F. Buscariolo, “Formula One front wing simulation with Nektar++,” 2020. [Online]. Available: https://www.nektar.info/formula-one-front-wing-simulation-with-nektar/. [Accessed 19 April 2020].

[9] G. E. Karniadkis and S. Sherwin, Spectral/hp Element Methods for Computational Fluid Dynamics, 2nd Edition ed., London: Oxford University Press, 2005.

[10] E. Juda, “Development of a Python Interface to Nektar++,” Department of Aeronautics Imperial College London, London, 2018.

[11] M. Turner, “High-Order Mesh Generation for CFD Solvers,” Imperial College London Faculty of Engineering Department of Aeronautics, London, 2017.

[12] P. Frey, “The Finite Difference Method,” 9 May 2017. [Online]. Available: https://www.ljll.math.upmc.fr/frey/cours/UdC/ma691/ma691_ch6.pdf. [Accessed 14 April 2020].

[13] esrd, “A Brief History of FEA,” 2019. [Online]. Available: https://www.esrd.com/simulation-technology/brief-history-of-fea/. [Accessed 15 April 2020].

[14] D. Gottlieb and S. Orszag, Numerical Analysis of Spectral Methods: Theory and Applications, 1st ed., Philadelphia: SIAM, 1977.

[15] P. Schlatter, “Spectral Methods,” KTH Royal Institute of Technology in Stockholm, Stockholm, 2009.

[16] A. Patera, “A spectral element method for fluid dynamics: Laminar flow in a channel expansion,” *Journal of Computational Physics,* vol. 54, no. 3, pp. 468-488, 1984.

[17] H. Xu, C. Cantwell, C. Monteserin, C. Eskilsson, A. Engsig-Karup and S. Sherwin, “Spectral/hpelementmethods:Recentdevelopments,applications, and perspectives,” The Authors, London/Lyngby/Aalborg/Borås, 2017.

[18] W. L. Hosch, “Quadrature,” 2006. [Online]. Available: https://www.britannica.com/science/quadrature-mathematics. [Accessed 17 November 2019].

[19] E. W. Weisstein, “Gaussian Quadrature,” 2019. [Online]. Available: http://mathworld.wolfram.com/GaussianQuadrature.html. [Accessed 17 November 2019].

[20] ParthManiyar, “Octree | Insertion and Searching,” 2020. [Online]. Available: https://www.geeksforgeeks.org/octree-insertion-and-searching/. [Accessed 28 April 2020].

[21] B. Stroustrup, “Bjarne Stroustrup's FAQ,” 2020. [Online]. Available: http://www.stroustrup.com/bs_faq.html#why. [Accessed 7 April 2020].

[22] History Computer, “Simula,” 2007. [Online]. Available: https://history-computer.com/ModernComputer/Software/Simula.html. [Accessed 7 April 2020].

[23] TekSlate, “Advantages and Disadvantages of C Language,” 2020. [Online]. Available: https://tekslate.com/advantages-c-language. [Accessed 7 April 2020].

[24] L. Tung, “Programming languages: Python developers now outnumber Java ones,” 2019. [Online]. Available: https://www.zdnet.com/article/programming-languages-python-developers-now-outnumber-java-ones/. [Accessed 7 April 2020].

[25] S. Bjarne, “C++ Applications,” 2019. [Online]. Available: http://www.stroustrup.com/applications.html. [Accessed 7 April 2020].

[26] P. Elance, “History of Python,” 2019. [Online]. Available: https://www.tutorialspoint.com/history-of-python. [Accessed 10 April 2020].

[27] G. van Rossum, Interviewee, *A Conversation with Guido van Rossum.* [Interview]. 20 January 2003.

[28] D. Robinson, “The Incredible Growth of Python,” 2017. [Online]. Available: https://stackoverflow.blog/2017/09/06/incredible-growth-python/. [Accessed 10 April 2020].

[29] K. Matthews, “6 Reasons Why Python is Suddenly Super Popular,” 2017. [Online]. Available: https://www.kdnuggets.com/2017/07/6-reasons-python-suddenly-super-popular.html. [Accessed 10 April 2020].

[30] Python, “Python 2.7,” 2015. [Online]. Available: https://legacy.python.org/download/releases/2.7/. [Accessed 10 April 2020].

[31] Python, “Python 3.0 Release,” 2020. [Online]. Available: https://www.python.org/download/releases/3.0/. [Accessed 10 April 2020].

[32] L. Abrams, “Python 2.7 Reaches End of Life After 20 Years of Development,” 2020. [Online]. Available: https://www.bleepingcomputer.com/news/software/python-27-reaches-end-of-life-after-20-years-of-development/. [Accessed 10 April 2020].

[33] Dataflair Team, “Python Applications - Unleash the power of Python,” 2019. [Online]. Available: https://data-flair.training/blogs/python-applications/. [Accessed 10 April 2020].

[34] PyPI, “Find, install and publish Python packages with the Python Package Index,” 2020. [Online]. Available: pypi.org. [Accessed 10 April 2020].

[35] Techopedia, “Glue Language,” 2016. [Online]. Available: https://www.techopedia.com/definition/19608/glue-language. [Accessed 10 April 2020].

[36] Vartika02, “Disadvantages of Python,” 2020. [Online]. Available: https://www.geeksforgeeks.org/disadvantages-of-python/. [Accessed 11 April 2020].

[37] H. Stutter and A. Alexandrescu, C++ Coding Standards: 101 Rules, Guidelines, and Best Practices, 1st Edition ed., Addison-WEsley Professional, 2004.

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

[39] Python, “PEP 8 -- Style Guide for Python Code,” 2020. [Online]. Available: https://www.python.org/dev/peps/pep-0008/. [Accessed 26 April 2020].

[40] J. d. Guzman and D. Abrahams, “Boost.Python Tutorial,” 2005. [Online]. Available: https://www.boost.org/doc/libs/1_68_0/libs/python/doc/html/tutorial/index.html. [Accessed 11 April 2020].

[41] Boost C++ Libraries, “Boost Library Documentation,” 2020. [Online]. Available: https://www.boost.org/doc/libs/?view=condensed. [Accessed 12 April 2020].

[42] Python Software Foundation, “Introduction,” 2020. [Online]. Available: https://docs.python.org/2/c-api/intro.html. [Accessed 12 April 2020].

[43] D. Moxey, C. D. Cantwell, Y. Bao, A. Cassinelli, G. Castiglioni, S. Chun, E. Juda, E. Kaxemi, K. Lakhove, J. Marcon, G. Mengaldo, D. Serson, M. Turner, H. Xu, J. Perió and R. Kirby, “Nektar++: enhancing the capability and application of high-fidelity spectral/hp element methods,” 8 June 2019. [Online]. Available: https://arxiv.org/abs/1906.03489.

[44] techopedia, “Glue Language,” 2016. [Online]. Available: https://www.techopedia.com/definition/19608/glue-language. [Accessed 19 April 2020].

[45] TIOBE, “TIOBE Index for April 2020,” 2020. [Online]. Available: https://www.tiobe.com/tiobe-index/. [Accessed 20 April 2020].

[46] Nektar++, “Nektar++ Developer Guide,” 16 May 2019. [Online]. Available: https://gitlab.nektar.info/nektar/developer-guide/-/tree/master/library. [Accessed 22 April 2020].

[47] esrd, “A Brief History of FEA,” 2019. [Online]. Available: https://www.esrd.com/simulation-technology/brief-history-of-fea/. [Accessed 16 November 2019].

[48] T. English, “A Look at the History of Computational Fluid Dynamics,” 2020. [Online]. Available: http://shortsleeveandtieclub.com/a-look-at-the-history-of-computational-fluid-dynamics/. [Accessed 23 April 2020].