2011-08-10

SMB-Interp: an N-Th Order Accurate, Distributed Interpolation Library

Stephen Mardson McQuay
Brigham Young University - Provo

SMBInterp: an Nth-Order Accurate, Distributed Interpolation Library

Stephen M. McQuay

A thesis submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Mechanical Engineering

Brigham Young University

December 2011

ABSTRACT

SMBInterp: an Nth-Order Accurate, Distributed Interpolation Library

Stephen M. McQuay
Department of Mechanical Engineering, BYU
Master of Science

The research contained herein yielded an open source interpolation library implemented in and designed for use with the Python programming language. This library, named `smbinterp`, provides an interpolation to an arbitrary degree of accuracy. The library is parametric in that is can take input from the user to adjust the underlying interpolation mechanism. The characteristics and behavior of the library according to the adjustment of these parameters is presented herein, as well as the results of a mesh resolution study depicting the accuracy obtained by the library.

The `smbinterp` library was designed with parallel computing environments in mind. The library includes modules that allow for its use in high-performance computing environments. These modules were implemented using built-in Python modules to simplify deployment. This implementation was found to scale linearly approximately 180 participating compute processes.

The `smbinterp` library was designed to be mesh agnostic. A plugin system was implemented that allows end users to conveniently and consistently present their numerical results to the library for rapid prototyping and integration. Two plugins are provided as examples and for documentation of the plugin mechanism.

Keywords: Stephen M. McQuay, `smbinterp`, N-th-order accurate general interpolation, distributed calculation schemes, multiphysics simulation

ACKNOWLEDGMENTS

I would like to thank Dr. Steven E. Gorrell for his willingness to advise me, his patient attitude, and understanding during the lengthy process of compiling this research. His kind motivation and technical assistance during this time were both essential to the completion of this work, and will never be forgotten.

I am also grateful for the friendly and crucial assistance provided by Marshall Galbraith which helped clarify the complicated parts of the implementation of the numerical method used herein. Also, I am grateful for Alex Esplin for his help explaining the more esoteric computer science-related concepts, and Matthew Peet whose blog entry helped put an end to a long stretch of debugging.

I am especially grateful to have performed this research during a time when information is so freely shared and readily available; I am truly indebted to all of the contributors to the Python and Scipy projects. I would also like to acknowledge the engineers in the aerospace group at Pratt & Whitney for the contribution of the research topic and for the partial funding provided at the beginning of this research.

I am particularly thankful for the incessant nagging, computer assistance, and editing help provided in large helpings by my father, Dr. Mardson Q. McQuay, who, despite his abnormally refined technical prowess, was surprised to find out that people these days still use `vi` and LaTeX.

Lastly, no acknowledgment is complete without an unfairly terse and obscenely understated thanks to the author's wife; Vanessa, thank you.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

NOMENCLATURE

n	Total number of points in cloud
i	Index for a point in cloud
V	Set of all points P in the cloud of points from a destination mesh
P_i	each point in V, the set $P_0, P_1, \ldots, P_n$
N	spatial dimensionality (two-dimensinal space, or three-dimensional space)
ξ_i	Spatial locations for i-th point in V
Ξ	Spatial location to which interpolation is performed
(x,y,z)	Spatial coordinates
ϕ_j	Barycentric coordinates, also linear interpolant, also basis functions
$\triangle R$	The simplex named R
R_j	A point in a simplex $\triangle R$
A_j	Area j, triangular portion of the simplex $\triangle R$
S_k	A point in the neighborhood of Ξ not in $\triangle R$
m	Total number of points in $S_1, S_2, \ldots, S_m$
q	Physical quantity of interest (e.g. temperature)
$q(\xi)$	Value of q at position ξ. May be exact or interpolated
$q(\Xi)$	Interpolated value of q at Ξ
q_{linear}	The linear interpolant of $q(\Xi)$
$f(\Xi)$	Least Squares approximation of error terms
a,b,c	Three unknowns in equation $f(\Xi)$, calculated using a least squares method
A	Transpose of (a,b,c)
B	Matrix used in least squares approximation of error involving spatial locations of extra points S_k
$B^T B$	Covariance matrix
w	Vector used in least squares calculation, involving the values of $q(S_k)$
ν	Order of requested interpolation, quadratic,cubit,etc.
ε	Error of an interpolation
μ	Number of vertexes in a destination mesh

CHAPTER 1. INTRODUCTION

As engineers attempt to find numeric solutions to large physical problems, simulations involving multiple physical models or phenomena, called multiphysical simulation, must be employed. These multiphysical simulations involve the coupling of disparate computer codes [1]. When modeling physically different phenomena the numeric models used to find solutions to these problems employ meshes of varying topology and density in their implementation. For example, the unstructured/structured mesh interfaces seen in the combustor/turbo machinery interface [2], or the coupling of Reynolds-Averaged Navier-Stokes and Large Eddy Simulation (RANS/LES) CFD codes in Computational Fluid Dynamics (CFD) [3]. A similar situation with disparate meshes arises in the analysis of helicopter blade wake and vortex interactions, as for example when using the compressible flow code SUmb and the incompressible flow code CDP [4]. When this is the case, and the mesh elements do not align, the engineer must perform interpolation in the direction of the flow of information: from the upstream code to the downstream code, or bi-directionally.

The Center for Integrated Turbulence Simulation (CITS) at Stanford University has developed an integrated multi-code simulation framework called CHIMPS. CHIMPS stands for *Coupler for High-performance Integrated Multi-Physics Simulations* [5]. CHIMPS is a Fortran/Python Application Programming Interface (API) that efficiently handles three-dimensional physical data look-up and linear interpolation across compute nodes, or servers, containing parts of computational domains using the Message Passing Interface (MPI) library [6, 7]. This framework gives engineers an interface for coupling High Performance Computing (HPC) codes of disparate types that run on a distributed cluster.

According to the authors of CHIMPS, the implementation of frameworks of this variety present two main obstacles: efficient, distributed data look-up, and accurate interpolation. They provide an elegant solution that efficiently locates data in a distributed environment. However, they acknowledge that “although linear (bi-linear, tri-linear) interpolation is relatively straightforward to implement, it is unable to guarantee the accuracy, conservativeness, and stability of the coupled solution except in the limit of an infinitely fine mesh" [5].

Researchers at Pratt & Whitney have suggested the use of a framework similar to CHIMPS as a mechanism to assist with high-performance interpolation during multiphysics simulations. In fact, the use of the CHIMPS API to combine two in-house CFD codes has already been demonstrated. However, there is a concern with the potential loss of accuracy with the linear interpolation method provided by CHIMPS, specifically when dealing with two meshes of differing types (structured/unstructured), or meshes with disparate mesh densities.

One way to improve the accuracy of the solution is to increase the complexity of the interpolation used. There exist many higher-order interpolation methods, but most are not designed to be used on grids with irregular topologies. These concepts are briefly explored in chapter 2.

In its latest implementation, the CHIMPS framework can provide the user with higher-order interpolation results, but only if the end user provides the framework with gradient information [8]. This method is involved, and obtaining the higher-order terms may be prohibitive. A need exists for an improved, automatic, and general solution for obtaining more accurate interpolation, especially when the required gradient information is not available a priori.

Work has been accomplished by Galbraith [9] in the development of a general interpolation algorithm named AVUSINTERP (Air Vehicles Unstructured/Structured Interpolation Tool). The method employed utilizes a generalized interpolation that yields either linear or quadratic interpolation using a least squares error correction of a linear interpolation as described by Baker [10]. AVUSINTERP provides the engineer with a more generic interpolation scheme, but it was not implemented in a parallel fashion, nor does it allow for the engineer to arbitrarily choose the order of the interpolation.

There exists the need for a tool that can perform interpolation to an arbitrary degree of accuracy, and that can be parallelized to take take advantage of modern HPC architectures. The research work described herein includes the development of a method to provide engineers with an interpolation library that is itself a union of the best parts of the aforementioned tools. Namely, this research provides an API, named `smbinterp`, that implements the N-th order accurate interpolation of a physical value in both two- and three-dimensional space. Previous implementations of similar interpolation schemes were only implemented to third order accuracy [9]. Interpolation using the API described in this research can be performed to any order of accuracy, and in both two and three spatial dimensions.

The library described herein was implemented with mesh generality in mind. The API has been implemented in an abstract way, where plugins are developed that implement class methods as defined by the API (see chapter 3 for details). As a consequence of this plugin-based architecture, the API is capable of working with topologically varied meshes. This means that the library can be used with structured or unstructured grids, and also on a cloud of points with no predefined mesh topology. This research provides examples of the development and use of plugins that implement this interface for an unstructured mesh produced by Gmsh [11], and clouds of points with no predefined underlying mesh structure.

While the API can provide the end user with improved interpolation results, it is not designed to function correctly without a good underlying mesh. Meshes for physical simulations are typically designed after having an a priori understanding of the underlying equations being solved by the physical models. The onus of mesh design lies typically upon the engineer responsible for a given physical domain. It is assumed that the meshes being employed in a multiphysics simulation appropriately capture changes in the underlying physical quantities. The design of these meshes is outside of the scope of this research.

The API was intended to be used by engineers in situations where all that is known is physical data (e.g. temperature) at two- or three-dimensional locations. If higher-order information is known (e.g. gradient quantities), then an interpolation of the variety presented by Hahn [8] could be utilized. However, this API can be used in situations where that information is expensive, difficult, or impossible to acquire. As long as the mesh was designed to properly resolve the underlying fluctuations in physical quantities, the API described here generally provides an improved interpolation by comparison with simple linear interpolation.

In order to minimize the time it takes to perform all requisite interpolations for a given simulation, the API described herein has been implemented in a way that takes advantage of high-performance, parallel computers, and has been demonstrated to scale in a distributed environment. While there are more elegant parallel implementations for the solution of data look-up, this research was designed to perform well according to the hardware limitations of the day. When CHIMPS was originally implemented, the average amount of ram per compute core was on the order of hundreds of Megabytes per compute core [5]. Current HPC architecture currently yields an order of magnitude more RAM per core (sometimes substantially more), and the library was designed to take advantage of this fact. This research was implemented to utilize memory-intensive spatial data structures that are now reasonable to be used on modern HPC systems.

While benchmarks are presented, the scope of the research was limited to the implementation and study of the proposed design. `smbinterp` is not intended to out perform other applications or libraries, nor does it purport to have a direct impact on the overall speed of a multiphysics simulation. Furthermore, the `smbinterp` library is not intended to magically improve all interpolations. One must provide the library with a mesh that is intended to capture relevant physical information. As an example a well designed CFD mesh is typically more dense in areas where physical information is changing rapidly in any spatial dimension, specifically near physical boundaries. This is typically done to appropriately resolve boundary layer flow characteristics. If information of this variety is correctly captured, `smbinterp` does provide a good interpolation.

Chapter 2 contains a review of background concepts and literature requisite to understand the context of the problem. This includes information on interpolation and various interpolation schemes, a discussion on various spatial data structures, and a high-level discussion of various parallelization schemes. Afterward an explanation of the research method is presented in chapter 3. This includes an explanation of the main numerical method used in the `smbinterp` library and rationale for design decisions, including detail of how the API implements the general interpolation scheme, and how the plugin architecture and network queue were designed and implemented. Next, in chapter 4 results and benchmarks of the implementation are presented, including a discussion of these results. Finally, conclusions and recommendations are presented in chapter 5.

CHAPTER 2. LITERATURE REVIEW

This chapter comprises a review of relevant literature consulted during the course of this research. Various methods of interpolation were reviewed and assessed, and an overview of the benefits and shortcomings of each method is presented here. Next, a review of concepts relevant to the selection of appropriate spatial data structures researched is presented, followed by a review of literature pertaining to the distribution of interpolation workload.

2.1 Interpolation

When numerically solving a set of partial differential equations used to model a particular physical phenomenon, engineers discretize the spatial domain into a mesh and solve for physical properties of interest at discrete locations. Meshes of varied density and topology are used when coupling meshes that are designed to solve two disparate physical models. At solution boundaries and at mesh overlap, the coupled solution is dependent on information flowing from the upstream domain to the downstream domain, vice versa, or in both directions. When the points of the two domains do not overlap exactly, an approximation for the quantity of interest at locations in the downstream mesh must be calculated. Interpolation is used to find values at spatial locations not exactly resolved by the mesh, and thereby providing the required information at the interface.

Because the mechanisms employed in this research come from various fields which have their own nomenclature and definitions, the meaning of one-, two-, and three-dimensional data shall be explicitly defined for clarity. For the purposes of this research we are interested in the value of a physical quantity at a specific spatial location. Therefore, when discussing a method that works for one-dimensional data, it is meant one spatial dimension, x, and one variable representing a physical quantity, q, typically visualized in a two-dimensional plot where the abscissa represents the spatial location, and the ordinate represents the physical quantity, are shown in figure 2.1. By extension, a two-dimensional data set consists of two spatial dimensions (x, y) plus one physical

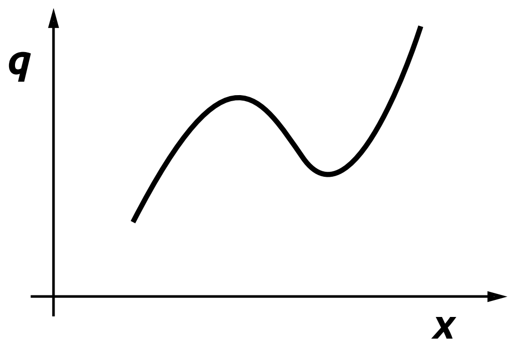


Figure 2.1: Visualization of a One-Dimensional Data Set

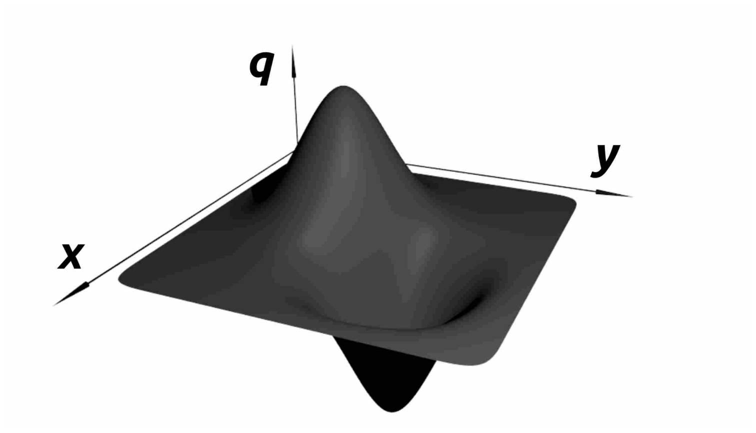


Figure 2.2: Visualization of a Two-Dimensional Data Set

quantity variable (q) shown in figure 2.2, and so on for three-dimensional data. Data of higher dimensionality are not investigated in this work.

2.1.1 Polynomial Interpolation

Three popular methods for fitting a curve of degree n through a set of $n+1$ points, as shown in figure 2.3, are presented here. All methods require a priori knowledge of two sets of information. First, the location of a set of points $P_0, P_1, \ldots, P_n$. Secondly, parameter values of the underlying curve at those points, denoted $t_0, t_1, \ldots, t_n$, which represent the location of the point along the curve in the curve's *parameter space* $[t_0, t_n]$, typically $0 \leq t \leq 1$.

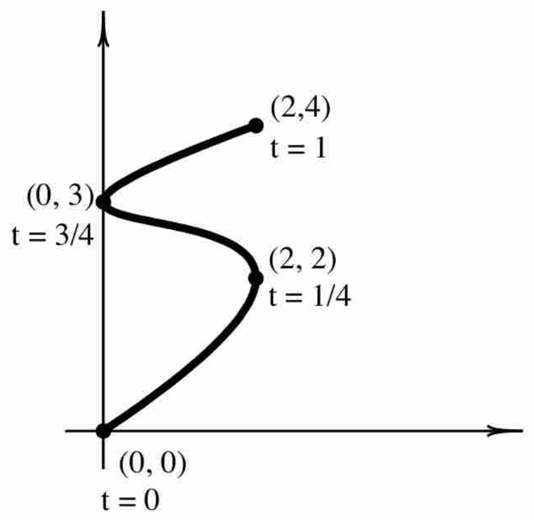


Figure 2.3: Example Data Set for Curvilinear Interpolation [12]

The first curvilinear interpolation method investigated is known as Lagrange interpolation. The Lagrange interpolation curve is defined by

$$P(t) = P_0 L_0^n(t) + P_1 L_1^n(t) + \ldots + P_n L_n^n(t) \tag{2.1}$$

$$L_i^n(t) = \prod_{j=0}^{n} \frac{(t - t_j)}{(t_i - t_j)}, j \neq i, \tag{2.2}$$

which has the property that the basis polynomials (L_i^n) are 1 at point P_n and exactly 0 at all other points in the interpolation. This property assists in obtaining an intuitive understanding of this interpolation method, in that, when evaluating the curve at point P_i, the other points in the interpolation $P_j, j \neq i$ have no weight or pull on the curve. It is also what allows the curve to interpolate point P_i at t_i [12]. When calculated explicitly from the above equations, Lagrange polynomials have $O(N^2)$ complexity, but Werner [13] provides an interesting reformulation of the method that performs with $O(N)$ complexity. The notation $O(N)$, also known as Big O Notation, comes from the field of algorithm analysis, and signifies that generally, for N inputs, an algorithm will scale linearly in time. $O(N^2)$ describes an algorithm that scales quadratically, $O(1)$ runs in constant time, and so forth.

A similar curvilinear interpolation method uses Newton polynomials. The method is similar to Lagrange interpolation in that it interpolates all points, but can be computed using a table of *divided differences*, which is, after initial construction, computationally cheap ($O(1)$) since the values of previous calculations are cached and reused [12].

Another method for fitting a curve to a set of points is known as the method of Undetermined Coefficients [12]. First the fundamental equation of the curve is expressed. As an example, consider the curve in figure 2.3. In order to fit a curve of third degree to these data it could be assumed that the underlying equation is a third-degree polynomial curve. Consider the x component of the Bernstein polynomial form of a third-degree curve:

$$x = a_0(1-t)^3 + 3a_1t(1-t)^2 + 3a_2t^2(1-t) + a_3t^3. \tag{2.3}$$

Using the known points and parameter values, this equation is evaluated yielding a system of linear equations. Evaluating only the x component at the known points yields

$$\begin{aligned}
x_0 &= a_0(1-t_0)^3 + 3a_1t_0(1-t_0)^2 + 3a_2t_0^2(1-t_0) + a_3t_0^3 \\
x_1 &= a_0(1-t_1)^3 + 3a_1t_1(1-t_1)^2 + 3a_2t_1^2(1-t_1) + a_3t_1^3 \\
x_2 &= a_0(1-t_2)^3 + 3a_1t_2(1-t_2)^2 + 3a_2t_2^2(1-t_2) + a_3t_2^3 \\
x_3 &= a_0(1-t_3)^3 + 3a_1t_3(1-t_3)^2 + 3a_2t_3^2(1-t_3) + a_3t_3^3.
\end{aligned} \tag{2.4}$$

Lastly, the unknown coefficients a_0, a_1, a_2 and a_3 are obtained by solving this system of linear equations. Solving for a_i and performing a similar computation for the y Bernstein polynomial yields the Bézier control polygon illustrated in figure 2.4.

These curvilinear interpolation methods suffer from shortcomings that make them inadequate for the general interpolation sought after in this research. If a dense collection of points that follow a curve of a particular degree are to calculate the parameters of a curve of an even higher degree, then the curve may exhibit the undesirable side effect of oscillatory artifacts known as Runge's phenomenon [14]. Runge's phenomenon is also manifested when trying to interpolate a high order curve with points that are equidistant. Berrut [15] discusses the benefits and implementation of Barycentric interpolation, which is a variant of Lagrange polynomial interpolation that

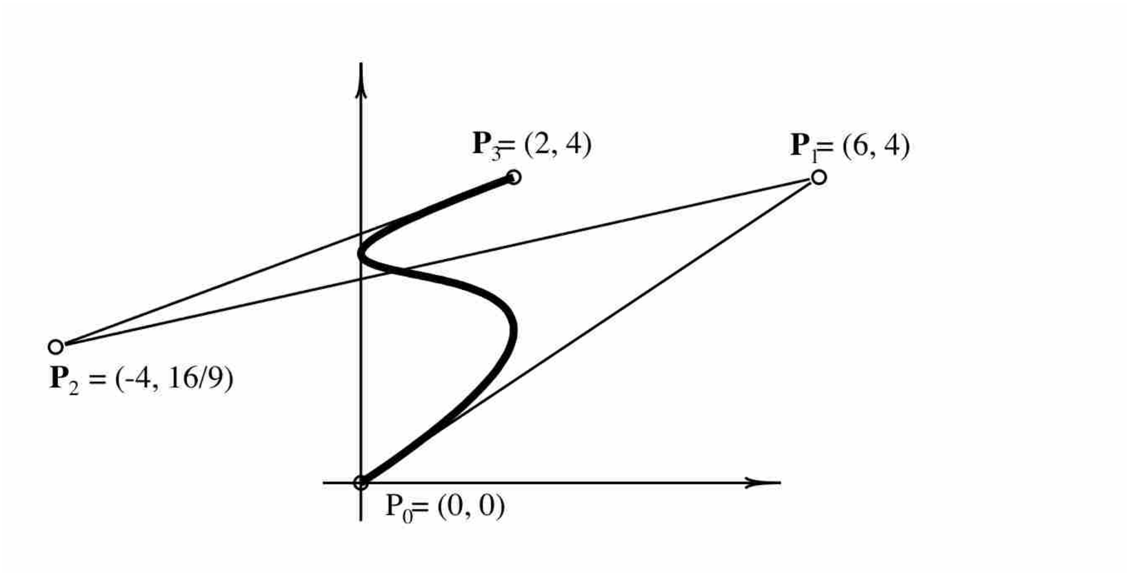


Figure 2.4: Control Polygon of a Bézier Curve Calculated Using the Method of Undetermined Coefficients [12]

yields fast and stable results and does not suffer from Runge's phenomenon. Also, when it is possible to adjust the spacing of the points used during interpolation, Chebyshev spacing can be used to mitigate oscillatory effects [16]. Therefore, the effects of Runge's phenomenon in the Lagrange method may be mitigated by adjusting the spacing of the points, by manipulating Lagrange polynomials through barycentric interpolation, and by sampling the domain in a more degree-appropriate fashion.

Another shortcoming of these methods is that both the curve's parameter values t_i at each of the points P_i and the spatial locations of P_i must be known a priori. The interpolated value is strongly dependent on the parameter values, and if t is chosen arbitrarily or inappropriately the interpolation may be incorrect. Also, while these methods apply to curves in two- and three-spatial dimensions, they are only applicable to curves, whereas the general interpolation in this research must apply to both surfaces and fields. Insight was sought by investigating these methods, but they are not known to generalize to surfaces (two-dimensional data) or fields (three-dimensional data) for meshes of irregular topologies.

Furthermore, the stencil of participating points P_i is not compact. All points P_i are required to participate in every interpolation; curvature matching is only guaranteed when using all points in the space. Shukla [17] proposes a method that yields arbitrarily high-order compact polynomial interpolations, however it is more difficult and rigorous to author a scheme that has a more compact stencil, is less computationally expensive, and yet still matches curvature [18]. The following section describes a method that overcomes a significant number of these shortcomings.

2.1.2 Butterfly Interpolation

In the field of computer graphics and animation there exist geometric surface control techniques that allow artists to manipulate a relatively small number of control points which affect the shape of a more complicated, smooth surface. While the control points of a NURBS (Non-Uniform, Rational B-Spline) surface afford the artist an abstracted, high-level control mechanism, they require that the control points of the surface exist topologically in two dimensions (typically s and t) [12, 19]. The topological rigidity in surfaces of this variety limit the artist's ability to create models with complex topologies.

As a consequence of the need to provide artists with simple-to-use yet flexible surface control mechanisms schemes have been designed to take a control net of elements, typically composed of quadrilateral and triangular elements, and recursively subdivide the elements therein until a smooth surface is formed. These schemes are categorized into two groups. Primal subdivision schemes retain the last iteration's vertexes for use in future subdivision [20]. Dual schemes discard the previous iteration's information. Primal schemes are then subdivided into two categories: approximation schemes, where the location of the original vertexes change, and interpolating schemes, where the resulting surface interpolates (passes through) the vertexes in the original control mesh. Approximating schemes are in heavy use in the animation industry [21, 22], but, as they do not interpolate the original mesh, are not directly applicable to this research. While subdivision methods exist that combine schemes of multiple varieties [23], primal schemes that interpolate the control mesh are the most applicable to this research since they can be used to fit a cloud of points in a smooth manner, and, therefore, can be used to perform continuity-enforced interpolation.

In order to understand the benefits and shortcomings of the original Butterfly scheme, a few continuity-related terms must be defined. Curves and surfaces that meet at a common vertex have C^0, or coincidental continuity. Those that demonstrate tangential continuity with other curves and surfaces are given the label C^1 continuous. Curves and surfaces that match curvature are labeled

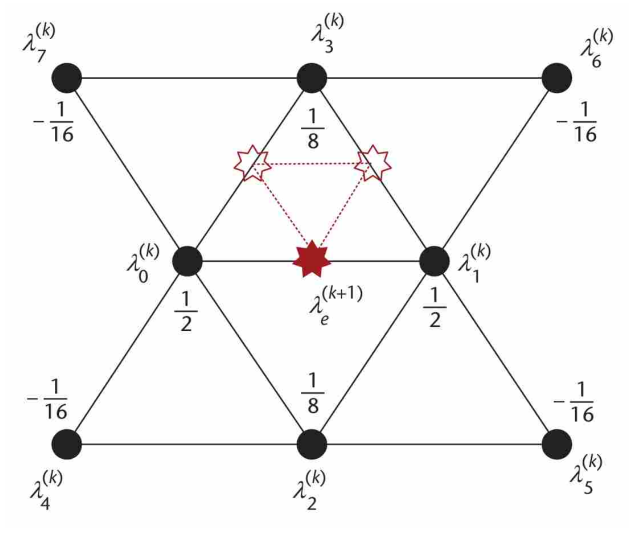


Figure 2.5: Stencil Used in the Original Butterfly Interpolation Scheme [20]

C^2 continuous [12]. Also, the number of edges that meet at a vertex determine the vertex's *valence*. *Extraordinary points* are points that have an undesirable valence.

The following surface interpolating subdivision schemes described in this review are incremental improvements upon the original Butterfly interpolation scheme described by Dyn [24]. This scheme subdivides each triangle in a triangular mesh so that at each recursive step all of the vertexes, including the original vertexes, are interpolated. Each recursive subdivision yields four new triangles per triangular patch. At each step, a new vertex $\lambda_e^{(k+1)}$ for the edge $\overline{\lambda_0^k \lambda_1^k}$ is created using neighboring vertexes $\lambda_0^k, \lambda_1^k, \ldots, \lambda_7^k$ at the k-th recursion level, as illustrated in figure 2.5. The geometric location of the new point is determined by the following formula:

$$\lambda_e^{(k+1)} = \frac{1}{2}(\lambda_0^k + \lambda_1^k) + \frac{1}{8}(\lambda_2^k + \lambda_3^k) - \frac{1}{16}(\lambda_4^k + \lambda_5^k + \lambda_6^k + \lambda_7^k). \tag{2.5}$$

In the butterfly scheme, vertexes that do not have valence six are extraordinary points. Under conditions where no extraordinary points exist, the original Butterfly scheme yields C^1 continuity. However, the original butterfly scheme yields undesirable effects around extraordinary points. The effects of extraordinary points is shown in the left image of figure 2.7. Since each vertex in a tetrahedron has valence three (not six), the original butterfly scheme yields sharp, C^0 continuous points at each vertex.

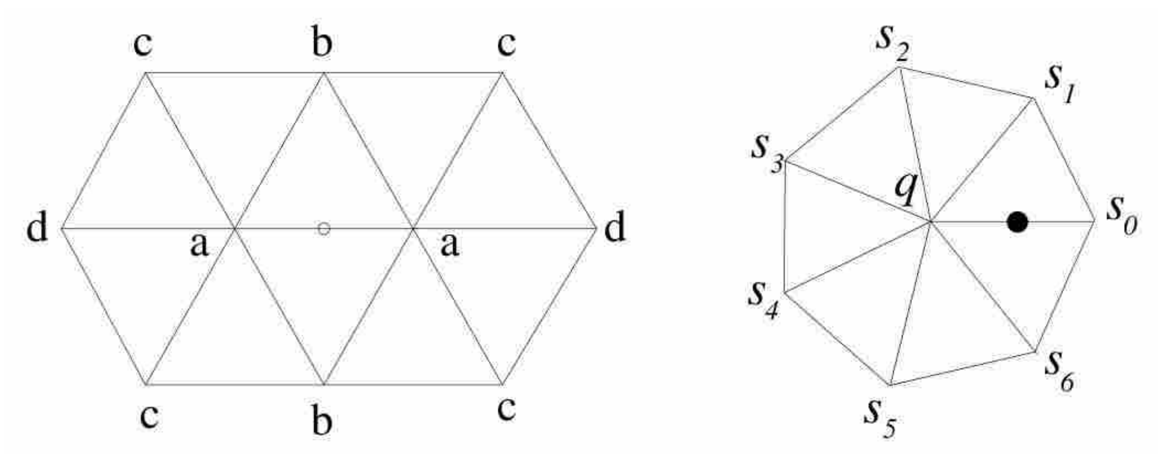


Figure 2.6: Modified Stencil Employed in Zorin's Improved Butterfly Scheme (Left), Stencil Used with Extraordinary Points (Right) [25]

Zorin [25] provides an improvement to Dyn's interpolation scheme that, while it does not improve upon the C^1 continuity of the scheme, it does yield an interpolation that is generally less sharp and creased. If the edge being subdivided has two vertexes of valence six, the Zorin scheme uses a stencil of ten vertexes as shown in figure 2.6. Three special rules exist for vertexes that do not have valence six, and are treated in full by Zorin [25].

The visual effect of the larger stencil used in the Zorin interpolation is demonstrated in the tetrahedron interpolation on the right of figure 2.7, where the interpolation is generally more smooth around the extraordinary vertexes. The general effects of extraordinary points on the butterfly scheme is also shown in figure 2.8. The image on the left is the original mesh. The image in the middle was produced using the original butterfly interpolation, and exhibits the visual artifacts affected by the extraordinary points, which are manifest in the creases and unnatural sharpness surrounding these points. The smoothing effect of the modified method is demonstrated by the model on the right, shown in figure 2.8. This image was produced by applying the modified butterfly scheme to the mesh on the left.

Yang [20] developed a significantly improved extension of the Butterfly scheme, named the Twin-Butterfly scheme. This interpolation scheme uses a much larger stencil, shown in figure 2.9. When mesh topology allows for the use of this larger stencil, the Twin-Butterfly scheme yields C^2 continuity. When unable to use the complete stencil, Yang proposes an algorithm that attempts to subdivide each face with progressively less ideal algorithms, depending on the ability of

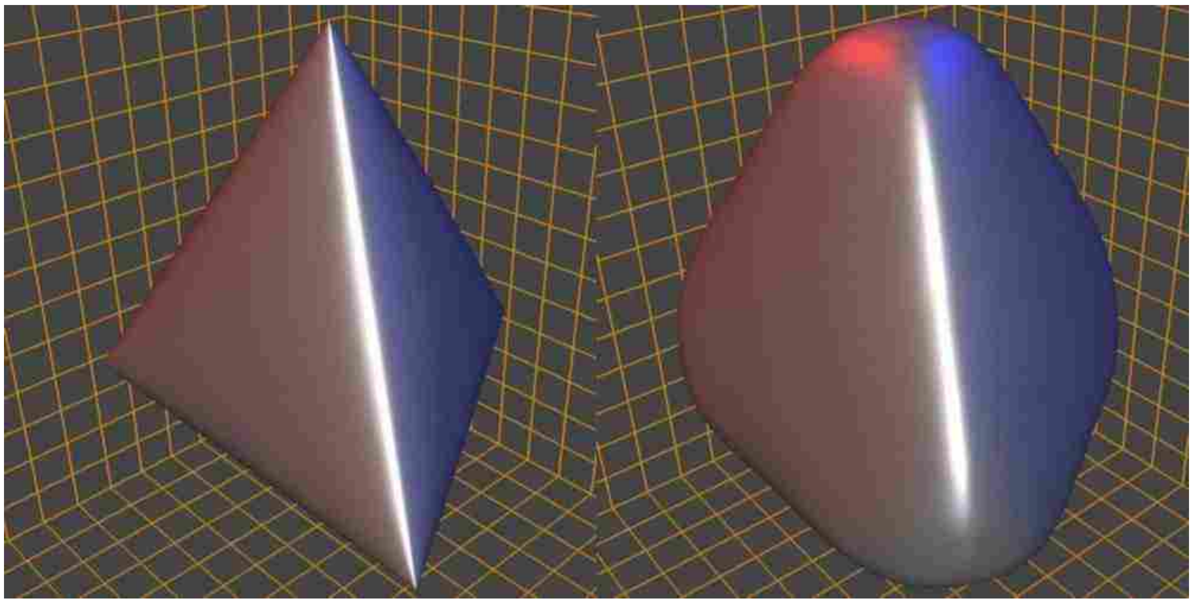


Figure 2.7: Example of the Effect of Extraordinary Points on the Original Butterfly Scheme (Left), Improvements Proposed by Zorin (Right) [25]

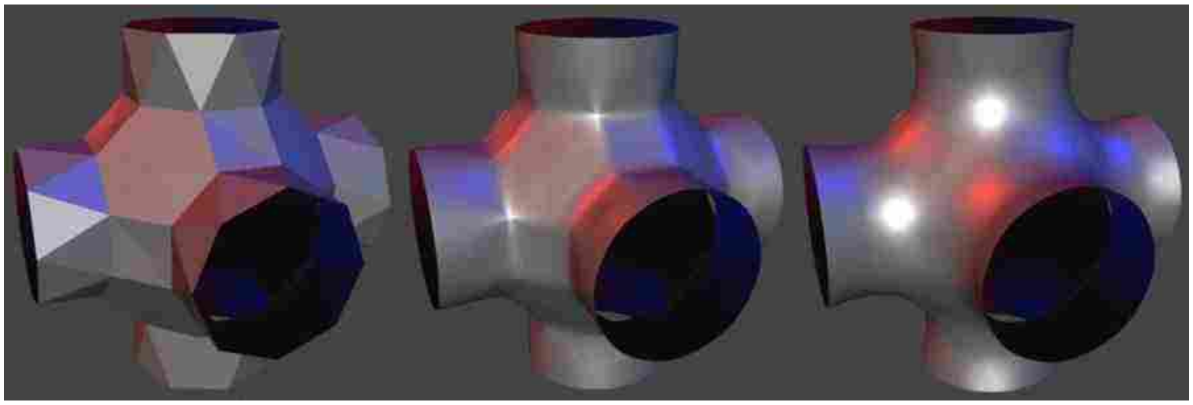


Figure 2.8: Example of the Improvement in Visual Appeal Provided by the Zorin Subdivision Scheme [25]

being able to form particular stencils with the topologically adjacent vertexes. The first attempted interpolation is the C^2 Twin-butterfly scheme, followed by Zorin's C^1 ten-point stencil, Dyn's C^1 eight-point interpolation, and then two simpler interpolations introduced by Yang. The first is called the Rhombus scheme, named for its use of the four topologically closest vertexes. The last scheme attempted always works since it is simply the geometric average of the two points in the edge being subdivided.

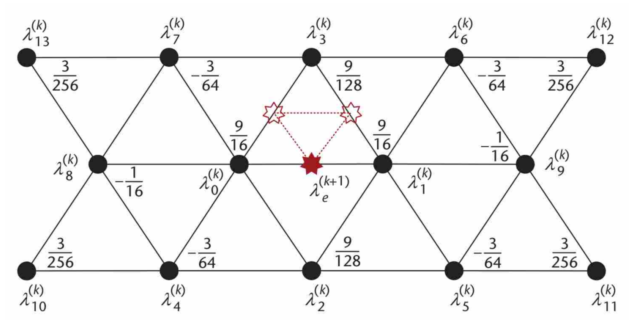


Figure 2.9: Stencil Used in the Improved Twin-butterfly Interpolation Scheme [20]

These subdivision interpolation schemes have definite advantages. First of all, they are implemented using a compact stencil. Unlike the curvilinear interpolation methods that generally need $n+1$ points to give a curve of order n, the butterfly schemes use a localized set of vertexes in each calculation. Another benefit to the Yang scheme is that, under the conditions of an ideal mesh, C^2 continuity can be obtained, degrading gracefully to C^1 and C^0 continuity.

One of the disadvantages of these methods is that at each subdivision, exponentially more vertexes are created. This increases memory consumption and calculation times exponentially, so there is a limit on how many subdivisions can be performed in a reasonable amount of time. For example, using the geometric modeling software package Blender [26], memory consumption of the repeated subdivision of a single triangular element is shown in Table 2.1 which shows that the scaling of memory is exponential. The use of a subdivision interpolation scheme would therefore rely on using a small number of subdivisions and then performing a linear interpolation against a triangular element that is “small enough“ to give accurate results.

Another disadvantage of the surface subdivision interpolation schemes is that while these subdivision schemes hold promise for use in data interpolation when $N = 2$, there exist no known three-dimensional volumetric interpolatory subdivision schemes that work on meshes with general topologies. The schemes that do exist either volumetrically interpolate rigid mesh topologies, or they they do not interpolate the original mesh. McDonnell [27] pointed out that “the original butterfly algorithm does not generalize directly to 3D,” but implemented an interpolatory scheme which requires hexahedral meshes. Bajaj [28] implemented volumetric subdivision, also restricted to hexahedral mesh elements, but the scheme was approximate, so it does not interpolate the original mesh. Chang [29] implemented a scheme for volumetric subdivision with arbitrary topologies, but

Table 2.1: $O(c^N)$ Scaling of Memory Demonstrated When Subdividing a Single Triangular Face

Subdivision	Vertex Count	Memory
0	3	5.03M
1	6	5.04M
2	15	5.04M
3	45	5.04M
4	153	5.06M
5	561	5.13M
6	2145	5.41M
7	8385	6.51M
8	33153	10.91M
9	131841	28.45M
10	525825	98.53M
11	2100225	378.68M

it was non-interpolatory. While Chang [30] later implemented a volumetric subdivision scheme for arbitrary topologies, it does not guarantee "higher order continuity across extraordinary vertexes and edges".

2.1.3 Baker's Interpolation Method

Baker [10] describes a method that works with either two-dimensional or three-dimensional data, can be implemented in a way that allows an arbitrarily high order of approximation, and places practically no limits on the underlying mesh topology. This method approximates a surface fit to neighboring points of almost any mesh topology to an arbitrary degree. It would seem that it is the correct method to employ for the mesh-agnostic, arbitrary order, $N = 2, 3$ interpolation sought after in this research. By solving a collection of reasonably sized linear systems of equations (explained in detail in chapter 3) approximate error correction terms to any arbitrary order may be obtained. Compared to other conventional techniques, the Baker interpolation method performs relatively well, but it does suffer from some limitations.

First of all, interpolation situations arise which Baker termed "pathological situations" [10]. In all of these cases, the covariance matrix used to calculate the error approximation term is singular. While Baker states that the system of equations is consistent, and will therefore always have a solution, “a non-singular covariance matrix ensures that the least squares solution is unique”. This covariance matrix is singular when more than $m-2$ of the m extra points $S_1, S_2, \ldots, S_m$ lie on a line that is C^0 with any of the vertexes in the simplex $\triangle R$, including any of the simplex’s edges (C^0 with any N R_j), explained in more detail in chapter 3. While Baker states that “For a general arrangement of points it is unlikely that these pathological situations will arise”, it assumes that an engineer will never want to arrange points in a regular fashion, which is exactly how some meshes are designed [31].

There are two approaches to solving this problem. The logic to protect against this situation could be implemented with a mesh plugin. More detail on this approach is found in chapter 3, section 3.3. The authored plugin would select the neighboring points according to an intelligent S_j selection algorithm designed to minimize the likelihood of a singular covariance matrix. Alternatively, as suggested by Galbraith [9], if the covariance matrix is singular the higher order approximations for a particular interpolation may simply be ignored, and the interpolation remains linear. However, if the singularity is due to the topology of the underlying mesh structure, e.g., the mesh is regular, the implementation of an intelligent plugin would be a superior alternative.

Lastly, this method does not guarantee continuous derivatives across the entire domain [10]. Specifically, whenever there is a change in stencil, there is potential for a discontinuity in the interpolation. While this is true, Baker’s method has been shown to provide reasonable interpolation results in practice using up to a quadratic approximation [9, 10].

2.2 Spatial Data Structures and Distributed Algorithms

As stated in chapter 1, the implementation of a general purpose interpolation library presents two main challenges: accuracy in interpolation and efficiency in spatial data querying. The use of Baker’s interpolation method addresses the challenge of interpolation accuracy in that it provides a mechanism that can be used to improve the interpolation. While the primary focus of this research is on the implementation of a powerful interpolation library, spatial data structures and distributed algorithms play an important role in the performance of said library. Both subjects are briefly addressed in this section.

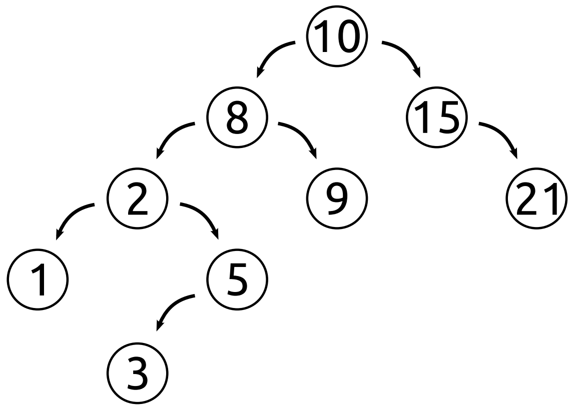


Figure 2.10: A Binary Search Tree Data Structure

2.2.1 Spatial Tree Structures

The selection of a good data structure is a key element to designing good computer software. Algorithms that are otherwise well designed could be slowed down if an incorrect data structure is used. As an example, consider searching through a list of randomly ordered, random numbers

$$[3, 1, 9, 15, 21, 8, 5, 10, 2, \ldots, M].$$

The test for the inclusion of a number in this list is on average $O(M)$. While it is possible that the number may be found early in the list in general and on average all M numbers must be inspected to test for inclusion. The number may not be in the list at all, and still all members of the list are inspected to verify this fact. However, if the numbers can be stored in order while being generated, in a tree structure as shown in figure 2.10, the average performance becomes $O(log_2(M))$. At each step of a traversal, the search continues down the half of the tree likely to contain the number. The time difference between $O(log_2(M))$ and M can be significant, especially for large M.

Similar tree data structures exist for use in spatial data indexing and searching. Recursively subdividing a spatial domain is generally referred to as binary space partitioning [32]. There exist many popular spatial partitioning trees [33–36]. Each specific variety of spatial partitioning data structure has its own benefits. As an example the spatial database extension named PostGIS to the open source database PostgreSQL, offers a column index implemented as one type of spatial data structure called an R-tree. This index allows for rapid querying of spatial data points, but does not provide mechanisms for querying nearest neighbors [37, 38]. The parallel Alternating Digital Tree

(ADT) used in CHIMPS is heavily optimized for equally distributed data across a low-memory system [5]; the implementation and benefits of an ADT are described by Bonet [39]. Other trees, such as the kd-tree and the sr-tree are good at handling data with higher dimensionality [40].

Garth [41] provides an overview of the benefits and shortcomings of three structures: octrees, kd-trees, and the celltree. In octrees, space is recursively subdivided into eight uniform octants, which is not memory efficient in the case of spatially non-uniform vertex distributions. Kd-trees, where space is subdivided at each point, "facilitates non-uniform subdivision, at the cost of generally deeper trees and a storage overhead" [41]. The celltree, authored by Garth, has many benefits. It works with unstructured grids, is numerically robust, can be used with both CPU and GPU applications, and makes optimal and adjustable use of memory.

While adding new vertexes to a kd-tree can be computationally expensive [35], the mechanism implemented in this research is expected to work on a very specific set of vertexes between long-running numeric computations. Therefore, the entire tree generally remains constant during the interpolation phase of a single iteration. The Python libraries Numpy and Scipy offer scientific data structures and algorithms for use in scientific computing [42]. NumPy provides users of the Python language with high-performance multi-dimensional arrays. Scipy is a collection of scientific algorithms that make use of the fast arrays implemented by NumPy. The interpolation library in this research makes use of these excellent tools. Scipy contains a kd-tree implementation [43], which was found to be sufficiently performant, and is the data structure selected for use in this research.

2.2.2 Distribution of Workload

Embarrassingly parallel workload is workload "for which little or no effort is required to separate the problem into a number of parallel tasks", and which typically involves little or no communication between the participating threads of execution [44]. As stated previously, the data are generally constant during the transfer of interpolated data between the upstream code and downstream code. This being the case, if data can be inexpensively replicated, then the action of interpolation can be posed in an embarrassingly parallel fashion. Each compute unit with access to the data can divide and conquer a fraction of the total workload, and, after performing an interpolation, report the result.

Two mechanisms were investigated for use in the parallelization of this library. First, the problem was posed in a fashion that mirrors the architecture commonly found in high-performance web sites of the variety of Google.com or Amazon.com. These sites are designed to distribute the task of responding to requests so an army of servers in order to handle many thousands of requests per second. Devlin [45] presents an overview of commonly used terminology in this field, and Cardellini [46] provides an overview of the state of the art, including various configurations in modern use. The configuration investigated involved the proxying of requests to a farm of servers designed to perform the interpolation being requested. Because the proxying of requests is generally far less computationally expensive than the actual interpolation calculations, this is a reasonable approach to parallelization, assuming the data can be properly replicated for each compute unit. The replication of databases is an active area of software development and implementation. Three clustering solutions for the PostgreSQL database, Slony-I, PGCluster, and Bucardo, were investigated and found to scale well with requests [47–49]. However, the only spatial indexing scheme available, PostGIS, does not implement nearest neighbors queries in three dimensions [38]. Therefore this form of data replication and proxy-based distribution of queries was found not to be usable for this research.

The Python language has recently (since version 2.6) included a module designed to assist in the implementation of parallel processing applications, named `multiprocessing` [50]. Because it does not use threads, `multiprocessing` does not suffer from the shortcomings of Python's Global Interpreter Lock, or GIL. The GIL is a Python peculiarity that does not allow for more than one thread to execute Python bytecode at any given time, which inhibits parallelism, but simplifies the thread-safe implementation of Python's built-in data structures [51]. The `multiprocessing` library circumvents this limitation by implementing a processes-based rather than thread-based approach to concurrency, and provides a collection of classes and modules that perform in a truly parallel fashion.

The `multiprocessing` library contains a `Manager` class, which is a mechanism for sharing objects between processes of the `multiprocessing` module. These objects are accessible from any network-connected computer, firewalls permitting. An object of the `Manager` class can be given stewardship over objects of the `Queue` class. This class implements a multi-producer, multi-consumer First In First Out (FIFO) queue, to which multiple worker processes, or minions, can connect. A simple message passing interface can be defined where a single producer of workload can enqueue workload that all minions are capable of processing. When a minion has accomplished a piece of workload, or in this case an interpolation, it enqueues the result into a results queue. The ability for a centralized service to proxy requests to an army of workers, which was the most useful mechanism in the previous architecture, is therefore preserved and implemented using primitives found in the core libraries of the language.

The primary focus of this research was providing an API that generally and arbitrarily solved the problem of interpolation to any degree of accuracy. However, other solutions to the distribution of workload that utilize message queues exist, and were investigated. ØMQ, and it's python bindings pyzmq, are a concurrency framework that permits the terse authoring of complicated networking concepts including message queueing [52, 53]. ØMQ can be used in a similar fashion to the MPI libraries, albeit ØMQ is significantly less complicated. Tatotek demonstrated that implementing a network queue using zeromq instead of the multiprocessing Queue and Manager objects is approximately seven times more productive [54]. Salt is an open source project that provides a distributed execution framework built upon the ØMQ libraries [55]. Begun in February of 2011, salt is still in very active development, and therefore not currently suitable for use. The investigation of these nascent, performant, alternative libraries and frameworks may therefore be recommended for future work.

Because nearest neighbor three-dimensional queries are not yet available to open source databases, the route of implementation similar to distributed web applications was abandoned. While higher performance versions of what is included in the Python standard library exist, the built-in modules were selected due to their maturity and ubiquity.

Baker's interpolation method was selected for use in the library provided in this research. It was selected for use because it can be used with meshes of varying type and topology, or no topology at all. Also, Baker's method requires a very small stencil of points in order to perform interpolation. Lastly, it can be used to calculate arbitrarily approximate interpolations. The `KDTree` structure found in `scipy.spatial` can be used to perform rapid nearest-neighbor queries in two- and three-dimensions, and is used in the results of this research. The methods employed in the authoring of the `smbinterp` API are described in the following chapter.

CHAPTER 3. METHOD

The numerical method implemented by the `smbinterp` API yields an interpolation library with the characteristics described in the previous chapter. Namely, the library is general-purpose and functions with few restrictions on mesh type or topology. The interpolation library yields an interpolation that fits an approximating surface of any requested order to a relatively small stencil of points from a larger cloud of points. This library was implemented to take advantage of parallel computing environments to minimize execution time. This chapter contains an explanation of the numerical methods used in the implementation of the `smbinterp` interpolation library. First, an explanation of the numerical method employed for interpolation is presented, including novel and dynamic approaches to the implementation. Next, a description of the plugin architecture is explained, which is the mechanism that provides an abstract interface to the library and allows the library to operate on a variety of meshes. An explanation of the parallelization mechanism employed then follows.

3.1 Baker Method

The interpolation method proposed by Baker [10] exhibits the characteristics required to implement a general, arbitrarily approximate, two- and three-dimensional interpolation required of this research. Although there exist no general order (limited to quadratic interpolation) and parallel implementations of Baker's numerical method prior to `smbinterp`, an understanding of this method would provide the reader with a basis so as to better understand the benefits and limitations of the resultant `smbinterp` API. The Baker method is thus explained in detail first.

Let V be defined as a set of n randomly distributed points $V = \{P_0, P_1, \ldots, P_n\}$. Let N represent the dimensionality of the space, such that each point has N vector components, (x,y) in two-dimensions, and (x,y,z) in three-dimensions. Let q represent a physical quantity of interest, e.g. temperature. Each point P_i in V has a value q_i for each n locations. Denoting each *location* in V

with ξ_i, define $q(\xi_i) = \{q_0, q_1, \ldots, q_n\}$. Let Ξ be the location of the point to which an interpolation is required. In Baker's original derivation, the terms x and X are used to denote the spatial locations in place of ξ and Ξ; since x and X specifically refer to one of the orthogonal spatial directions, it is changed to ξ and Ξ herein for clarity.

Baker's interpolation method comprises the adjustment of a linear interpolation by a least squares estimate of the error of that interpolation calculated using a linear extrapolation to a stencil of surrounding points. Therefore, the Baker interpolation of the value of q to the point Ξ is defined by:

$$q(\Xi) = q_{linear}(\Xi) + f(\Xi), \tag{3.1}$$

where q_{linear} is the linear interpolation, and $f(\Xi)$ is an estimation of the higher-order error terms. The process of calculating the Baker interpolant therefore involves the calculation of $q_{linear}(\Xi)$ and $f(\Xi)$; the definition and derivation of these terms follows.

3.1.1 Linear Interpolant

There exists a clever and convenient way to linearly interpolate from the set of points in a simplex to a position within the convex hull of that simplex. This section treats the calculation of $q_{linear}(\Xi)$ from equation 3.1. A simplex is defined as the "generalization of the notion of a triangle or tetrahedron to arbitrary dimension" [56]. The term *simplex* has a more rigorous and broadly applicable mathematical definition [57], but the simple definition of a simplex is used for the purposes of this research: namely a triangle in two-dimensions ($N = 2$), and a tetrahedron in three-dimensions ($N = 3$). Note that a simplex always has $N+1$ vertexes. Therefore let the simplex defined by $R_j, 1 \leq j \leq N+1$, or $\triangle R$, denote the simplex composed of $N+1$ points surrounding Ξ, labeled R_1, R_2, R_3. In order for the interpolation to succeed, it is required that the simplex $\triangle R$ contain the point Ξ.

The other participating geometric entities are shown in figure 3.1. The red point Ξ is the point to which an interpolation is required. R and S are points in the cloud V in the neighborhood of Ξ. The blue points R_j denote the vertexes in the simplex $\triangle R_1 R_2 R_3$, and the green points S_k are extra points in the neighborhood of Ξ, not already in R_j. The three triangles formed by joining Ξ with each vertex in $\triangle R$ are denoted A_1,A_2, and A_3, and $A_{total} = A_1 + A_2 + A_3$.

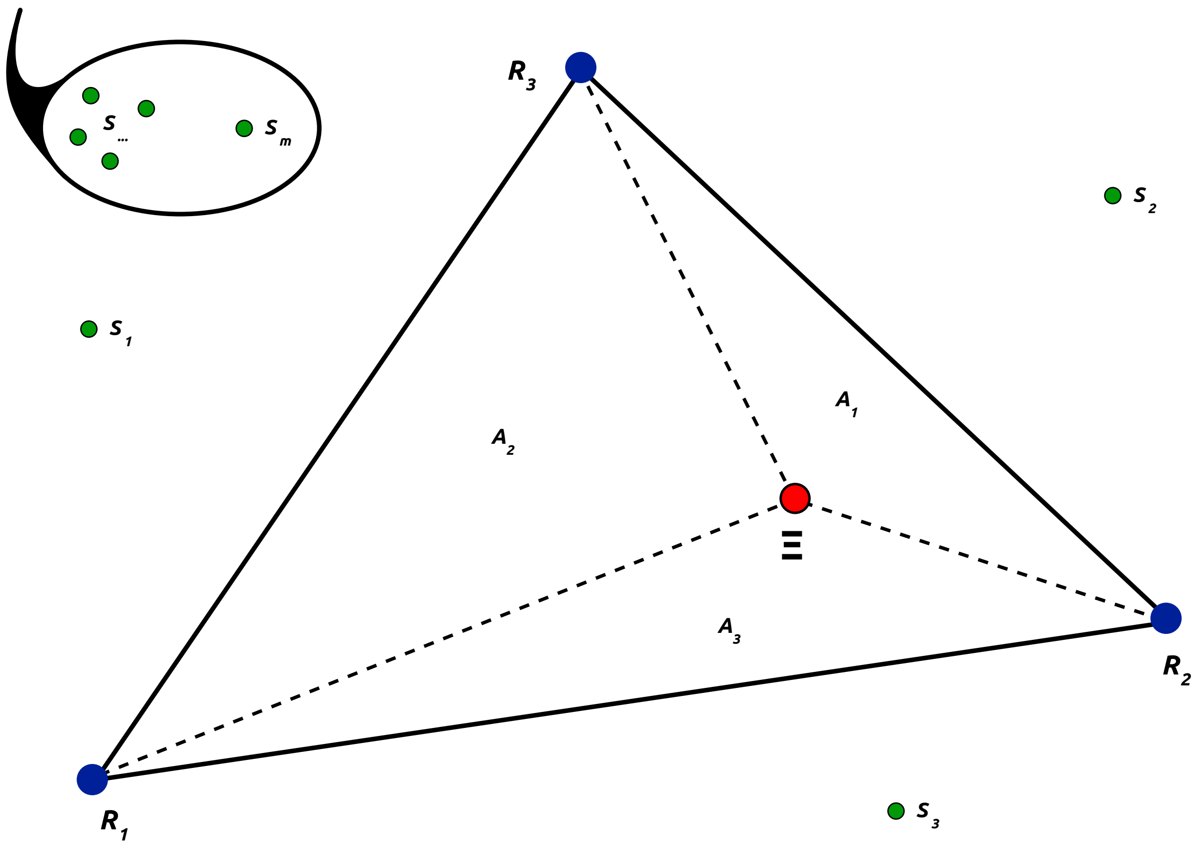


Figure 3.1: A Planar Simplex as Required for Baker's Interpolation Scheme

Barycentric coordinates are used to perform the linear interpolation; these coordinate are also referred to as areal coordinates because they relate to the area of the triangles $A_1, \ldots, A_{N+1}$ of a simplex [58]. In geometric terms, the barycentric coordinates $\phi_j(\xi)$ of a point in a simplex are the values of the normalized areas A_j/A_{total}, opposite the vertex R_j in the simplex $\triangle R$. One way to conceptualize the meaning of the barycentric coordinates is to visualize how the geometric areas A_1,A_2, and A_3 change as Ξ changes within R_j. If Ξ is selected such that $\Xi = R_1$, the ratios of the areas $A_2/A_{total} = A_3/A_{total} = 0$, and $A_1/A_{total} = 1$.

The barycentric coordinates define the influence of each point's value on the linear interpolation. In other words, the ratio of A_j/A_{total} represents the influence from $0 \leq \phi \leq 1$ that $q(R_j)$ has over the linear interpolant in the simplex. If $\Xi = R_j$, the value of $q_{linear}(\Xi)$ should then be influenced entirely by the known value of $q(R_j)$. If Ξ is placed in such a way as to give

$$\frac{A_1}{A_{total}} = \frac{A_2}{A_{total}} = \frac{A_3}{A_{total}} \tag{3.2}$$

the value $q(R_j)$ at each point R_j contributes equally to the calculated value of $q_{linear}(\Xi)$. The linear interpolant, which takes for inputs the simplex $\triangle R$ and Ξ, is defined as

$$q_{linear}(\triangle R, \Xi) = \sum_{j=1}^{N+1} q(R_j) \frac{A_j}{A_{total}}. \tag{3.3}$$

The linear basis functions for this scheme, $\phi_j(\Xi)$, must be defined in such a way as to have a value of $\phi_j(\Xi) = 1$ when being evaluated at the simplex vertex $R_j = \Xi$. All other basis functions at all other points in $\triangle R$ should yield $\phi(R_\iota) = 0, \iota \neq j$. Thus, defining the basis functions to be the barycentric coordinates, the following is obtained:

$$\phi_j(\Xi) = \frac{A_j}{A_{total}}. \tag{3.4}$$

Replacing terms in equation 3.3 using equation 3.4, the following expression is obtained for q_{linear}:

$$q_{linear}(\triangle R, \Xi) = \sum_{j=1}^{N+1} q(R_j) \phi_j(\Xi). \tag{3.5}$$

Recall that $q(R_j)$ is given a priori, and is the known physical quantity of interest at the simplex vertexes. Beginning with two-dimensional data, or planar space with $N = 2$, and expanding equation 3.5 for each R_j in the simplex for this dimension yields:

$$q_{linear}(\triangle R, \Xi) = q(R_1)\phi_1(\Xi_x, \Xi_y) + q(R_2)\phi_2(\Xi_x, \Xi_y) + q(R_3)\phi_3(\Xi_x, \Xi_y). \tag{3.6}$$

According to equation 3.6, if the point Ξ is at R_1, $\phi(\Xi_1) = 1, \phi(\Xi_2) = \phi(\Xi_3) = 0$, and the value of q_{linear} is determined solely from the value q_1, with no influence from the other R_j in the simplex. This equation exhibits the desired interpolatory behavior.

The solution for the value of the basis functions $\phi_j(\Xi)$ in equation 3.5 is the only unknown quantity required to calculate the linear interpolation. To solve for $\phi_j(\Xi)$ a system of linear equations will be defined involving the points in the simplex R_j, Ξ, and equation 3.6. If $q(\Xi)$ is a constant, $q_1 = q_2 = q_3 = q_{linear} = q_{constant}$, and equation 3.6 can be modified by dividing by $q_{constant}$, that is:

$$\phi_1 + \phi_2 + \phi_3 = 1. \tag{3.7}$$

Furthermore, the basis functions must be calculated so that equation 3.6 also interpolates geometric location of the point Ξ, hence

$$R_{1x}\phi_1(\Xi)+R_{2x}\phi_2(\Xi)+R_{3x}\phi_3(\Xi)=\Xi_x \tag{3.8}$$

$$R_{1y}\phi_1(\Xi)+R_{2y}\phi_2(\Xi)+R_{3y}\phi_3(\Xi)=\Xi_y. \tag{3.9}$$

The values of the basis functions $\phi_j(\Xi)$ can be found by solving the following system of linear equations involving equations 3.7, 3.8, and 3.9:

$$\begin{bmatrix} 1 & 1 & 1 \\ R_{1x} & R_{2x} & R_{3x} \\ R_{1y} & R_{2y} & R_{3y} \end{bmatrix} \begin{bmatrix} \phi_1(\Xi) \\ \phi_2(\Xi) \\ \phi_3(\Xi) \end{bmatrix} = \begin{bmatrix} 1 \\ \Xi_x \\ \Xi_y \end{bmatrix}. \tag{3.10}$$

Extension of equation 3.5 to $N=3$ is straight forward. The three-dimensional case requires the use of a four point simplex (tetrahedron). Adding a fourth basis function expression associated with the new point to equation 3.6 yields

$$\begin{aligned} q_{linear}(\triangle R,\Xi) =& q(R_1)\phi_1(\Xi_x,\Xi_y,\Xi_z)+q(R_2)\phi_2(\Xi_x,\Xi_y,\Xi_z) \\ &+q(R_3)\phi_3(\Xi_x,\Xi_y,\Xi_z)+q(R_4)\phi_4(\Xi_x,\Xi_y,\Xi_z). \end{aligned} \tag{3.11}$$

Using similar logic as was used for the derivation of equations 3.7, 3.8, and 3.9, $\phi(\Xi_j)$ can be calculated from

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ R_{1x} & R_{2x} & R_{3x} & R_{4x} \\ R_{1y} & R_{2y} & R_{3y} & R_{4y} \\ R_{1z} & R_{2z} & R_{3z} & R_{4z} \end{bmatrix} \begin{bmatrix} \phi_1(\Xi) \\ \phi_2(\Xi) \\ \phi_3(\Xi) \\ \phi_4(\Xi) \end{bmatrix} = \begin{bmatrix} 1 \\ \Xi_x \\ \Xi_y \\ \Xi_z \end{bmatrix}. \tag{3.12}$$

The process of calculating $q_{linear}(\Xi)$ is fairly straight forward. First, populate the matrix in equation 3.10 for two-dimensional data or the matrix 3.12 for three-dimensional data. Then solve the system of linear equations for ϕ, and then evaluate the dimension-appropriate equation for $q_{linear}(\Xi)$, i.e., either equation 3.6 or 3.11. The complete source code listing of the implementation of this method in the `smbinterp` module can be found in appendix A.

3.1.2 Least Squares Approximation of Error Terms

Continuing with the explanation and derivation of Baker's interpolation method [10], the least squares approximation of error terms $f(\Xi)$ of equation 3.1 for two-dimensional data is now derived; the case where N is 3 is explained in section 3.2. If $q(\Xi)$ is evaluated at any of the points R_j in the simplex, then $q(R_j)$ is exact, and there is no need for an error adjustment at R_j, hence $f(\Xi) = 0$. Similarly, if $q(\Xi)$ is being evaluated along any of the opposite edges to R_ι of the simplex $\triangle R$, the error term should have no influence from $\phi_\iota(\Xi)$, as $A_\iota = 0$. This condition is satisfied when expressing the error terms using the linear basis functions as

$$f(\Xi) = a\phi_1(\Xi)\phi_2(\Xi) + b\phi_2(\Xi)\phi_3(\Xi) + c\phi_3(\Xi)\phi_1(\Xi). \tag{3.13}$$

In equation 3.13 the three double products of basis functions are the set of distinct products of basis functions that are quadratic in the two spatial dimensions x and y, and zero when evaluated at each of the vertexes in $\triangle R$. This term represents a third-order accurate approximation for the error up to and including the quadratic terms. This equation introduces three unknowns whose values must be solved, namely a, b, and c.

Recall that $S_k, k = 1, 2, \ldots, m$ is the set of m points surrounding Ξ that are not in the simplex R_j. A least squares system of equations must be defined using the values of the basis functions at these points, the values of a linear extrapolation for each of those points using the simplex $\triangle R$, and the values of a, b, and c in equation 3.13. Define A as $(a, b, c)^T$. Applying least squares theory [59] a, b, and c are found by inverting the following 3×3 matrix:

$$B^T BA = B^T w. \tag{3.14}$$

The matrix B is defined using the identical basis function pattern as in equation 3.13. Denote $\phi_j(S_k)$ as the value of ϕ_j evaluated using equation 3.6 using the data point S_k instead of Ξ.

The matrix B in equation 3.14 is thus defined:

$$B = \begin{bmatrix} \phi_1(S_1)\phi_2(S_1) & \phi_2(S_1)\phi_3(S_1) & \phi_1(S_1)\phi_3(S_1) \\ \phi_1(S_2)\phi_2(S_2) & \phi_2(S_2)\phi_3(S_2) & \phi_1(S_2)\phi_3(S_2) \\ \vdots & \vdots & \vdots \\ \phi_1(S_m)\phi_2(S_m) & \phi_2(S_m)\phi_3(S_m) & \phi_1(S_m)\phi_3(S_m) \end{bmatrix}. \tag{3.15}$$

The value of $q(S_k)$ is known. The value of $q_{linear}(S_k)$ (the linear extrapolant) can also be calculated using equation 3.5. Define w in equation 3.14 as

$$w = \begin{bmatrix} q(S_1) - q_{linear}(\triangle R, S_1) \\ q(S_2) - q_{linear}(\triangle R, S_2) \\ \vdots \\ q(S_m) - q_{linear}(\triangle R, S_m) \end{bmatrix}. \tag{3.16}$$

Equation 3.14 is populated with the information from each of the surrounding points in S_k, then the unknown A can be calculated. Knowing A, equation 3.13 is evaluated for $f(\Xi)$. Subsequently the previously calculated value of $q_{linear}(\Xi)$ and the recently calculated value of $f(\Xi)$ are used to solve equation 3.1 for $q(\Xi)$. A full listing of the implementation of Baker's method in the `smbinterp` Python module can be found in appendix A.

3.2 Basis Function Pattern

The existing literature does not describe how the combination of basis functions in equations 3.13 and 3.15 generalizes to higher orders of approximations, or to higher spatial dimensions, nor does it provide a mechanism by which these terms can be calculated [10]. The implementation and explanation of this method constitutes a contribution of this research.

A pattern exists to define any error approximation function $f(\Xi)$ and covariance matrix $B^T B$ parametrized by order of approximation and dimension. Define ν as the desired order of interpolation. As defined above, N is the spatial dimension. The pattern for the combinations of basis functions that are used to define $f(\Xi)$ is the collection of ν-th ordered combinations of $N+1$ basis functions ϕ_j that are unique and non-duplicate, triplicate, etc. As an example, when

$\nu = 2$, $f(\Xi)$ is composed of the double combinations of ϕ products as shown in equation **??**. When expressing $f(\Xi)$ for the case of cubic approximation in two dimensions, where $\nu = 3$ and $N = 2$ dimensions, the error approximation term is defined as:

$$\begin{aligned} f(S_k) = {} & a\phi_1(S_k)\phi_1(S_k)\phi_2(S_k) + b\phi_1(S_k)\phi_2(S_k)\phi_2(S_k) \\ & + c\phi_1(S_k)\phi_1(S_k)\phi_3(S_k) + d\phi_1(S_k)\phi_3(S_k)\phi_3(S_k) \\ & + e\phi_2(S_k)\phi_3(S_k)\phi_3(S_k) + f\phi_2(S_k)\phi_2(S_k)\phi_3(S_k) \\ & + g\phi_1(S_k)\phi_2(S_k)\phi_3(S_k). \end{aligned} \tag{3.17}$$

The case when $N = 3$ is a natural extension of the planar case. For $\nu = 2$, the six double products of basis functions are the set of distinct products that are quadratic in (x, y, z). For a quadratic fit in three-dimensional space, there are $N + 1 = 4$ vertexes in the simplex and therefore four basis functions, and $f(S_k)$ is defined by combining the basis functions in quadratic, non-duplicate combinations:

$$\begin{aligned} f(S_k) = {} & a\phi_1(S_k)\phi_2(S_k) + b\phi_1(S_k)\phi_3(S_k) + c\phi_1(S_k)\phi_4(S_k) \\ & + d\phi_2(S_k)\phi_3(S_k) + e\phi_2(S_k)\phi_4(S_k) + f\phi_3(S_k)\phi_4(S_k). \end{aligned} \tag{3.18}$$

Implementation of this pattern is listed in full in appendix A. First the Cartesian product of all integers from zero to the size of the simplex is computed. Next the products that only contain a single number are removed, which enforces the requirement that the groups of ϕ be non-duplicate, non-triplicate, etc. If the current entry is not composed of a single number, it is sorted, converted into a Python tuple (an immutable sequential data structure), and appended to the list of all potential entries. This list will not be a unique set of products, but is made so by creating a Python set of the sorted entries, which enforces uniqueness of the terms in the collection.

The dynamic calculation of the basis function pattern in this fashion is powerful, in that it can be calculated for any arbitrary ν, and for planar, volumetric, or higher dimension spaces (although only N of 2 and 3 are dealt with herein). However, for each point Ξ the calculation of the pattern must be performed once for the calculation of $f(\Xi)$ and once per extra point S_k participating in the current interpolation for each row in the B matrix. There is only one valid pattern per set of inputs N and ν, which must both remain constant throughout a single interpolation. The calculation of the pattern is a computationally intensive operation, and so a caching mechanism has been implemented in the `smbinterp` API that only calculates the pattern if it has not been previously calculated. This concept is known as memoization, the implementation thereof is in appendix A, and a flowchart of the algorithm is shown in figure 3.2.

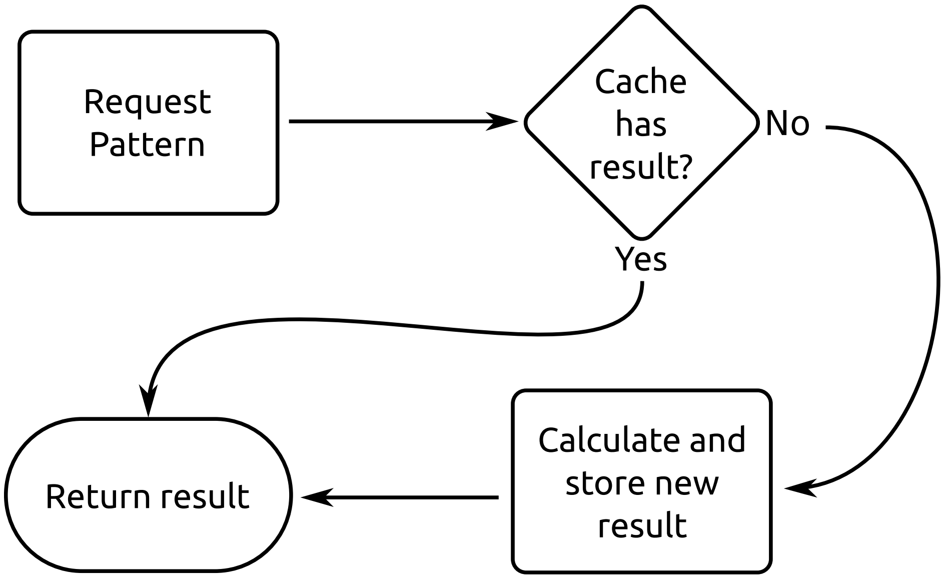


Figure 3.2: Flowchart of the Pattern Implementation

The memoizing function forms a closure over a Python dict object named cache, i.e. the same cache dict will be available and used by any future call to the memoized inner function. As such, when the function that calculates the $f(\Xi)$ pattern is called it first checks to see if the inputs N and ν have not yet been used. If the inputs have not yet been used, a pattern is calculated and returned, otherwise the value in the cache dict matching the input variables is returned.

This mechanism has a slightly higher memory footprint, but runs at a significantly lower computational expense per call, $O(1)$ with memoization, rather than $O(N)$ without. In the `smbinterp` API, the memoized pattern function is used to calculate the rows in the B matrix, and in the calculation of $f(\Xi)$, and is listed in its entirety in appendix A.

3.3 Mesh Plugins

Multiphysics simulations are simulations involving multiple, disparate physical models. The individual simulations that compose an encompassing multiphysical simulation generally employ varied mesh types and topologies to correctly solve the disparate physical models involved in the larger, overarching simulation. The `smbinterp` API developed in this research was designed and implemented to function with different types of meshes and mesh topologies so as to not limit the scope of applicability. In order to solve equations 3.5, 3.13, 3.14, and eventually 3.1, two geometric entities, and their associated q quantities are required: the simplex $\triangle R$ and the extra points S_k. The simplex $\triangle R$ that contains the point Ξ is required to calculate the linear interpolant. The simplex $\triangle R$ and a collection of S_k of surrounding vertexes are used to calculate the error approximation term. In order to be used with the interpolation library, a given mesh structure must provide the API with both $\triangle R$ and S_k.

3.3.1 $\triangle R$ and S_k Vertex Selection

While Baker's method gives a reasonable interpolation solution for a general cloud of points, it does not specifically address the question of how to select points from an existing mesh. The method suggested by Baker consists of simply selecting the nearest points. This is the most general vertex selection algorithm for the terms $\triangle R$ and S_k. It consists specifically of collecting the $(N+1)+m$ closest vertexes to the destination point Ξ, using the nearest $N+1$ in the simplex, and using the remaining points to compose S_k. This selection algorithm does not always provide the best vertexes for interpolation results as shown in figure 3.3. The images on either side of the figure represent the same source donor mesh (red points) and destination point Ξ (blue). The black triangle represents the selection for the simplex $\triangle R$. The selection on the left was calculated using nearest-neighbor vertex selection. While it may seem that selecting the geometrically closest points to Ξ would yield the same containing simplex, the vertex six rows above the destination point is closer than the vertex directly above the vertex directly to the left of Ξ. In this mesh, a connectivity-based vertex selection would yield the simplex shown in the image on the right. The mesh was designed to capture the gradient information, and therefore the mesh topology should be respected. Simply selecting the closest points to Ξ would therefore yield inferior results. By

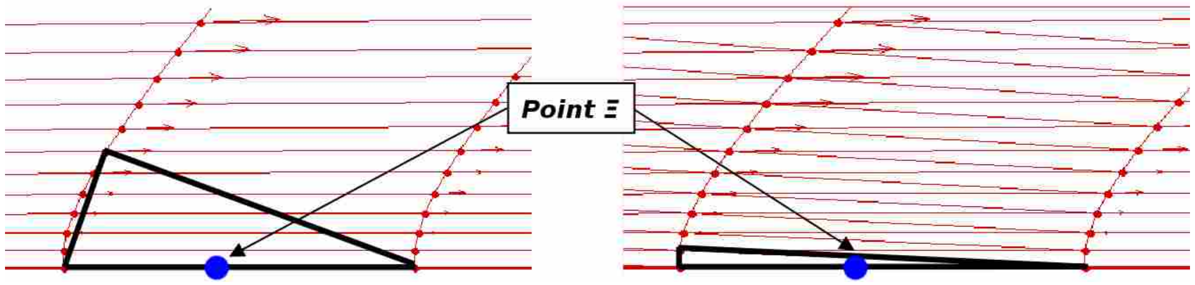


Figure 3.3: Graphical Representation of Point Selection Algorithms: Nearest-Neighbor Point Selection (left), Connectivity-Based Point Selection [9]

selecting the more topologically adjacent points the information intended to be captured in the mesh's design will be preserved.

Alternatively, consider the case with a regular grid as shown in figure 3.4. Again, the red points are vertexes from the source mesh, the blue point is the destination point Ξ, and the black triangle is the simplex $\triangle R$. While the simplex selection represents the topology of the mesh , if the selection of extra points is performed using a nearest-neighbor selection algorithm the nearest points will all be collinear with an edge of the simplex. If the number of points in S_k is small, there is a high likelihood of selecting m points such that more than $m-2$ of the extra points are on a line that is coincident with two of the points in the simplex. This condition is one of the pathological cases described by Baker [10]. If this occurs, the covariance matrix $B^T B$ will be singular, the solution will not be unique, and the error approximation will not generally aid in improving the interpolation.

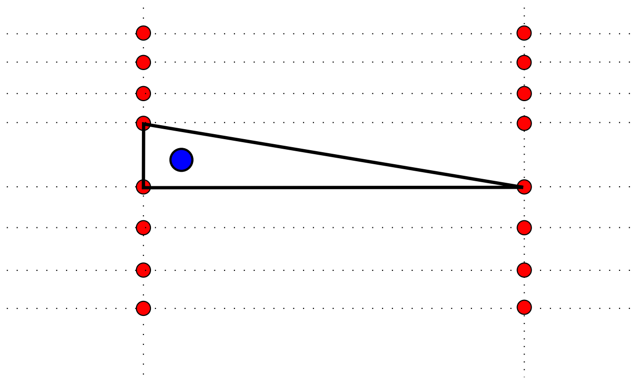


Figure 3.4: Regular Grid With Extra Points That are Collinear with an Edge of the Simplex $\triangle R$

3.3.2 Plugin System Design

Rather than designing and implementing a mesh object for each and every possible mesh type and topology, the `smbinterp` API contains a plugin system for integration with new mesh types. This abstraction layer allows users of the API to rapidly develop the code required to integrate results from their numerical analyses into the interpolation framework. By using the plugin system, the engineer can precisely control how points are selected to avoid situations where an unsophisticated vertex querying algorithm would provide inferior results. An overview of the plugin system is provided here, but a complete listing is provided in its entirety in appendix A for reference.

The base grid class for all possible mesh types, named the grid class, is defined in the `smbinterp`'s grid module. The base grid class takes all the vertexes in the mesh and the values of $q(\xi)$ at those vertexes as parameters for object instantiation. Each instantiation of this class contains a Python list whose values are the spatial locations of the known quantities q. The spatial locations in this list are used to populate a kdtree for rapid nearest-neighbor querying. Also, the grid base class contains a list containing the values of q at the locations in the verts list.

The grid base class also contains two other data structures that assist in the discovery of the simplex in an upstream mesh that contains the given vertex Ξ. The first is a key value mapping that provides a mapping mechanism between a cell name and an actual cell object. Because not all cell identifiers are generally known at object instantiation, a dict was chosen over a list for $O(N)$ insertion of new cells. The second internal structure in the base grid class is a another mapping which maps each vertex id to all cells that contain the vertex with the key id. The cell objects that populate these two structures consist of the connectivity information linking the aforementioned geometric entities: a list of all vertexes that compose that cell, as well as a list that contains references to all neighboring cells. These data structures are populated so that each cell may be quickly queried for information regarding either vertexes in that cell or information about topologically adjacent, neighboring cells. This adjacency information is generally computed before a round of interpolations, and is stored in the grid object.

Each plugin must provide the Baker's method with two pieces of information: the simplex $\triangle R$ and surrounding points not already in the simplex, S_k. To determine the simplex $\triangle R$, the base grid class, and all classes derived therefrom, use a default algorithm for locating the simplex that contains the point Ξ. In this simplex querying method the vertex nearest to Ξ is found using the kdtree structure. Next the cells that are adjacent to the closest vertex are identified using the internal mapping structures. These adjacent cells are visited in turn, and each cell is tested if it contains the vertex Ξ. Once all cells that are immediately adjacent to the nearest vertex to Ξ are visited, all adjacent cells to visited cells are recursively visited until an enclosing simplex is found, or until a configurable limit is reached. The method employed in this research used to locate the containing simplex in an upstream mesh is similar to what has been previously described by multiple authors [41,60,61]. First the spatial tree structure is used to find the location of the nearest vertex to the point of interest, then other cells are visited in topologically adjacent order. The selection of the extra points S_k is also implemented in the base grid class. This algorithm simply queries the kdtree structure for $(N+1)+m$ points and discards the points that are in the simplex $\triangle R$. This method could be overridden to provide a more complicated S_k selection algorithm.

3.3.3 Provided Plugins

Two plugins are provided by this research. One plugin for meshes calculated using Gmsh (an open source meshing utility) is provided. Another plugin takes a cloud of points, and calculates connectivity information based on a Delaunay triangulation of those points. Delaunay triangulation comprises an algorithm to connect points into simplexes such that no vertex of any element in the set of formed simplexes resides in the circumcircle of any of the other simplexes in the set [62]. The complete listing of these plugins is provided in appendix A for reference. The gmsh plugin demonstrates the parsing of a file on disk, whereas the Delaunay plugin shows that the information does not necessarily need to reside on disk to be used by the API.

3.3.4 Benefits of the Plugin System

The plugin architecture implemented in `smbinterp` provides two distinct features related to computational flexibility. First, each plugin implements the mechanism for populating the structures used in the default simplex finding and extra-point-finding algorithms. A distinct plugin should be developed for each type of mesh (structured, unstructured, etc.), and can be implemented to enforce any form of mesh topology traversal. As an example, a plugin for a structured i, j, k mesh could be developed to verify that any additional points being added to S_k are not collinear with any of the lines in the simplex $\triangle R$ already selected for a point Ξ. A plugin of this variety would prevent the pathological situation described above.

The second benefit comes from the ability to override functions in the base class. If an alternative simplex location or extra point location algorithm is desired, the plugin can simply override the base class methods that implement these algorithms. The ability to override and enhance the default behavior of the default grid object as implemented by `smbinterp` is what enables this flexibility. It allow the end users of the API to use the interpolation engine by either using an existing plugin, or by developing a relatively small piece of code that simply presents the data to the API according to the interface as defined in the appendix A. This is what enables the API to be used with practically any mesh type or topology.

3.4 Parallel Execution Framework

The calculation of $q(\Xi)$ generally involves the solution of two non-trivial linear systems of equations per point Ξ. This computational expense is exacerbated by querying spatial data structures multiple times for the discovery of appropriate $\triangle R$ and S_k terms. Depending on the number of vertexes in the downstream mesh, the calculation of all vertexes could take a substantial amount of time. The problem is therefore bound primarily by the ability to perform the spatial queries and calculations more than any other factor. The `smbinterp` API provides a mechanism by which interpolation workload may be distributed amongst a set of participating parallel computers.

Due to the abundance of memory on modern HPC nodes, large grid objects can be persistently loaded in their entirely in ram, multiple times on multiple nodes, against which interpolations can be performed in parallel by compute units. The compute units, named *minions* in `smbinterp`, can perform multiple parallel queries against a persistent data set; the interpolation of many spatial locations therefore lends itself to being solved in a parallel fashion. It was found that the performance issues could be addressed in a scalable fashion by composing an embarrassingly parallel solution to the problem using libraries that are built into the Python language. Embarrassingly parallel workloads are those “for which little or no effort is required to separate the problem into a number of parallel tasks” [44].

The crux of the solution lies in providing the minions with a steady stream of work, and a pipeline for the reporting of the resultant interpolations. If the minions could subscribe to a collection of work to be done, and conversely report completed workload back to a centralized location, the problem would be solved. The `Queue` module in the Python standard library provides a `Queue` class which implements a multi-producer, multi-consumer First-In-First-Out (FIFO) queue, to which multiple simultaneous minions on a single machine can connect. Furthermore, the `multiprocessing` module provides a `manager` class, which enables the sharing of `Queue` objects over a distributed network. A reference implementation of the parallelization scheme implemented in this research is shown in the flowchart in figure 3.5. What follows is a brief explanation of the constituent parts; the source in its entirety is provided in appendix A.

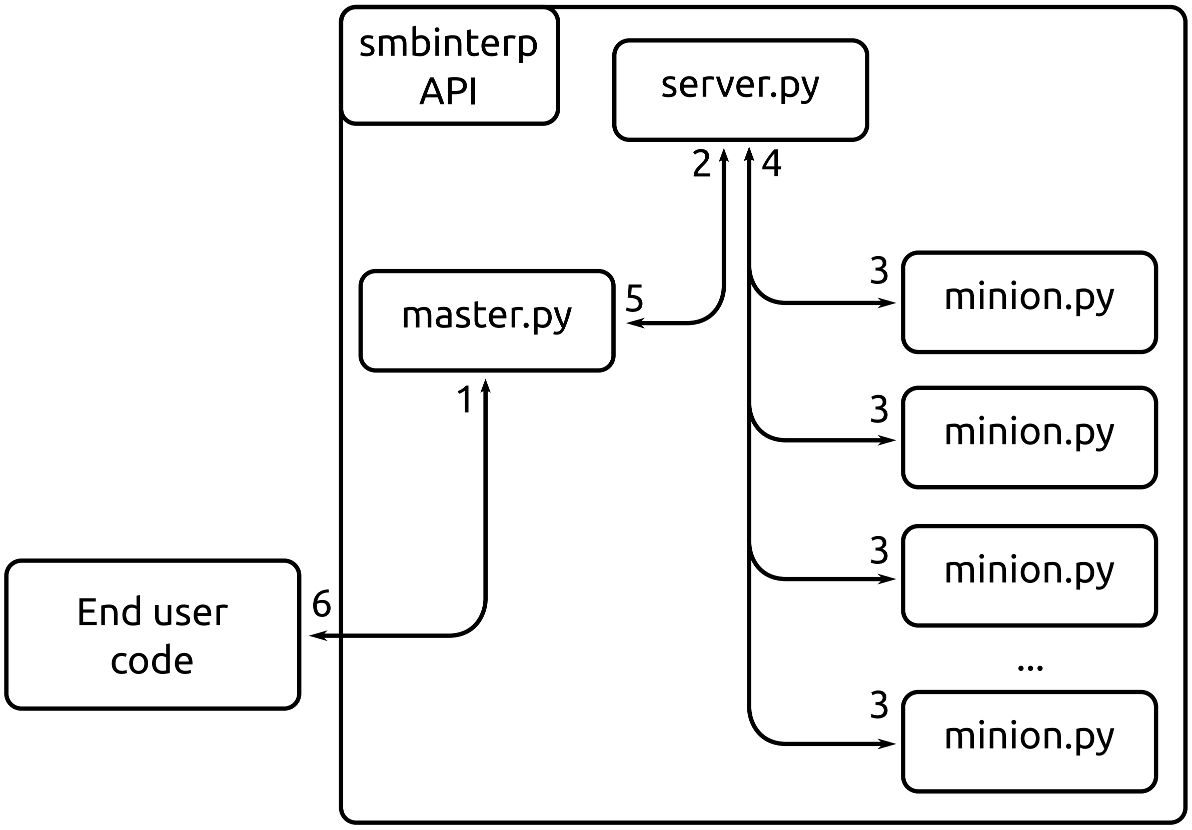


Figure 3.5: Flowchart of the Parallelization Architecture

The task of parallel interpolation begins with code, authored by the end user, submitting interpolations to be performed. This can be done in two steps as shown in the flowchart, or in one step by the `master.py` script, as demonstrated in appendix A. The `master.py` script is responsible for orchestrating the submission of interpolations and events associated with starting and stopping a set of interpolations. Once all of the interpolations have been submitted into the `server.py` script, the `master.py` script then signifies to all participating minions that all workload is ready for consumption. Each of the multiple `minion.py` scripts begins requesting workload from the `server.py` script and calculating interpolations in rapid succession. When the interpolations are calculated, they are fed back to the `server.py` script, which then returns all collected interpolations ether to the `master.py` script or to end user code. The `minion.py` scripts remain active and ready to perform interpolations against a set of data until the master script sends a termination message, at which point their service is finished, and resources are relinquished to the system.

The Python code required to share multiple `Queue` classes over a network as required to implement the `server.py` script is demonstrated in appendix A. In this code four queues are made available to other scripts on the same network: a queue for tasks to be performed, a queue for results, and two queue objects for orchestrating the control of a round of interpolations between a master and a set of minions. Masters and Minions authenticate and connect to these four queues to accomplish the tasks shown in the flowchart in figure 3.5.

The `smbinterp` API was designed to implement and enhance the numerical method proposed by Baker. It does so by providing the first library that allows for arbitrary and dynamic accuracy specification. Also, it was designed to allow for end users provide alternative point selection algorithms via its plugin architecture. Lastly, the interpolation API is implemented in a parallel fashion. Results of the implementation of the `smbinterp` API a presented in the following chapter.

CHAPTER 4. RESULTS AND DISCUSSION OF RESULTS

The `smbinterp` library was written to meet the needs stated in previous chapters. Namely, the library should provide a mesh-agnostic interpolation library that can calculate interpolations to an arbitrary degree of accuracy, and can be run in parallel. The library designed and implemented in this work provides a generic, nth-order approximate interpolation that can take advantage of parallel computing environments. This chapter contains the results of various benchmarks and tests of the `smbinterp` library intended to show that these research goals were met. Section 4.1 of this chapter describes a study performed with the library that parametrically tests various configurations and mesh resolutions for accuracy and temporal performance. In general the library took longer to calculate more accurate results due to the use of higher-order settings. The second section, section 4.2, presents the results of a study on the efficacy of the parallelization algorithm implemented in the `smbinterp` library, which was found to scale quasi-linearly to 180 participating minion processes.

4.1 General Library Performance

A study of the effects of changing the parameters of the `smbinterp` library for a given mesh on temporal performance and accuracy is presented here. Both the temporal performance and the accuracy of the library are primarily dependent upon two factors, which are both parametric inputs to the library. The order of the error approximation term is the first configurable input of `smbinterp`. While the library provides quadratic error approximation by default, specifying an alternative order is optional [1] The size of the stencil of extra points (S_j) used to calculate the error approximation term is the second input; it is equally configurable. These two parameters affect both the accuracy and the duration of each interpolation.

[1]This interpolation library goes to eleven.

The exact equation used to benchmark the results of the `smbinterp` library needed to be chosen carefully. When interpolating a function that is constant or linear in the spatial dimensions linear interpolation always provides an exact solution. A function that varied smoothly in the domain was therefore required to calculate the value of $q(\xi_i)$ for all known points in the source mesh in order to obtain meaningful results. Equations 4.1 and 4.2, which are slightly modified versions of the equations used to validate the original numerical method [10], were used to validate the performance of the library in two- and three-dimensions, respectively. A plot of the two-dimensional equation is shown in figure 4.1.

$$q(x,y) = (\sin(x\pi)\cos(y\pi))^2 \tag{4.1}$$

$$q(x,y,z) = \left(\sin\left(\frac{x\pi}{2}\right)\sin\left(\frac{y\pi}{2}\right)\sin\left(\frac{z\pi}{2}\right)\right)^2. \tag{4.2}$$

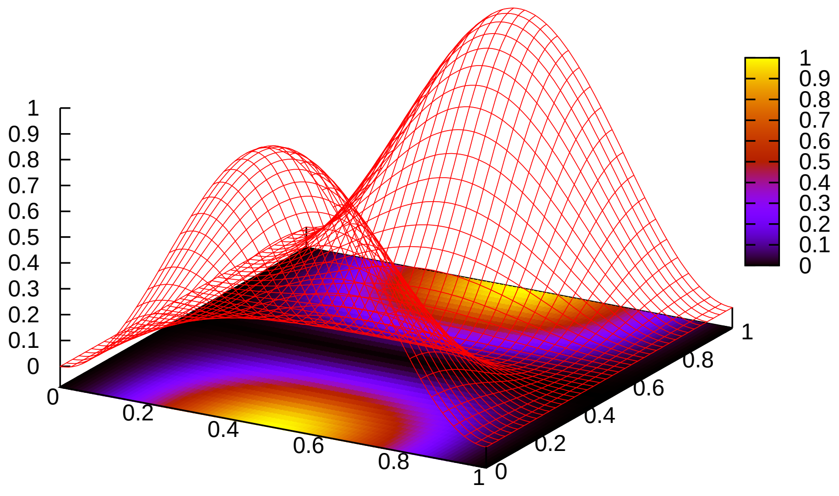


Figure 4.1: Plot of Equation 4.1

Equations 4.1 and 4.2 were also used to calculate the exact value of q at each point Ξ when calculating the error of each interpolation. Mesh resolution was varied by creating meshes of varying cell density using the open source tetrahedral meshing program `gmsh` [11]; the quantity of vertexes and elements of each of the meshes is shown in table 4.1, and an example of the two coarser two-dimensional meshes are show in in figure 4.2. The order of error approximation (ν) was varied from quadratic (2) to quintic (5), inclusive. The size of the stencil of extra points (S_j) was varied in an approximately exponential fashion as follows: 4, 6, 8, 12, 16, 20, 32, 48, 64, 96, 128, 192, and 256. A collection of 1000 random points within the domain (0 to 1 in each orthogonal dimension) of each mesh was generated for the study, and the same set of generated points was used during each parametric permutation. This collection of points represents points that would be unknown in a destination mesh during a multiphysics simulation.

Table 4.1: The Mesh Vertex and Element Counts Used in the Parametric Library Study

Mesh ID	Vertexes	Elements
2D		
1	529	1060
2	5083	10168
3	50354	100710
4	534608	1069218
3D		
5	1915	10635
6	170297	1008870
7	1701554	10188506

4.1.1 Interpolation Improvement

The root mean square (RMS) of the errors (ε_i) was used to determine the accuracy of the `smbinterp` library. Each error ε_i was calculated as the difference between the actual value (from equations 4.1 and 4.2) and calculated interpolations (at each point in the destination domain using `smbinterp`), or $\varepsilon_i(\Xi) = q_{exact}(\Xi) - q_{calculated}(\Xi)$. RMS was calculated according to equation 4.3 for all μ points in the destination mesh.

$$\varepsilon_{rms} = \sqrt{\frac{\sum_{i=1}^{\mu} \varepsilon_i^2}{\mu}} \tag{4.3}$$

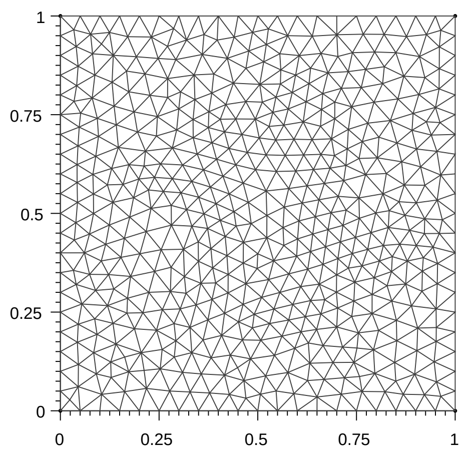
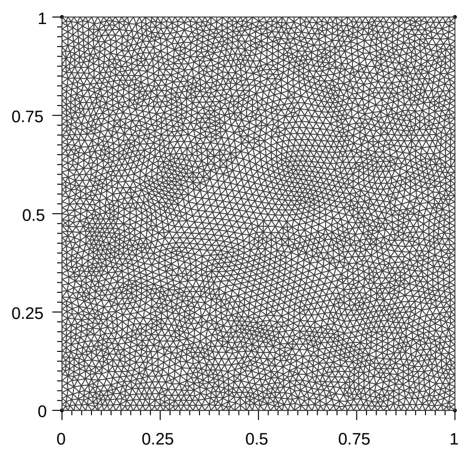


Figure 4.2: Plots of 2-D Meshes, Resolution 1 and 2

The qualitative notion of result improvement was determined by comparing the ratio of interpolations improved by using the `smbinterp` library (by comparison with a linear interpolation) to the total number of performed interpolations. A value of 1.0 indicates that every calculated interpolation was an improvement over a linear interpolation; a value of 0.5 indicates that half of the interpolations were less correct than the linear interpolation. The comparison provides a qualitative benchmark for the library against other interpolation libraries which only provide linear interpolation. A given interpolation was found to have provided a better result than linear interpolation by meeting following inequality:

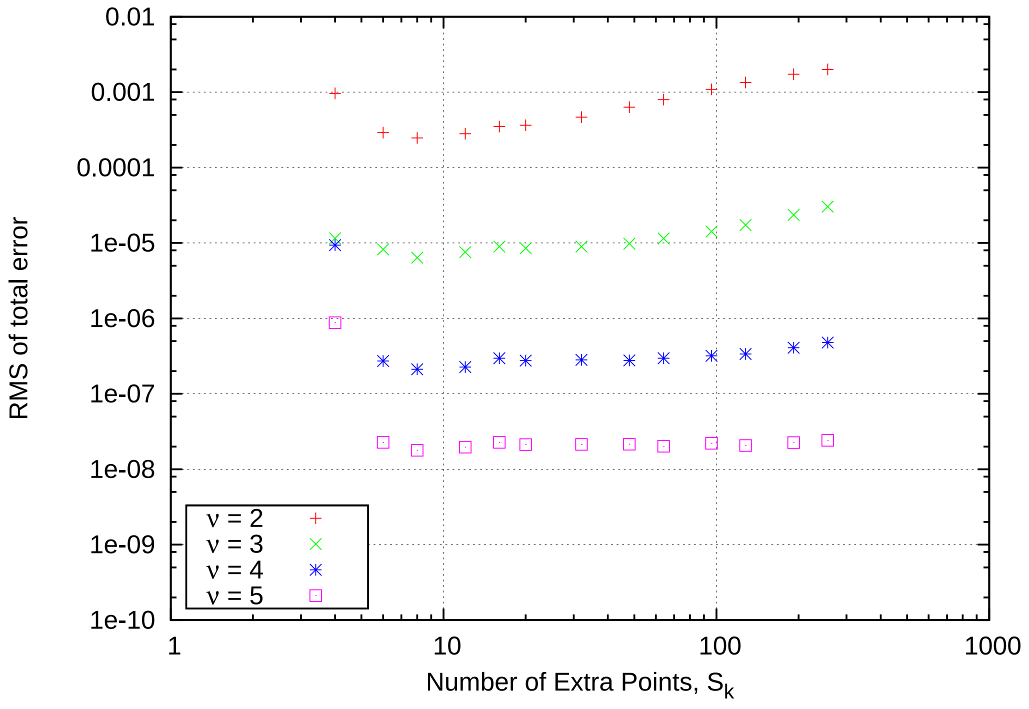
$$\left|q_{final}(\Xi) - q_{exact}(\Xi)\right| \leq \left|q_{linear}(\Xi) - q_{exact}(\Xi)\right| \tag{4.4}$$

These two quantities (RMS of errors and percent improvement) were calculated for all parameters in the study and the results for the two-dimensional meshes are shown in the plots of figures 4.3-4.6. The top plot in each of these figures shows the RMS of errors of the interpolation at all points in the destination mesh as they were found by varying order of interpolation and the number of participating points in the stencil S_k for a particular mesh density. The x-axis of these plots represents the number of extra points in an interpolation, and the y-axis is the RMS of error for all points in the destination mesh. Plots of the the improvement ratio are shown on the bottom of these figures. These plots also show the number of extra points on the x-axis, and show the fraction, from 0 to 1, of improved interpolations on the y-axis.

The plots in figure 4.3 plot the results obtained by using the most coarse mesh and by varying the parameters of the `smbinterp` library. The coarse mesh is shown on the right of figure 4.2. The RMS of error ε_{rms} drops slightly for all orders of interpolation when the number of extra points increases from 4 to 6. Thereafter, up to 256 extra points, the number of extra points has little effect on the RMS of error for the three higher degrees of interpolation shown. The results obtained using the coarse mesh show that generally the order of interpolation has more influence on the RMS of error than increasing the number of extra points. However, the RMS of error for the quadratic interpolation (*nu* of 2) continues to increase until the RMS of error with 256 extra points is higher than it was when just using 4 extra points, and approximately an order of magnitude higher than it was at the very lowest, the value obtained with 8 extra points. The effect of this increase in error can be seen in the bottom plot; the quadratic interpolation provides a worse interpolation than linear for over 30 percent of all attempted interpolations. For higher-order interpolations this effect is less substantial, and for ν of 4 and 5 yield an improvement ratio of above 0.98.

For values of S_k between 12 and 256 the plots in figure 4.4 show similar trends to the plots in figure 4.3; the RMS of error is however an order of magnitude lower. The plots in figure 4.4 were produced using the mesh on the right of figure 4.2. The most marked difference between the two plots is that the error decreases significantly between 6 and 12 extra points. After crossing this threshold increasing the number of extra points does not change the number of successful interpolations, except for the lower-order interpolations. A similar trend for the quadratic interpolation is shown: increasing the number of extra points S_k increases the number of failed interpolations to 20 percent at 256 extra points. Another more subtle difference is that the RMS of error creeps up for all four orders of interpolation after crossing the threshold between 6-12 extra points, whereas the RMS of error appeared to be more constant in figure 4.3.

The plots of the in figures 4.5 and 4.6 are very similar to those in figure 4.4, and were produced with the meshes with next more refined resolution, meshes 3 and 4. In these figures there exists a similar sudden dip in RMS of error as that seen in 4.4, the RMS of error is generally an order of magnitude lower for each sequentially more refined mesh, and increasing the number of extra points past the threshold increases the ratio of failed quadratic interpolations. The main



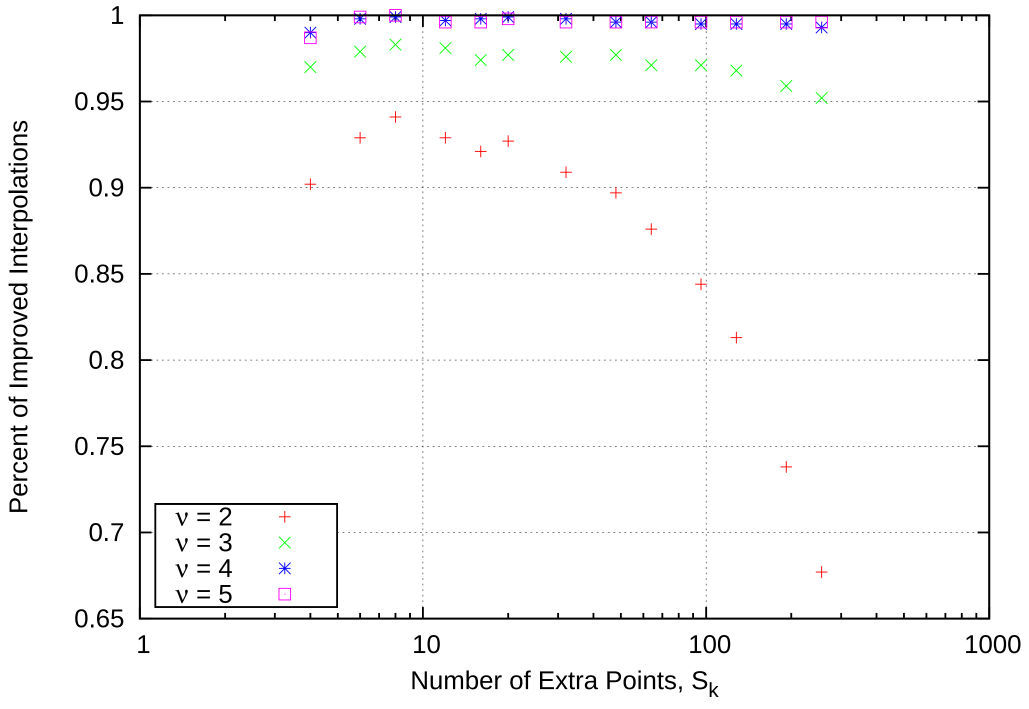


Figure 4.3: Plot of the RMS of Error and Improvement Ratio for Mesh 1

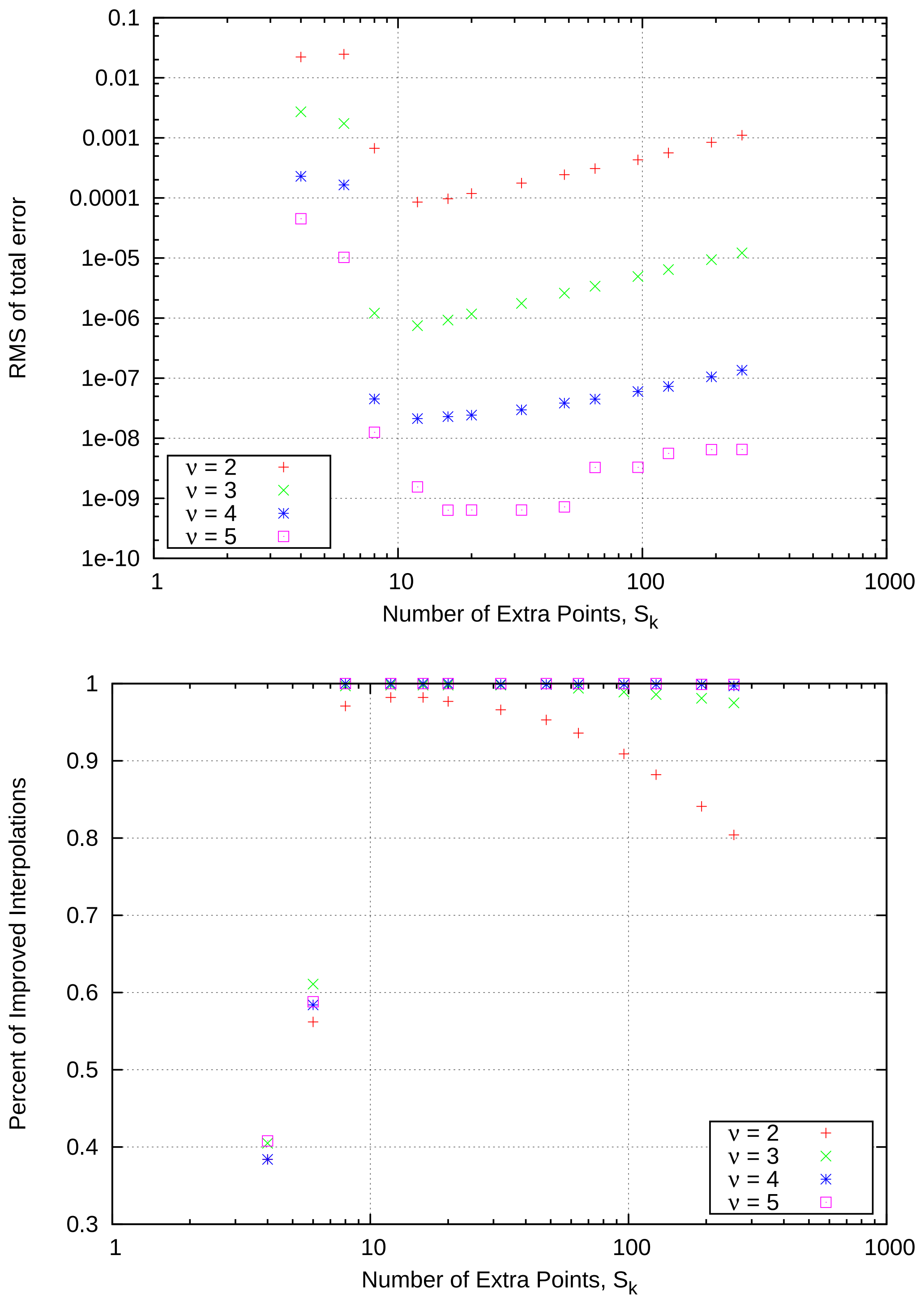
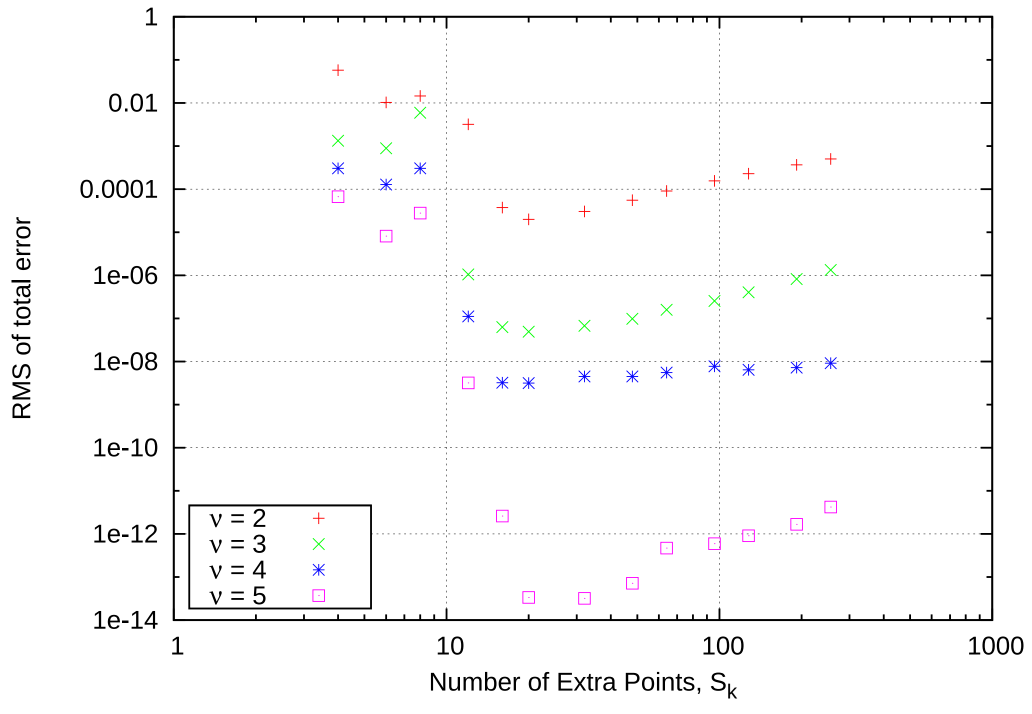


Figure 4.4: Plot of the RMS of Error and Improvement Ratio for Mesh 2



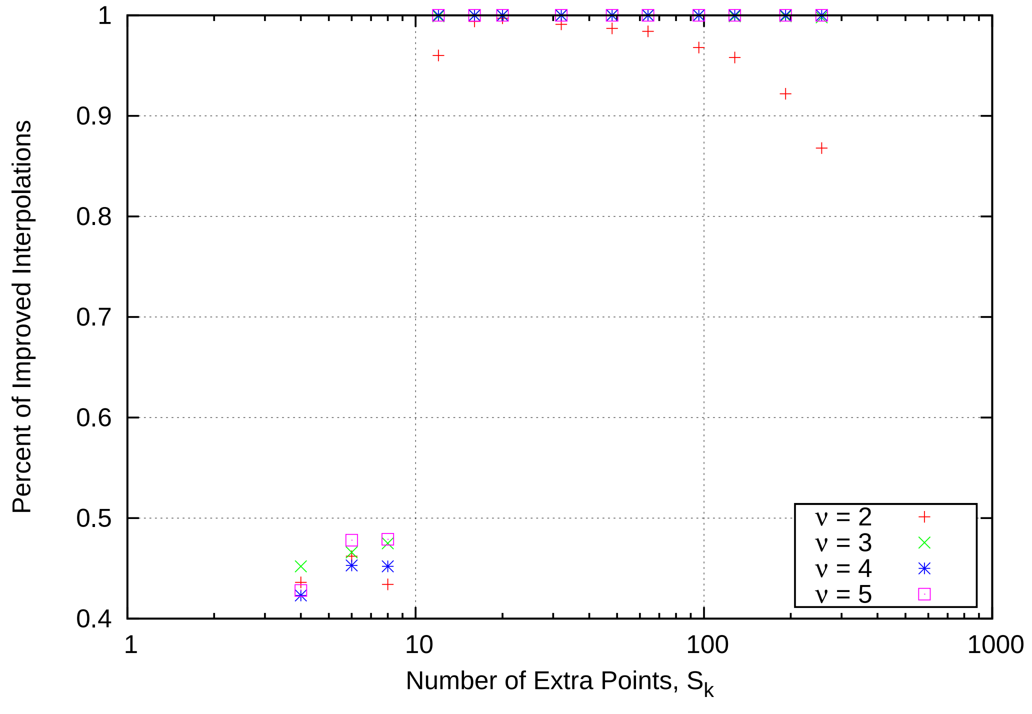
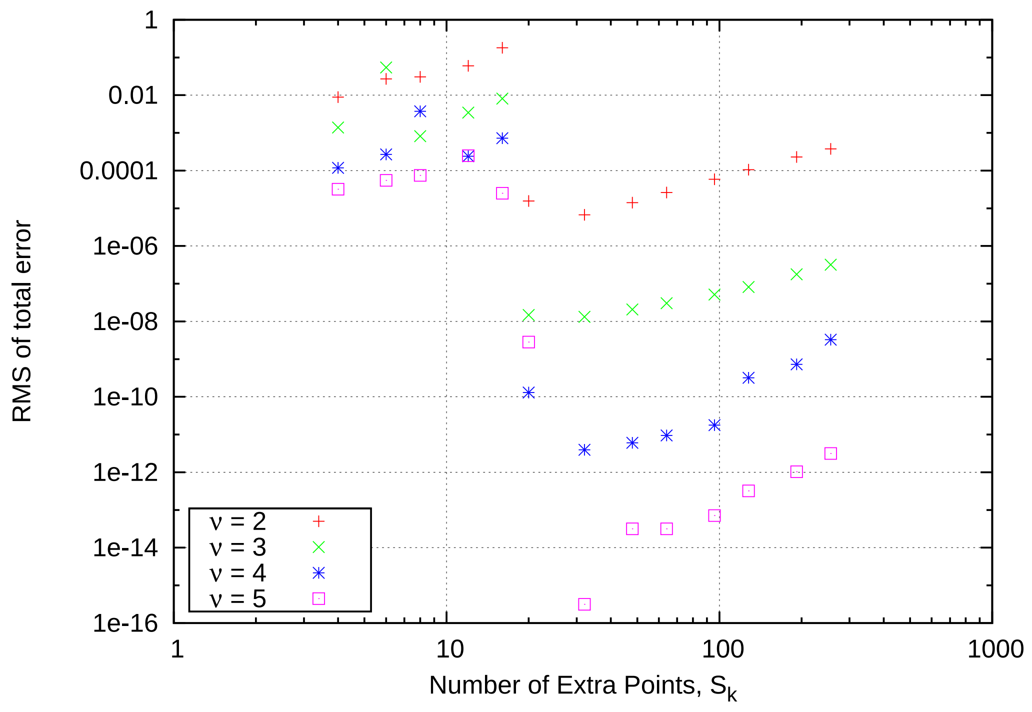


Figure 4.5: Plot of the RMS of Error and Improvement Ratio for Mesh 3



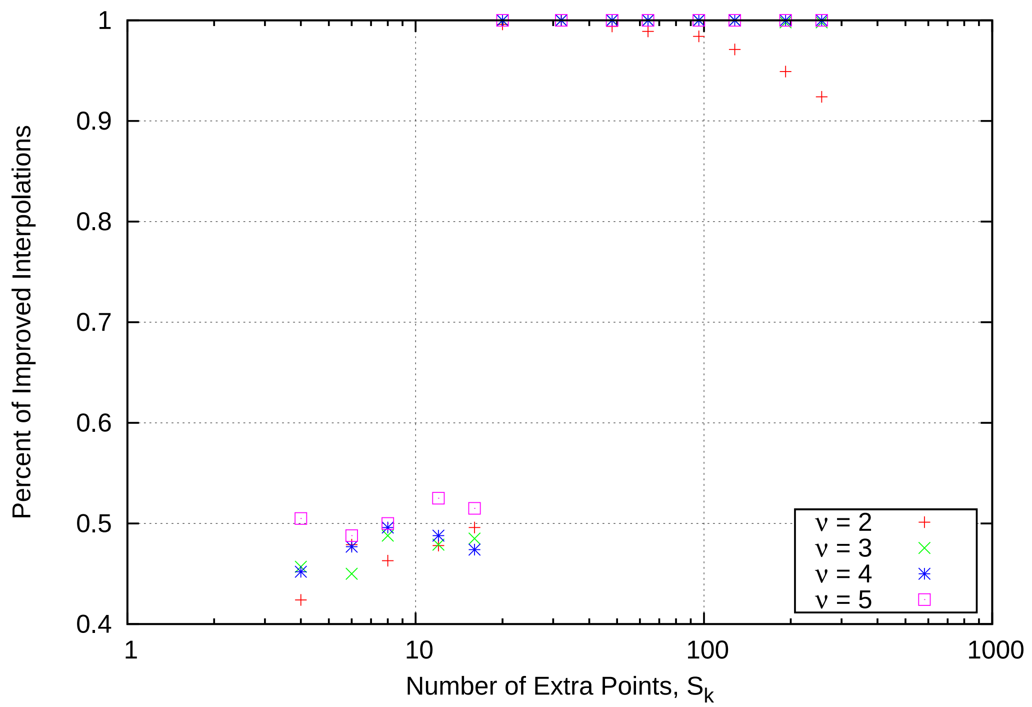


Figure 4.6: Plot of the RMS of Error and Improvement Ratio for Mesh 4

difference between the results in figures 4.5 and 4.6 and those shown in figure 4.4 is that the sudden jump in RMS of total error occurs at a higher number of extra points as the mesh resolution increases with the lowest point reached at 12, 20, and 32 extra points for each of the finer meshes, respectively.

As the order of interpolation increases the RMS of the error generally decreases. Specifically, for a given mesh density, the higher-order interpolation generally yielded more accurate results. Also, as the mesh density increases, the accuracy also increases. This increase in accuracy is what justifies the use of more computationally expensive settings such as a higher order error approximation or a more resolved mesh. As the number of vertexes in the stencil is increased, the RMS of the error generally undergoes a sharp decline, followed by a gradual increase. This can significantly impact the accuracy of the library as shown in figures 4.3-4.6. When using a very small stencil (less than 12 vertexes), a significant fraction of all interpolations (50-60 percent on all but the coarsest mesh) were worse after the adjustment by the error approximation term. However, after passing the threshold of sharp error decline the interpolations performed with the `smbinterp` library improved for almost all Ξ in the destination mesh. This generally occurs at a value of $16 \leq m \leq 32$ As the mesh density increases, this jump requires a larger stencil, with the worst case for two dimensions involving order 5 interpolation requiring 32 extra vertexes before achieving minimum RMS error. It is noteworthy that this sudden jump in RMS of error occurs at the same size of stencil for the three finer mesh cases. Also, interpolations performed after crossing this threshold (using more than 32 extra points) on the finest mesh provided the lowest RMS of error.

An increase in the quantity of extra vertexes for low-order approximations yielded slightly counterintuitive results with respect to RMS of total error and improvement ratio. The quadratic error approximation also exhibits the sudden decrease of RMS of error (at 10-12 extra vertexes), however the RMS of error increases when increasing the size of the stencil past this point. This effect is particularly noticeable in bottom plot of figure 4.3 which shows the fraction of all interpolations that improved using the most coarse two-dimensional mesh. Using any more than 8 extra vertexes has a harmful effect on the RMS of error during interpolation; in the case of the coarse mesh over 30 percent of the interpolations calculated using the error approximation term (calculated using 256 extra vertexes) provided worse results than a linear interpolation alone.

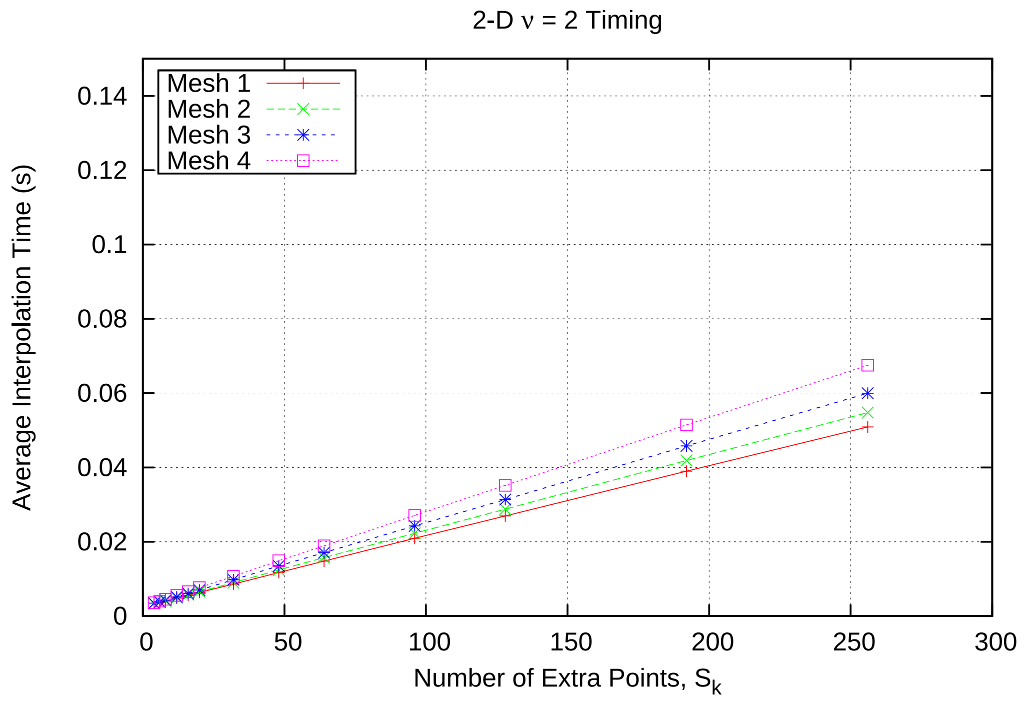
This phenomenon is evidence of an artificial smoothing effect caused by using a low-order approximation of error terms for an underlying equation of a higher order. In this case it is recommended to use a higher order interpolation or a finer mesh; even for the coarse mesh the fourth- and fifth-order interpolations provided very good interpolations, with over 98 percent of all interpolations improved by using `smbinterp` by comparison with a linear interpolation alone for stencil sizes larger than 6.

As with all discrete numerical methods, care should be taken to match the order of interpolation to the order of the underlying mathematical equations. While the use of the `smbinterp` library can provide the user with excellent interpolation results, it is not a silver bullet intended to perform regardless of the underlying equations, or meshes used to present the requisite information to the library.

As an example, the quadratic interpolation in figure 4.3 shows that if too many extra points are used with a quadratic interpolation, the interpolation has a smoothing effect that makes up to 30 percent of the interpolations worse for having used the library. Also, the mesh resolution should resolve pertinent gradient information. While this is common of well designed meshes, the library performs better when this is guaranteed. Furthermore, if information is known a priori about the data, which could be used during the interpolation algorithm, the known information could be used to intelligently select the order of interpolation and the number extra points used to calculate the error estimation term. Investigation of the active adjustment of the interpolation parameters might be suggested for future work.

4.1.2 Temporal Performance

Figures 4.7 and 4.8 show the results of the temporal performance study. The x-axis in these figures represents the number of extra points used in calculating the error approximation term. The y-axis shows the average time spent per interpolation. The order of approximation and the mesh resolution was varied as explained in the previous section, and results for all permutations for a particular spatial dimension are shown on a single plot. The plots in figure 4.7 shows results for the timing of the two-dimensional test cases, and the plots in figure 4.8 show results for the three-dimensional test cases.



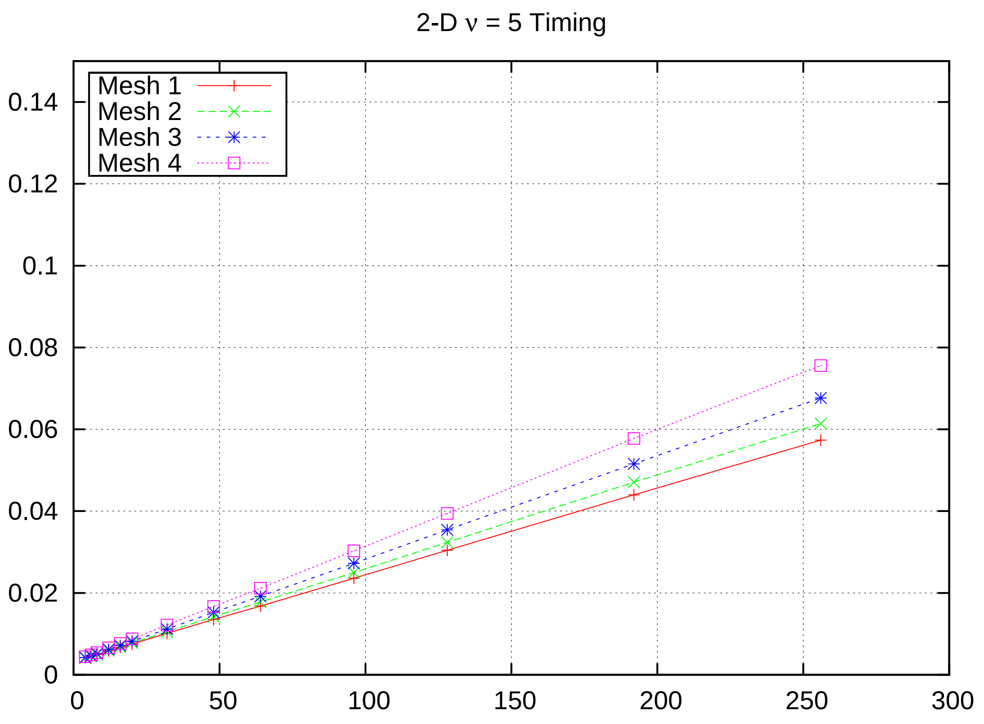
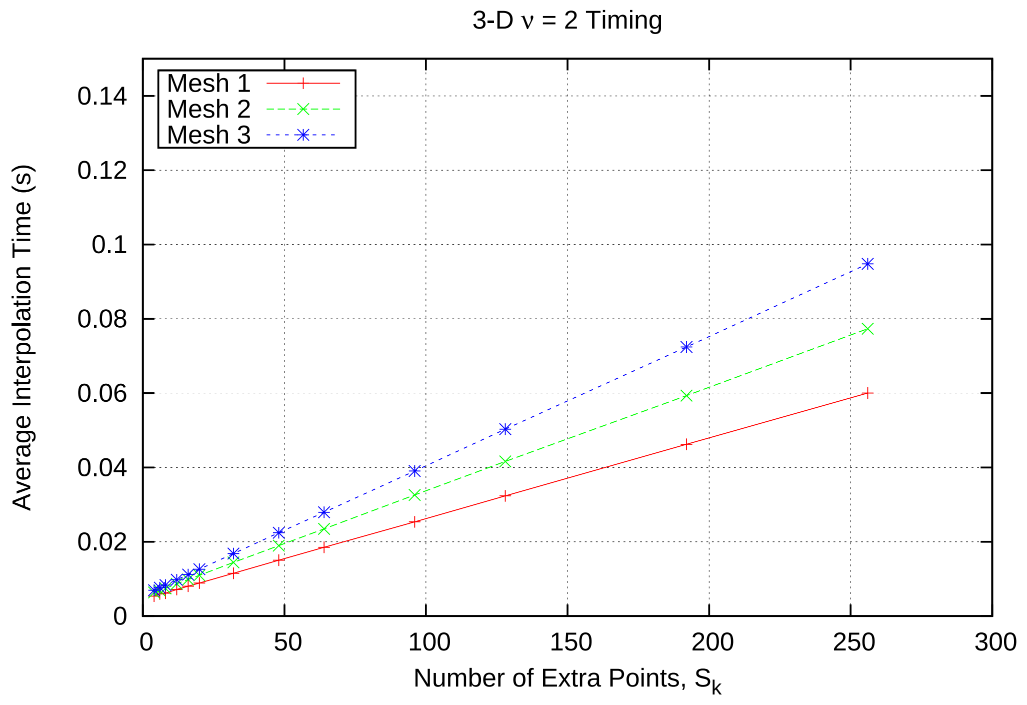


Figure 4.7: The Effects of the Variation of S_j, Interpolation Order, and Mesh Resolution on Average Time Spent per Interpolation for 2-D Meshes



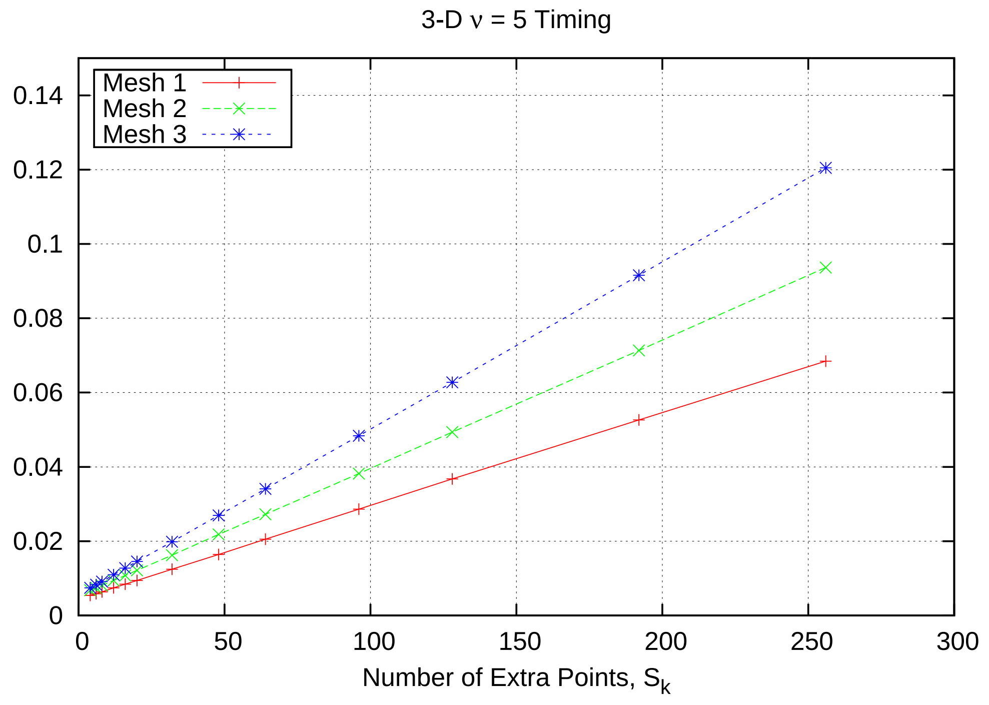


Figure 4.8: The Effects of the Variation of S_j, Interpolation Order, and Mesh Resolution on Average Time Spent per Interpolation for 3-D Meshes

It was found that in general increasing the dimension N, the mesh resolution, the order of interpolation ν, or S_k (extra points) caused an increase in interpolation time. The interpolation time increased linearly with the number of extra points. Also, the rate of increase of time for three-dimensional meshes is twice that of of two-dimensional meshes. The fastest of the three-dimensional interpolations, quadratic interpolation on a coarse mesh, was slower than half of all combinations of two-dimensional interpolations. Lastly, the interpolation for points against meshes of higher element density generally took longer than those of coarser element density. In the most extreme case, the `smbinterp` library exhibits a ten times increase in time by using the maximum density, dimension, number of extra points, and order of interpolation by comparison with the minimum values of any given permutation of the input parameters.

When solving for the interpolated values at each point Ξ the linear portion of the interpolant and the error approximation terms the numerical method require matrices of a larger rank for higher spatial dimensions, and is therefore more computationally costly. Furthermore, spatial querying of a kdtree suffers from both increased complexity in higher dimensions (exponential scaling) and more costly spatial queries when querying for a point amongst an increasingly large collection vertexes (linear scaling). All three of these effects are manifest in both two- and three-dimensions in figures 4.7 and 4.8, respectively.

4.1.3 Mesh Resolution Study

A mesh resolution study, similar in nature to those performed by Baker [10] and Galbraith [9], was performed using the `smbinterp` library. The purpose of the study was to determine how the accuracy of the interpolation library improved as the density of the underlying mesh was increased. The results of this study are show in figure 4.9 and 4.10. The main difference between the study performed herein and the one performed by Baker is that in Baker's study the destination points were placed in a geometrically optimal location: precisely between the source points. Because real-world use use of the library will not generally provide the same ideal vertex positions the results shown herein were generated using the same random destination points as were used in the previous study described in section 4.1. It was hoped that this would provide results for a more realistic use case.

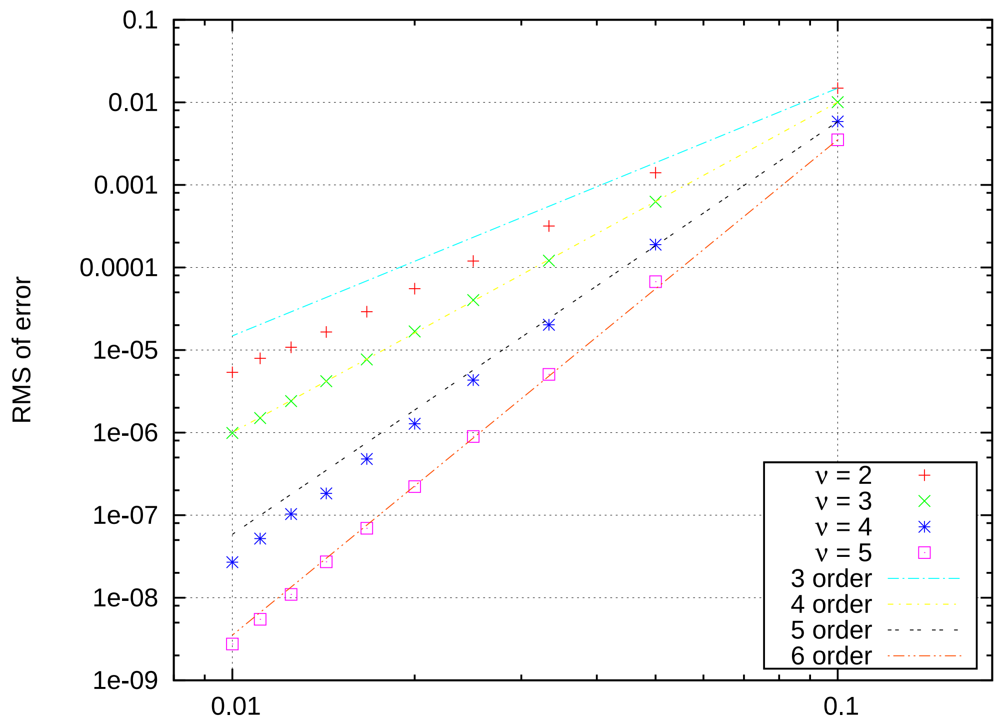


Figure 4.9: Scaling of RMS of Error vs. Mesh Spacing for 2-D Meshes

Another key difference between this and other studies is that the interpolations performed against the regular meshes produced for the three-dimensional studies consistently yielded singular covariance matrices. While this may be due to tolerances in the linear algebra libraries that are used by `smbinterp`, the covariance matrix was found to be singular regardless of how the extra points were selected and brings into question results obtained with a regular source mesh. While the library detects this occurrence by catching a thrown Python exception and subsequently performing the requisite calculations using a pseudo-inverse, the calculation of the covariance matrix using the pseudo inverse has a detrimental influence on the accuracy of the method. As mentioned before, a singular covariance matrix provides a non-unique solution to the least squares problem. Therefore, the same three-dimensional meshes as used in section 4.1 were used to calculate the RMS of error in figure 4.10.

Figures 4.9 and 4.10 plot the relationship between mesh spacing and RMS of error of all interpolations in the collection of destination vertexes for the two-dimensional and the three-dimensional test meshes, respectively. The x-axis represents the average spacing between vertexes

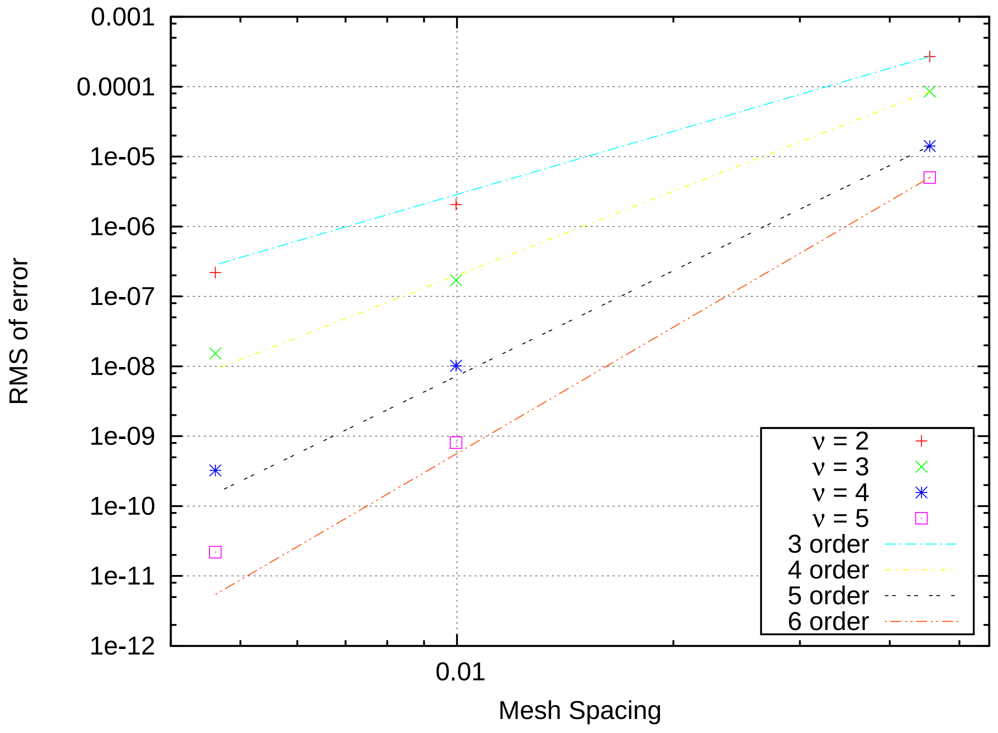


Figure 4.10: Scaling of RMS of Error vs. Mesh Spacing for 3-D Meshes

in the three-dimensional test meshes, and the exact spacing between the regular mesh elements in two dimensional test meshes. The y-axis was calculated as it was in section 4.1. Furthermore the lines in each plot are representative of the slope that each collection of data should follow if the underlying numerical method is truly accurate to that degree of accuracy. As an example, the collection of points for ν of 2 should be third-order accurate, and should follow a line with slope of 3; this is closely demonstrated in the plots.

Figure 4.9 shows the results of the resolution study for the two-dimensional meshes. As the meshes were refined the error decreased. The fourth- and sixth-order results (ν of 3 and 5) matched the slope lines almost exactly, whereas the third- and fifth-order results were slightly lower than expected for that level of accuracy.

Figure 4.10 shows the results of the resolution study for the three-dimensional meshes. Similar to the two-dimensional mesh resolution test, the RMS of error decreased as the mesh resolution increased. The three-dimensional test cases do not have the same banded effect as the two-dimensional test cases, but the error was slightly above the expected slope lines at the finest resolution for $\nu = 3, 4, 5$.

As mesh element size decreased, the RMS of error decreased as well for both dimensions. The RMS of error for the highest ν decreased more than that of the lowest ν. The RMS of error of the most coarse mesh ranges within a single order of magnitude, whereas the RMS of errors at the most fine spacing span four orders of magnitude for the two-dimensional meshes. The four lines are evenly spaced for the three-dimensional test case, whereas the results for two-dimensional meshes exhibit a slight banding, or unevenness between each order. Also, the data very closely matches the plotted lines of slope, indicating that the order of accuracy is indeed provided using this numerical method.

The rate at which error decreases as the average mesh element size decreases in figures 4.9 and 4.10 is indicative of the accuracy of the numerical method implemented in `smbinterp`. There is slight banding for the two-dimensional meshes between quadratic and cubic interpolation, and again for quartic an quintic interpolation. While this indicates that the method does not perfectly interpolate to those orders of accuracy, in general increasing the ν parameter of the `smbinterp` library provides a more accurate interpolation. Furthermore, the cases where the points diverge from the slope of appropriate order, the divergence occurs in a favorable direction. Also, the fine meshes experience a more significant decrease in RMS of error than the coarse meshes while increasing the order of approximation, ν. While this is an intuitive result, it emphasizes the notion that mesh density should be chosen to best match the underlying physical systems and to provide as accurate of results as possible.

4.2 Parallelization Results

The parallel algorithm employed by `smbinterp`, described in detail in section 3.4, was found to scale quasi-linearly to approximately 180 participating `minion` processes as shown in figure 4.11. The plot in figure 4.11 is a plot of *speedup* for the parallelization algorithm. Speedup is defined as the ratio of time to execute an algorithm sequentially (T_1) divided by the time to execute the algorithm with p processors [63], or:

$$S_p = \frac{T_1}{T_p}. \tag{4.5}$$

A parallel algorithm is considered to have ideal speedup if $S_p = p$. The x-axis in the figure represents the number of participating minion processes, and the y-axis is the speedup. As shown in figure 4.11, S_p is equal to p up to approximately 128 participating minions. While the trend shown in figure 4.11 seems favorable up to 200 points, the actual efficiency of performance is not perfect, and this particular visualization of performance can therefore be misleading.

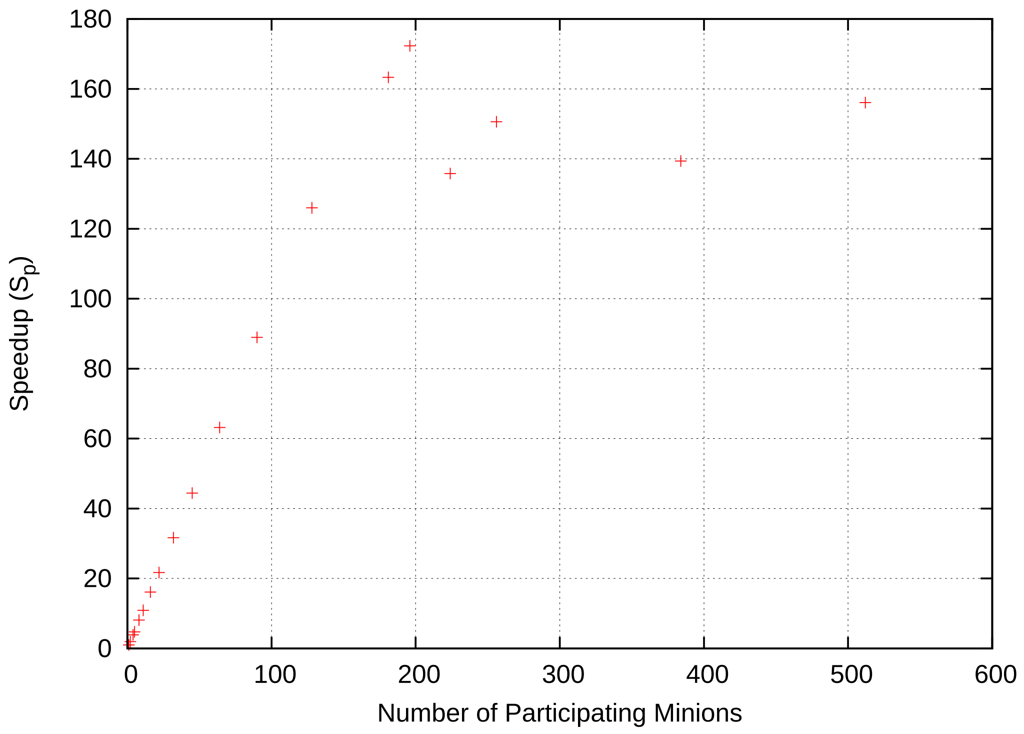


Figure 4.11: Speedup (S_p) of the Parallel Algorithm Employed by `smbinterp`

A more meaningful parameter for instrumenting the performance of a parallel algorithm is known as the efficiency of the algorithm, denoted E_p. Efficiency of a parallel algorithm is defined as the speedup divided by the number of participating processors, or:

$$E_p = \frac{T_p}{p}. \tag{4.6}$$

The efficiency of an algorithm ranges from 0 to 1, and is shown for `smbinterp` in the left plot in figure 4.12. If an algorithm does not have an efficiency of 1, it is usually indicative of communication overhead or bottlenecks of some form. The parallelization algorithm employed by the `smbinterp` library has near-linear speedup up to approximately 128 participating minions. It has an efficiency above 90 percent up to 181 participating nodes, but the efficiency drops substantially when using more minions.

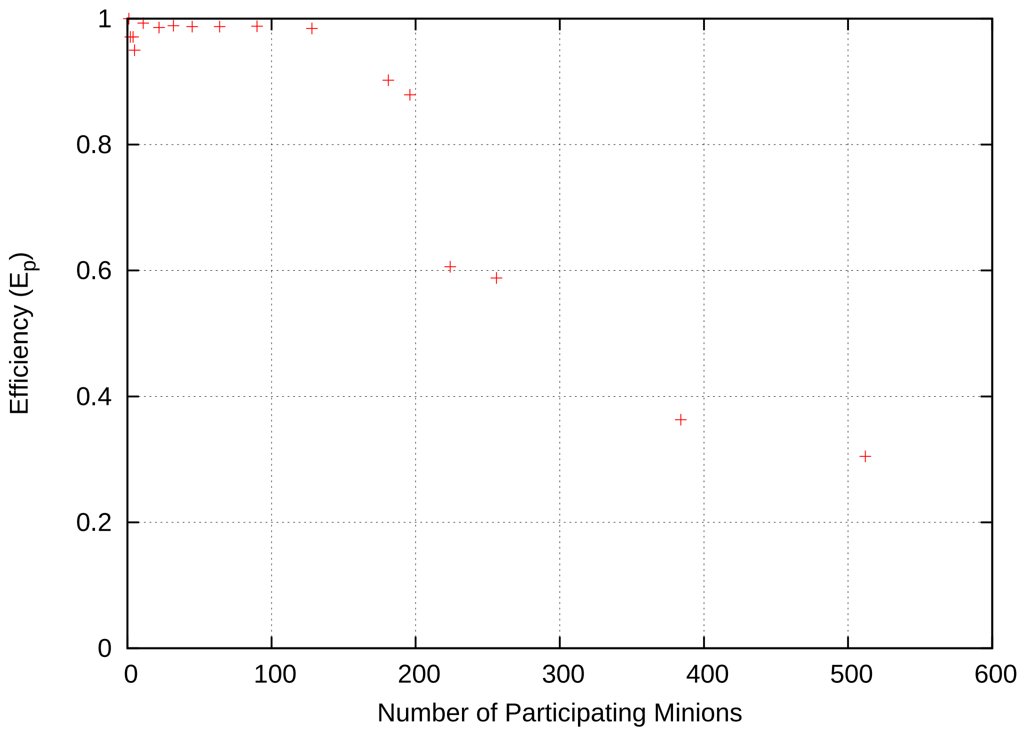


Figure 4.12: Efficiency (E_p) of the Parallel Algorithm Employed by `smbinterp`.

As shown in figure 3.5 all network traffic from the `master.py` process to each of the `minion.py` processes and back must pass through a single point: the `server.py` process. The processor load that this process consumes is shown in figure 4.13. The processor load scales linearly up to 180 participating `minion.py` processes, but levels off at two full processors worth of load (200 percent CPU utilization) after this point. The `manager` class of the `multiprocessing` Python module is implemented in a parallel fashion, which was observed by the existence of multiple threads of execution in the output of the `htop` Linux command, and by the fact that the process utilized more than one CPU (more than 100 percent). While the implementation of the `manager` class in the `multiprocessing` module seems to be heavily parallel, it can only consume two processors at any given time. The `server.py` process is a serial bottleneck.

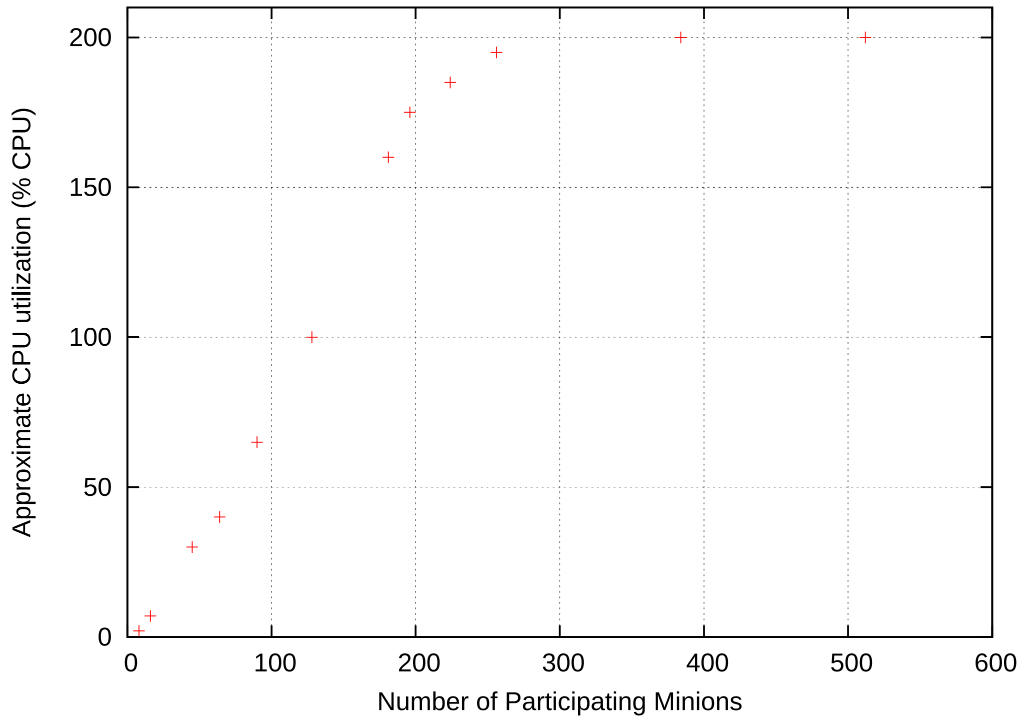


Figure 4.13: CPU Utilization of the `server.py` Process

The implementation of the `manager` class is the current bottleneck for the `smbinterp` library. While this scales approximately 180 times better than any other current high-order interpolation engine of this variety (all other known implementations are serial), there are a few avenues of research that might be suggested for future work. There exist many open source, high-performance message queuing systems. ØMQ was mentioned in chapter 2, and is a popular library amongst those who use python for high-performance scientific computing [53–55]. According to benchmarks performed by Knox [54], replacing the default, native Python queuing mechanism used by `smbinterp` should yield a seven time speedup. Furthermore, implementing the task distribution after a divide and conquer paradigm where minions are grouped under multiple `server.py` processes may allow the library to scale past 180 minions. The investigation of higher-complexity distribution schemes might be suggested for future work.

The implementation of the `smbinterp` library accomplished the main goals of this research. That is, to provide the scientific community with an open source library that implemented an n-th order accurate, distributed interpolation library. Tests were performed that showed that varying the ν parameter did increase the accuracy of the method. The library was implemented using distributed network task queues, and was found to scale to approximately 180 participating minions.

CHAPTER 5. CONCLUSIONS & RECOMMENDATIONS

The meshes used in multiphysics simulations are typically meshes of disparate topology and resolution. While these simulations represent the cutting edge of numerical simulations, linear interpolation is the method of interpolation generally employed when the spatial location of the source and destination meshes are not perfectly coincidental. Other limited solutions to this problem exist, but impose restrictions such as the requirement to collect and present higher-order terms at each point, limitations in how high of an order of interpolation can be requested, or the solution only being implemented in a serial fashion. The research contained herein yielded an interpolation library that provides interpolation to an arbitrary degree of accuracy, that is mesh agnostic, that only requires that the values (not derivatives) be acquired at the spatial locations of the points of interest, and that can be used in a parallel fashion in a high-performance computing environment.

The library, named `smbinterp`, provides an interpolation for a cloud of points to an arbitrary order of accuracy. It was shown, via a mesh resolution study, that the algorithm (and implementation thereof) provides the the end user with the expected level of accuracy, i.e. when performing cubic interpolation, the results are fourth-order accurate, quartic is fifth-order accurate, etc.

These higher-order results are obtained with a minor temporal penalty. Timing results were also published and showed at most a ten-fold increase of time required per interpolation using the most complicated settings (quintic interpolation on a fine three dimensional mesh using 256 extra points) by comparison with the least computationally intensive settings (quadratic interpolation on a coarse two-dimensional mesh using four extra points).

While the general results show that the library provides accurate interpolation care must be taken to wield it in an appropriate manner. Attempting to use the library with an order of interpolation that does not match the order of the underlying mathematical equations will yield unsatisfactory interpolation results. The underlying mathematical equations attempting to be fit should be understood, and an appropriate order of accuracy should be used. Furthermore, the underlying mesh should be designed according to good mesh construction practices; most importantly they should appropriately capture gradient information. This assumption was presumed safe as the most probable use of the library will be to interpolate data from one physical simulation to another where the onus of mesh design is on the designers of the meshes used in the simulations.

The second design goal of the `smbinterp` library was to make the interpolation algorithm that it implements easily usable with any mesh type or topology. The `smbinterp` library provides a convenient plugin infrastructure to provide this mesh agnosticism. Two example plugins are provided for reference in appendix A, and new plugins may be implemented with a relatively small effort. The two plugins are approximately 75 lines of code a piece, and implement plugins for meshes generated using the open source Gmsh meshing utility and for a general cloud of points.

Thirdly, `smbinterp` was implemented to take advantage of parallel computing environments. A parallel framework was implemented in the `smbinterp` library using built-in Python modules. This implementation was found to scale linearly approximately 180 participating compute processes.

Finally, `smbinterp` is available under a liberal license, and plans are in place to add it to the popular, open source scientific Python library named scipy [42]. Other future work may be suggested in two areas. First, the parallelization scheme, while it performs well, could be enhanced by using a faster queuing library and a more advanced participant partitioning scheme. Also, an algorithm to actively select parameters for the library based on the general shape of the solution may provide a temporally and computationally optimal choice of settings during a multiphysics simulation.

REFERENCES

[1] Wikipedia, 2011. `http://en.wikipedia.org/wiki/Multiphysics`, June.

[2] Shankaran, S., Shankaran, S., Alonso, J. J., Liou, M., and Liu, N., 2001. “A Multi-Code-Coupling Interface for Combustor/Turbomachinery Simulations” *AIAA paper 2001-974*, 39th AIAA Aerospace Sciences Meeting and Exhibit, Reno, NV, January.

[3] Medic, G., Kalitzin, G., You, D., Herrmann, M., Ham, F., van der Weide, E., Pitsch, H., and Alonso, J., 2006. “Integrated RANS/LES computations of turbulent flow through a turbofan jet engine” Center for Turbulence Research Annual Research Briefs, Stanford, CA.

[4] Hahn, S., Duraisamy, K., Iaccarino, G., Nagarajan, S., Sitaraman, J., Wu, X., Alonso, J. J., Baeder, J. D., Lle, S. K., Moin, P., and Schmitz, F., 2006. “Coupled High-Fidelity URANS Simulation for Helicopter Applications” Center for Turbulence Research Annual Research Briefs, Stanford, CA.

[5] Alonso, J., Hahn, S., Ham, F., Herrmann, M., Iaccarino, G., Kalitzin, G., LeGresley, P., Mattsson, K., Medic, G., Moin, P., Pitsch, H., Schlter, J., Svard, M., der Weide, E. V., You, D., and Wu, X., 2006. “CHIMPS: A High-Performance Scalable Module for Multi-Physics Simulations” *AIAA paper* 5274, 42nd AIAA/ASME/SAE/ASEE Joint Propulsion Conference & Exhibit, Sacramento, CA, July.

[6] Wikipedia, 2011. `http://en.wikipedia.org/wiki/Message_Passing_Interface`, June.

[7] Wikipedia, 2011. `http://en.wikipedia.org/wiki/API`, June.

[8] Hahn, S., Iaccarino, G., Ananthan, S., and Baeder, J. D., 2009. “Extension of CHIMPS for unstructured overset simulation and higher-order interpolation” Center for Turbulence Research Annual Research Briefs, Stanford, CA.

[9] Galbraith, M. C., and Miller, J. H., 2006. “Development and Application of a General Interpolation Algorithm” *AIAA paper 2006-3854*, 24th AIAA Applied Aerodynamics Conference, San Francisco, CA, June.

[10] Baker, T. J., 2003. “Interpolation from a cloud of points.” *International Meshing Roundtable*(12), September, pp. 55–63.

[11] Geuzaine, C., and Remacle, J.-F., 2009. “Gmsh: A 3-d finite element mesh generator with built-in pre- and post-processing facilities.” *International Journal for Numerical Methods in Engineering*, **79**(11), pp. 1309–1331.

[12] Sederberg, T. W., 2008. Computer aided geometric design CS557 Class Website, December `http://cagd.cs.byu.edu/~557/text/cagd.pdf`.

[13] Werner, W., 1984. “Polynomial interpolation: Lagrange versus newton.” *Mathematics of Computation,* **43**(167), pp. 205–217.

[14] Runge, C., 1901. “Über empirische funktionen und die interpolation zwischen äquidistanten ordinaten.” *Zeitschrift fr Mathematik und Physik,* **46**, pp. 224–243.

[15] Berrut, J.-P., and Trefethen, L. N. “Barycentric lagrange interpolation.” *SIAM Rev,* **46**, pp. 501–517.

[16] Fornberg, B., and Zuev, J., 2007. “The runge phenomenon and spatially variable shape parameters in rbf interpolation.” *Computers & Mathematics with Applications,* **54**(3), pp. 379 – 398.

[17] Shukla, R. K., and Zhong, X., 2005. “Derivation of high-order compact finite difference schemes for non-uniform grid using polynomial interpolation.” *Journal of Computational Physics,* **204**(2), pp. 404 – 429.

[18] Dyn, N., Gregory, J. A., and Levin, D., 1991. “Analysis of uniform binary subdivision schemes for curve design.” *Constructive Approximation,* **7**, pp. 127–147.

[19] Piegl, L., and Tiller, W., 1997. *The NURBS Book.*, 2 ed. Springer-Verlag, New York.

[20] Yang, Y., and Lian, J., 2010. “Making 3d object surfaces smoother.” *Computing in Science Engineering,* **12**(3), May-June, pp. 44 –51.

[21] DeRose, T., Kass, M., and Truong, T., 1998. “Subdivision surfaces in character animation.” In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques,* SIGGRAPH ’98, ACM, pp. 85–94.

[22] Catmull, E., and Clark, J., 1978. “Recursively generated b-spline surfaces on arbitrary topological meshes.” *Computer-Aided Design,* **10**(6), pp. 350 – 355.

[23] Oswald, P., and Schrder, P., 2003. “Composite primal/dual -subdivision schemes.” *Computer Aided Geometric Design,* **20**(3), pp. 135 – 164.

[24] Dyn, N., Levine, D., and Gregory, J. A., 1990. “A butterfly subdivision scheme for surface interpolation with tension control.” *ACM Trans. Graph.,* **9**, April, pp. 160–169.

[25] Zorin, D., Schröder, P., and Sweldens, W., 1996. “Interpolating subdivision for meshes with arbitrary topology.” In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, SIGGRAPH ’96, ACM, pp. 189–192.

[26] Blender, 2011. `http://blender.org`, June.

[27] McDonnell, K. T., Chang, Y.-S., and Qin, H., 2004. “Interpolatory, solid subdivision of unstructured hexahedral meshes.” *The Visual Computer,* **20**, pp. 418–436.

[28] Bajaj, C., Schaefer, S., Warren, J., and Xu, G., 2002. “A subdivision scheme for hexahedral meshes.” *The Visual Computer,* **18**, pp. 343–356.

[29] Chang, Y.-S., McDonnell, K. T., and Qin, H., 2002. "A new solid subdivision scheme based on box splines." In *Proceedings of the seventh ACM symposium on Solid modeling and applications*, SMA '02, ACM, pp. 226–233.

[30] Chang, Y.-S., McDonnell, K. T., and Qin, H., 2003. "An interpolatory subdivision for volumetric models over simplicial complexes." In *Proceedings of the Shape Modeling International 2003*, IEEE Computer Society.

[31] Szymczak, A., Rossignac, J., and King, D., 2002. "Piecewise regular meshes: Construction and compression." *Graphical Models,* **64**(3-4), pp. 183 – 198.

[32] Wikipedia, 2011. `http://en.wikipedia.org/wiki/BSP_tree`, June.

[33] Wikipedia, 2011. `http://en.wikipedia.org/wiki/B-tree`, June.

[34] Wikipedia, 2011. `http://en.wikipedia.org/wiki/R-tree`, June.

[35] Wikipedia, 2011. `http://en.wikipedia.org/wiki/Kd-tree`, June.

[36] Wikipedia, 2011. `http://en.wikipedia.org/wiki/Octree`, June.

[37] PostgreSQL – The World's Most Advanced Open Source Database, 2011. `http://postgresql.org`, June.

[38] PostGIS Documentation, 2011. `http://postgis.refractions.net/documentation`, June.

[39] Bonet, J., and Peraire, J., 1991. "An alternating digital tree (adt) algorithm for 3d geometric searching and intersection problems." *International Journal for Numerical Methods in Engineering,* **31**(1), pp. 1–17.

[40] Katayama, N., and Satoh, S., 1997. "The sr-tree: an index structure for high-dimensional nearest neighbor queries." *SIGMOD Rec.,* **26**, June, pp. 369–380.

[41] Garth, C., and Joy, K. I., 2010. "Fast, memory-efficient cell location in unstructured grids for visualization." *IEEE Transactions on Visualization and Computer Graphics,* **16**, November, pp. 1541–1550.

[42] Scientific Tools for Python, 2011. `http://scipy.org`, June.

[43] Spatial algorithms and data structures (scipy.spatial), 2011. `http://docs.scipy.org/doc/scipy/reference/spatial.html`, June.

[44] Wikipedia, 2011. `http://en.wikipedia.org/wiki/Embarrassingly_parallel`, June.

[45] Devlin, B., Gray, J., Laing, B., and Spix, G., 1999. "Scalability terminology: Farms, clones, partitions, packs, racs and raps." *Computing Research Repository*.

[46] Cardellini, V., Casalicchio, E., Colajanni, M., and Yu, P. S., 2002. "The state of the art in locally distributed web-server systems." *ACM Comput. Surv.,* **34**, June, pp. 263–311.

[47] Slony – Enterprise-Level Replication System for PostgreSQL, 2011. `http://slony.info/`, Feb.

[48] The Multi-Master and Synchronous Replication System for PostgreSQL, 2011. `http://pgcluster.projects.postgresql.org/`, Feb.

[49] Bucardo, 2011. `http://bucardo.org`, Feb.

[50] Python `multiprocessing`, 2011. `http://docs.python.org/library/multiprocessing.html`, June.

[51] Python Global Interpreter Lock, 2011. `http://docs.python.org/glossary.html`, June.

[52] ØMQ, 2011. `http://www.zeromq.org`, June.

[53] pyzmq, 2011. `http://www.zeromq.org/bindings:python`, June.

[54] Knox, B., 2011. `http://taotetek.wordpress.com/2011/02/03/python`, June.

[55] Salt, 2011. `https://github.com/thatch45/salt`, June.

[56] Wikipedia, 2011. `http://en.wikipedia.org/wiki/Simplex`, June.

[57] Rudin, W., 1976. *Principles of Mathematical Analysis*., 3 ed. McGraw-Hill.

[58] Coxeter, H., 2969. *Introduction to Geometry*., 2 ed. Wiley.

[59] Chapra, S. C., 2006. *Numerical Methods for Engineers*., 5 ed. McGraw-Hill.

[60] Khoshniat, M., Stuhne, G., and Steinman, D., 2003. "Relative performance of geometric search algorithms for interpolating unstructured mesh data." In *Medical Image Computing and Computer-Assisted Intervention - MICCAI 2003*, R. Ellis and T. Peters, eds., Vol. 2879 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, pp. 391–398.

[61] Bercovier, M., Pironneau, O., and Sastri, V., 1983. "Finite elements and characteristics for some parabolic-hyperbolic problems." *Applied Mathematical Modelling*, **7**(2), pp. 89 – 96.

[62] Wikipedia, 2011. `http://en.wikipedia.org/wiki/Delaunay_triangulation`, June.

[63] Wikipedia, 2011. `http://en.wikipedia.org/wiki/Speedup`, June.

APPENDIX A. `smbinterp` SOURCE CODE

A.1 `smbinterp` library

```
from setuptools import setup, find_packages
from interp import __version__

setup(
          name             = 'smbinterp',
          version          = __version__,
          packages         = find_packages(),

          install_requires = [
                               'progressbar',
                               'scipy',
                               'numpy',
                             ],


          author       = "Stephen M. McQuay",
          author_email = "stephen@mcquay.me",
          url          = "https://mcquay.me/hg/research",
          license      = "GPL",
      )
```

: ../research/setup.py

```
import sys

import numpy as np

from functools import wraps
```

```
import itertools

import interp
import logging
log = logging.getLogger('interp')

AGGRESSIVE_ERROR_SOLVE = True
RAISE_PATHOLOGICAL_EXCEPTION = False

def get_phis(X, R):
    """
        The get_phis function is used to get barycentric coordonites for a
        point on a triangle or tetrahedron. This is equation 3.3

        in 2D:

        X - the destination point (2D)
            X = [0,0]
        R - the three points that make up the 2-D triangular simplex
            R = [[-1, -1], [0, 2], [1, -1]]

        this will return [0.333, 0.333, 0.333]


        in 3D:

        X - the destination point (3D)
            X = [0,0,0]
        R - the four points that make up the 3-D simplex (tetrahedron)
            R = [
                [ 0.0000,  0.0000,  1.0000],
                [ 0.9428,  0.0000, -0.3333],
                [-0.4714,  0.8165, -0.3333],
                [-0.4714, -0.8165, -0.3333],
              ]

    this will return [0.25, 0.25, 0.25, 0.25]
  """

  # equations 3.12 and 3.10
  if len(X) == 2:
    log.debug("running 2D")
    A = np.array([
                  [      1,       1,       1],
                  [R[0][0], R[1][0], R[2][0]],
                  [R[0][1], R[1][1], R[2][1]],
                  ])
    b = np.array([     1,
                    X[0],
                    X[1]
                ])
  elif len(X) == 3:
    log.debug("running 3D")
    A = np.array([
                  [      1,       1,       1,       1 ],
                  [R[0][0], R[1][0], R[2][0], R[3][0]],
                  [R[0][1], R[1][1], R[2][1], R[3][1]],
                  [R[0][2], R[1][2], R[2][2], R[3][2]],
                  ])
    b = np.array([     1,
                    X[0],
                    X[1],
                    X[2]
                ])
  else:
    raise Exception("inapropriate demension on X")

  try:
    phi = np.linalg.solve(A,b)
  except np.linalg.LinAlgError as e:
    msg = "calculation of phis yielded a linearly dependant system (%s)" % e
    log.error(msg)
```

```
        # raise Exception(msg)
        phi = np.dot(np.linalg.pinv(A), b)

    log.debug("phi: %s", phi)

    return phi

def qlinear(X, R):
    """
        this calculates the linear portion of q from R to X

        This is equation 3.5

        X = destination point
        R = a inter.grid object; must have R.points and R.q
    """

    phis = get_phis(X, R.verts)
    qlin = np.sum([q_i * phi_i for q_i, phi_i in zip(R.q, phis)])

    log.debug("phis: %s", phis)
    log.debug("qlin: %s", qlin)

    return phis, qlin

def get_error(phi, R, S, order = 2):
    """
        Calculate the error approximation terms, returning the unknowns
        a,b, and c in equation 3.13.
    """
    B = [] # equation (3.15
    w = [] # equation (3.16

    cur_pattern  = pattern(len(phi), order)
    log.info("pattern: %s" % cur_pattern)

```

```
  for (s,q) in zip(S.verts, S.q):
    cur_phi, cur_qlin = qlinear(s, R)
    l = []
    for i in cur_pattern:
      cur_sum = cur_phi[i[0]]
      for j in i[1:]:
        cur_sum *= cur_phi[j]
      l.append(cur_sum)

    B.append(l)
    w.append(q - cur_qlin)

  log.info("B: %s" % B)
  log.info("w: %s" % w)


  B = np.array(B)
  w = np.array(w)

  A = np.dot(B.T, B)
  b = np.dot(B.T, w)

  try:
    abc = np.linalg.solve(A,b)
  except np.linalg.LinAlgError as e:
    log.error("linear calculation went bad, resorting to np.linalg.pinv: %s" %
        e)
    if not AGGRESSIVE_ERROR_SOLVE:
      return None, None
    abc = np.dot(np.linalg.pinv(A), b)

  error_term = 0.0
  for (a, i) in zip(abc, cur_pattern):
    cur_sum = a
    for j in i:
      cur_sum *= phi[j]
```

```
        error_term += cur_sum

    log.debug("error_term: %s" % error_term)
    return error_term, abc

def run_baker(X, R, S, order=2):
    """
        This is the main function to call to get an interpolation to X from the
        input meshes

        X -- the destination point

        R = Simplex
        S = extra points
    """
    log.debug("order = %d" % order)
    log.debug("extra points = %d" % len(S.verts))

    answer = {
                'qlin': None,
                'error': None,
                'final': None,
              }
    # calculate values only for the simplex triangle
    phi, qlin = qlinear(X, R)

    if order == 1:
        answer['qlin'] = qlin
        answer['final'] = qlin
        return answer
    elif order in xrange(2,11):
        error_term, abc = get_error(phi, R, S, order)

        # if a pathological vertex configuration was encountered and
        # AGGRESSIVE_ERROR_SOLVE is False, get_error will return (None, None)
        # indicating that only linear interpolation should be performed
        if (error_term is None) and (abc is None):
          if RAISE_PATHOLOGICAL_EXCEPTION:
            raise np.linalg.LinAlgError("Pathological Vertex Configuration
                Detected")
          answer['qlin'] = qlin
          answer['final'] = qlin
          return answer
    else:
        raise Exception('unsupported order "%d" for baker method' % order)

    q_final = qlin + error_term

    answer['qlin' ] =  qlin
    answer['error'] =  error_term
    answer['final'] =  q_final
    answer['abc'  ] =  abc

    log.debug(answer)

    return answer


def memoize(f):
    """
        for more information on what I'm doing here, please read:
        http://en.wikipedia.org/wiki/Memoize
    """
    cache = {}
    @wraps(f)
    def memf(simplex_size, nu):
        x = (simplex_size, nu)
        if x not in cache:
          log.debug("adding to cache: %s", x)
          cache[x] = f(simplex_size, nu)
        return cache[x]
    return memf
```

```python


@memoize
def pattern(simplex_size, nu):
  """
    This function returns the pattern requisite to compose the error
    approximation function, and the matrix B.
  """
  log.debug("pattern: simplex: %d, order: %d" % (simplex_size, nu))

  r = []
  for i in itertools.product(xrange(simplex_size), repeat = nu):
    if len(set(i)) !=1:
      r.append(tuple(sorted(i)))
  unique_r = list(set(r))
  return unique_r
```

: ../research/interp/baker/__init__.py

```python
import os

import numpy as np

import logging
log = logging.getLogger("interp")

def rms(errors):
  """
    root mean square calculation
  """

  # slow pure python way for reference:
  # r = 0.0
  # for i in errors:
  #   r += np.power(i, 2)
  # r = np.sqrt(r / len(errors))
  # return r
```

```

    return np.sqrt((errors**2).mean())

def baker_exact_2D(X):
    """
      the exact function (2D) used from baker's article (for testing, slightly
      modified)
    """
    x ,y = X

    answer = np.power((np.sin(x * np.pi) * np.cos(y * np.pi)), 2)
    log.debug(answer)
    return answer

def friendly_exact_2D(X):
    """
      A friendlier 2D func
    """
    x ,y = X
    answer = 1.0 + x*x + y*y
    log.debug(answer)
    return answer

def baker_exact_3D(X):
    """
      the exact function (3D) used from baker's article (for testing)
    """
    x = X[0]
    y = X[1]
    z = X[2]
    answer = np.power((np.sin(x * np.pi / 2.0) * np.sin(y * np.pi / 2.0) * np.
        sin(z * np.pi / 2.0)), 2)
    log.debug(answer)
    return answer

def friendly_exact_3D(X):
  x,y,z = X
  return 1 + x*x + y*y + z*z

def scipy_exact_2D(X):
  x,y = X
  return x*(1-x)*np.cos(4*np.pi*x) * np.sin(4*np.pi*y**2)**2

def improved_answer(answer, exact):
  if not answer['error']:
    # was probably just a linear interpolation
    return False

  log.debug('qlin:  %s' % answer['qlin'])
  log.debug('error: %s' % answer['error'])
  log.debug('final: %s' % answer['final'])
  log.debug('exact: %s' % exact)

  if np.abs(answer['final'] - exact) <= np.abs(answer['qlin'] - exact):
    log.debug(":) improved result")
    return True
  else:
    log.debug(":( damaged result")
    return False

def improved(qlin, err, final, exact):
  if np.abs(final - exact) <= np.abs(qlin - exact):
    return True
  else:
    return False
```

: ../research/interp/tools.py

A.2 `smbinterp` Plugin System

```
import sys
from    collections import defaultdict
```

```
import pickle

from xml.dom.minidom import Document

import numpy as np
from scipy.spatial import KDTree

from interp.baker import run_baker
from interp.baker import get_phis

import logging
log = logging.getLogger("interp")

MAX_SEARCH_COUNT = 256

class grid(object):
  def __init__(self, verts = None, q = None):
    """
      verts = array of arrays (if passed in, will convert to numpy.array)
              [
                [x0,y0 <, z0>],
                [x1,y1 <, z1>],
                ...
              ]

      q     = array (1D) of physical values
    """

    if verts != None:
      self.verts = np.array(verts)
      self.tree  = KDTree(self.verts)

    if q != None:
      self.q = np.array(q)

    self.cells           = {}
    self.cells_for_vert = defaultdict(list)

  def get_containing_simplex(self, X):
    if not self.cells:
      raise Exception("cell connectivity is not set up")

    # get closest point
    (dist, indicies) = self.tree.query(X, 2)
    closest_point = indicies[0]

    log.debug('X: %s' % X)
    log.debug('point index: %d' % closest_point)
    log.debug('actual point %s' % self.verts[closest_point])
    log.debug('distance = %0.4f' % dist[0])

    simplex = None
    checked_cells = []
    cells_to_check = list(self.cells_for_vert[closest_point])

    attempts = 0
    while not simplex and cells_to_check:
      attempts += 1

      if attempts > MAX_SEARCH_COUNT:
        raise Exception("Is the search becoming exhaustive? (%d attempts)" %
            attempts)

      cur_cell = cells_to_check.pop(0)
      checked_cells.append(cur_cell)

      if cur_cell.contains(X, self):
        simplex = cur_cell
        continue

      for neighbor in cur_cell.neighbors:
        if (neighbor not in checked_cells) and (neighbor not in cells_to_check):
          cells_to_check.append(neighbor)

    if not simplex:
      raise Exception('no containing simplex found')

    log.debug(”simplex vert indicies: %s” % simplex.verts)
    R = self.create_mesh(simplex.verts)
    log.debug(”R:\n%s”, R)

    log.debug('total attempts before finding simplex: %d' % attempts)
    return R

  def create_mesh(self, indicies):
    ”””
      this function takes a list of indicies, and then creates and returns a
      grid object (collection of verts and q).

      note: the input is indicies, the grid contains verts
    ”””

    return grid(self.verts[indicies], self.q[indicies])

  def get_simplex_and_nearest_points(self, X, extra_points = 3):
    ”””
      this returns two grid objects: R and S.

      R is a grid object that is a containing simplex around point X

      S : some verts from all points that are not the simplex
    ”””
    simplex_size = self.dim + 1
    log.debug(”extra verts: %d” % extra_points)
    log.debug(”simplex size: %d” % simplex_size)

    r_mesh = self.get_containing_simplex(X)

    # and some UNIQUE extra verts
    (dist, indicies) = self.tree.query(X, simplex_size + extra_points)
    log.debug("extra indicies: %s" % indicies)

    unique_indicies = []
    for index in indicies:
      close_point_in_R = False
      for rvert in r_mesh.verts:
        if all(rvert == self.verts[index]):
          close_point_in_R = True
          break

      if not close_point_in_R:
        unique_indicies.append(index)
      else:
        log.debug('throwing out %s: %s' % (index, self.verts[index]))

    log.debug("indicies: %s" % indicies)
    log.debug("unique indicies: %s" % unique_indicies)
    s_mesh = self.create_mesh(unique_indicies)

    return (r_mesh, s_mesh)

  def run_baker(self, X, order = 2, extra_points = 3):
    (R, S) = self.get_simplex_and_nearest_points(X, extra_points)
    answer = run_baker(X, R, S, order)
    return answer

  def for_qhull_generator(self):
    """
      this returns a generator that should be fed into qdelaunay
    """

    yield str(len(self.verts[0]));
    yield '%d' % len(self.verts)

    for p in self.verts:
      yield "%f %f %f" % tuple(p)

  def for_qhull(self):
    """
      this returns a single string that should be fed into qdelaunay
    """
    r  = '%d\n' % len(self.verts[0])
    r += '%d\n' % len(self.verts)
    for p in self.verts:
      # r += "%f %f %f\n" % tuple(p)
      r += "%s\n" % " ".join("%f" % i for i in p)
    return r

  def __str__(self):
    r = ''
    assert( len(self.verts) == len(self.q) )
    for c, i in enumerate(zip(self.verts, self.q)):
      r += "%d vert(%s): q(%0.4f)" % (c,i[0], i[1])
      cell_str = ", ".join([str(f.name) for f in self.cells_for_vert[c]])
      r += "  cells: [%s]" % cell_str
      r += "\n"
    if self.cells:
      for v in self.cells.itervalues():
        r += "%s\n" % v
    return r

  def normalize_q(self, new_max = 0.1):
    largest_number = np.max(np.abs(self.q))
    self.q *= new_max/largest_number


  def dump_to_blender_files(self, pfile = '/tmp/points.p', cfile = '/tmp/cells
      .p'):
```

```
    if len(self.verts[0]) == 2:
      pickle.dump([(p[0], p[1], 0.0) for p in self.verts], open(pfile, 'w'))
    else:
      pickle.dump([(p[0], p[1], p[2]) for p in self.verts], open(pfile, 'w'))

    pickle.dump([f.verts for f in self.cells.itervalues()], open(cfile, 'w'))

  def get_xml(self):
    doc = Document()
    ps = doc.createElement("points")
    doc.appendChild(ps)
    for i in zip(self.verts, self.q):
      p = doc.createElement("point")

      p.setAttribute("x", str(i[0][0]))
      p.setAttribute('y', str(i[0][1]))
      p.setAttribute('z', str(i[0][2]))
      p.setAttribute('q', str(i[1]   ))
      ps.appendChild(p)

    return doc

  def toxml(self):
    return self.get_xml().toxml()
  def toprettyxml(self):
    return self.get_xml().toprettyxml()


class cell(object):
  def __init__(self, name):
    self.name      = name
    self.verts     = []
    self.neighbors = []

  def add_vert(self, v):
    """
```

```
      v should be an index into grid.verts
    """
    self.verts.append(v)

  def add_neighbor(self, n):
    """
      reference to another cell object
    """
    self.neighbors.append(n)

  def contains(self, X, G):
    """
     X = point of interest
     G = corrensponding grid object (G.verts)

     because of the way i'm storing things, a cell simply stores indicies,
     and so one must pass in a reference to the grid object containing real
     verts.

     this simply calls grid.simplex.contains
    """
    return contains(X, [G.verts[i] for i in self.verts])

  def __str__(self):
    # neighbors = [str(i.name) for i in self.neighbors]
    return '<cell %s: verts: %s neighbor count: %s>' %\
          (
            self.name,
            self.verts,
            len(self.neighbors),
            # ", ".join(neighbors)
          )

  __repr__ = __str__


```

```
TOL = 1e-8

def contains(X, R):
  """
    tests if X (point) is in R

    R is a simplex, represented by a list of n-degree coordinates
  """
  phis = get_phis(X, R)

  r = True
  if [i for i in phis if i < 0.0 - TOL]:
    r = False
  return r
```

: ../research/interp/grid/__init__.py

```
import pickle

from   itertools   import combinations
from   collections import defaultdict

import numpy as np
from   scipy.spatial import KDTree

from   interp.grid         import grid
from   interp.grid         import cell

import logging
log = logging.getLogger('interp')



THREE_NODE_TRIANGLE = 2
FOUR_NODE_TET       = 4

EDGES_FOR_FACE_CONNECTIVITY    = 2
```

```
EDGES_FOR_VOLUME_CONNECTIVITY = 3



class ggrid(grid):

  def __init__(self, filename, dimension = 3):
    """
      construct an interp.grid.grid-compliant grid
      object out of a {2,3}D gmsh file
    """
    self.dim = dimension
    log.debug("dimension: %d", self.dim)

    gmsh_file = open(filename, 'r')


    gmsh_file.readline() # $MeshFormat
    fmat = gmsh_file.readline()
    gmsh_file.readline() # $EndMeshFormat

    gmsh_file.readline() # $Nodes

    node_count = int(gmsh_file.readline())

    self.verts = np.empty((node_count, dimension))
    self.q     = np.empty(node_count)

    for i in xrange(node_count):
      cur_line = gmsh_file.readline()
      (index, x,y,z) = cur_line.split()
      index = int(index) - 1

      self.verts[i][0] = float(x)
      self.verts[i][1] = float(y)

      if self.dim == 3:
        self.verts[i][2] = float(z)


    self.tree = KDTree(self.verts)

    # initialize rest of structures about to be populated (cells,
    # cells_for_vert)
    grid.__init__(self)

    gmsh_file.readline() # $EndNodes
    gmsh_file.readline() # $Elements

    # temporary dict used to compute cell connectivity
    neighbors = {}

    element_count = int(gmsh_file.readline())
    for i in xrange(element_count):
      cur_line = gmsh_file.readline()
      cur_line = cur_line.split()
      cur_cell_index, node_type, rest = (int(cur_line[0]),
                                          int(cur_line[1]),
                                          [int(j) for j in  cur_line[2:]])

      if    (node_type == THREE_NODE_TRIANGLE and self.dim == 2) \
         or (node_type == FOUR_NODE_TET       and self.dim == 3):
        points_for_cur_cell = [i-1 for i in rest[rest[0]+1:]]

        cur_cell = cell(cur_cell_index)

        for cur_point in points_for_cur_cell:
          self.cells_for_vert[cur_point].append(cur_cell)

        cur_cell.verts       = points_for_cur_cell

        self.cells[cur_cell_index] = cur_cell
      edges = [tuple(sorted(i)) for i in combinations(points_for_cur_cell,
          self.dim)]

      for edge in edges:
        if edge in neighbors:
          neighbors[edge].append(cur_cell_index)
        else:
          neighbors[edge] = [cur_cell_index]

    for k,v in neighbors.iteritems():
      if len(v) > 1:
        self.cells[v[0]].add_neighbor(self.cells[v[1]])
        self.cells[v[1]].add_neighbor(self.cells[v[0]])
```

: ../research/interp/grid/gmsh.py

```
import re
import logging

log = logging.getLogger("interp")

from interp.grid import grid as basegrid, cell

from subprocess import Popen, PIPE

def get_qdelaunay_dump(g):
  """
    pass in interp.grid g, and get back lines from a qhull triangulation:

    qdelaunay Qt f
  """
  cmd = 'qdelaunay Qt f'
  p = Popen(cmd.split(), bufsize=1, stdin=PIPE, stdout=PIPE)
  so, se = p.communicate(g.for_qhull())
  for i in so.splitlines():
    yield i

```

```
def get_qdelaunay_dump_str(g):
  return "\n".join(get_qdelaunay_dump(g))

def get_index_only(g):
  cmd = 'qdelaunay Qt i'
  p = Popen(cmd.split(), bufsize=1, stdin=PIPE, stdout=PIPE)
  so, se = p.communicate(g.for_qhull())
  for i in so.splitlines():
    yield i

def get_index_only_str(g):
  return "\n".join(get_index_only(g))

class dgrid(basegrid):
  cell_re = re.compile(r'''
                        -\s+(?P<cell>f\d+).*?
                        vertices:\s(?P<verts>.*?)\n.*?
                        neighboring\s facets:\s+(?P<neigh>[\sf\d]*)
                      ''', re.S|re.X)

  vert_re   = re.compile(r'''
                        (p\d+)
                      ''', re.S|re.X)

  def __init__(self, verts, q = None):
    self.dim = len(verts[0])
    basegrid.__init__(self, verts,q)
    self.construct_connectivity()

  def construct_connectivity(self):
    """
      a call to this method prepares the internal connectivity structure.
    """
    log.info('start')
    qdelaunay_string = get_qdelaunay_dump_str(self)

    with open('/tmp/qdel.out', 'w') as of:
      of.write(qdelaunay_string)

    cell_to_cells = []
    for matcher in dgrid.cell_re.finditer(qdelaunay_string):
      d = matcher.groupdict()

      cell_name           = d['cell']
      verticies           = d['verts']
      neighboring_cells   = d['neigh']

      cur_cell = cell(cell_name)
      self.cells[cell_name] = cur_cell

      for v in dgrid.vert_re.findall(verticies):
        vertex_index = int(v[1:])
        cur_cell.add_vert(vertex_index)
        self.cells_for_vert[vertex_index].append(cur_cell)

      nghbrs = [(cell_name, i) for i in  neighboring_cells.split()]
      cell_to_cells.extend(nghbrs)
    log.debug(cell_to_cells)

    for rel in cell_to_cells:
      if rel[1] in self.cells:
        self.cells[rel[0]].add_neighbor(self.cells[rel[1]])

    log.debug(self.cells)
    log.info('end')
```

: ../research/interp/grid/delaunay.py

A.3 Parallelization Scripts

```
from multiprocessing.managers import BaseManager
import Queue
```

```

tasks_q    = Queue.Queue()
results_q  = Queue.Queue()
minions_q  = Queue.Queue()
master_q   = Queue.Queue()

class QueueManager(BaseManager):
  """
    One QueueManager to rule all network Queues
  """
  pass

QueueManager.register('get_tasks_q'  , callable=lambda:tasks_q    )
QueueManager.register('get_results_q', callable=lambda:results_q  )
QueueManager.register('get_minions_q', callable=lambda:minions_q  )
QueueManager.register('get_master_q' , callable=lambda:master_q   )

def get_qs(qm):
  """
    pass in a QueueManager, and this function returns all relevant
    queues attached to that QueueManager.
  """
  return (qm.get_tasks_q(),
          qm.get_results_q(),
          qm.get_master_q(),
          qm.get_minions_q())
```

: ../research/interp/cluster/__init__.py

```
#!/usr/bin/env python

import sys
import os

import time
import shelve
from   collections import defaultdict
from optparse import OptionParser

import logging
log = logging.getLogger("interp")

import numpy as np

from interp.cluster import QueueManager, get_qs

from progressbar import *

if __name__ == '__main__':
  parser = OptionParser(usage = "usage: %s [options] <server> <interp count>")

  parser.add_option("-l", "--last-time",
    action="store_true", dest="last", default=False,
    help="when finished, send shutdown signal to connected nodes (default: %
        default)")

  parser.add_option('-n', '--node-count',
    type="int", dest="participants", default=None,
    help="specify how many participants we should wait for (default: %default)
        ")

  parser.add_option('-p', '--port',
    type="int", dest="port", default=6666,
    help="specify the port to use on the server (default: %default)")

  parser.add_option("-o", "--order",
    type="int", dest="order", default=2,
    help="order of interpolation (default: %default)")

  parser.add_option("-e", "--extra-points",
    type="int", dest="extra", default=3,
    help="number of extra points (default: %default)")

    parser.add_option('-s', '--shelve',
      type="str", dest="shelvename", default=os.path.expanduser('~/interp.shelve
          '),
      help="shelve output file (default: %default)")

    (options, args) = parser.parse_args()
    if len(args) != 2:
      parser.print_usage()
      sys.exit(1)

    server = args[0]
    count  = int(float(args[1]))

  m = QueueManager(address=(server, options.port), authkey='asdf')
  m.connect()

    tasksq, resultsq, masterq, minionsq = get_qs(m)

    workers = []

    if not options.participants:
      print "wait on all announced participants"
      participants = 0
      while not masterq.empty():
        participants += 1
        worker = masterq.get()
        workers.append(worker)
        print "%d: %s is ready" % (participants, worker)
      if participants == 0:
        print "nobody found"
        sys.exit(1)
    else:
      participants = options.participants
      print "wait on %d participants" % participants
      for i in xrange(participants):
        worker = masterq.get()
        workers.append(worker)
        print "%d of %d: %s is ready" % (i+1, participants, worker)

    if len(set(workers)) != len(workers):
      for i in workers:
        minionsq.put("slay")
      raise Exception("duplicate workers reported")

    results = []

    widgets = ['submit jobs: ', Percentage(), ' ', Bar(), ' ', ETA()]
    pbar = ProgressBar(widgets = widgets, maxval = count)
    pbar.start()
    submit_start = time.time()
    for i in xrange(count):
      X = np.random.random((1,3))[0]
      tasksq.put((i, options.order, options.extra, X))
      pbar.update(i+1)
    submit_end = time.time()
    pbar.finish()

    for i in xrange(participants):
      print "sending worker %d start message" % (i+1,)
      minionsq.put("start")

    receive_start = time.time()
    widgets = ['interpolate: ', Percentage(), ' ', Bar(), ' ', ETA()]
    pbar = ProgressBar(widgets = widgets, maxval = count)
    pbar.start()
    for i in xrange(count):
      cur_result = resultsq.get()
      results.append(cur_result)
      pbar.update(i+1)
    receive_end = time.time()
    pbar.finish()

```

```
  submit  = submit_end - submit_start
  receive = receive_end - receive_start

  # shut down all participants
  for i in xrange(participants):
    if options.last:
      minionsq.put("teardown")

  # post processing
  stats = {}
  stats['submit'      ] = float(submit)
  stats['receive'     ] = float(receive)
  stats['count'       ] = count
  stats['participants'] = participants
  stats['extra'       ] = options.extra
  stats['order'       ] = options.order

  print "%s" % stats
  log.error("stats: %s", stats)

  tasks_accomplished_by = defaultdict(int)
  for i in results:
    tasks_accomplished_by[i[1]] += 1
  stats['tasks'] = tasks_accomplished_by

  # npresults = np.array([(i[0],i[2],i[3],i[4], i[5]) for i in results])

  n = str(time.time())
  s = shelve.open(options.shelvename)
  s[n] = {
    'stats'   : stats,
    # 'results' : npresults,
  }
  s.close()
```

: ../research/bin/master.py

```
#!/usr/bin/env python

import sys
import os
import time

from   multiprocessing.managers import BaseManager

from optparse import OptionParser
import datetime

import numpy as np

from   interp.grid.gmsh import ggrid
from   interp.tools import baker_exact_3D as exact

from interp.cluster import QueueManager, get_qs

if __name__ == '__main__':
  parser = OptionParser(usage = "usage: %s [options] <server> <gmsh file>")

  parser.add_option("-v", "--verbose",
    action="store_true", dest="verbose", default=False,
    help="verbose flag (default: %default)")

  parser.add_option('-p', '--port',
    type="int", dest="port", default=6666,
    help="specify the port to use on the server (default: %default)")

  (options, args) = parser.parse_args()

  if len(args) != 2:
    parser.print_usage()
    sys.exit(1)

  server, input_file = args

  myname = "%s-%d" % (os.uname()[1], os.getpid())
  if options.verbose:
    print "%s: started" % myname


  m = QueueManager(address=(server, options.port), authkey='asdf')
  m.connect()

  tasksq, resultsq, masterq, minionsq = get_qs(m)

  if options.verbose:
    print "%s: starting parse input file" % myname
  g = ggrid(input_file)
  g.q = np.array([exact(x) for x in g.verts])
  if options.verbose:
    print "%s: done parsing input file" % myname


  while True:
    if options.verbose:
      print "%s: letting master know that I am ready" % myname
    masterq.put(myname)

    if options.verbose:
      print "%s: waiting for master to tell me to start" % myname
    action = minionsq.get()
    if options.verbose:
      print "%s: master said go!!" % myname

    if action in ('teardown', 'slay'):
      break

    while not tasksq.empty():
      i, o, e, X = tasksq.get()
      try:
        a = g.run_baker(X, order = o, extra_points = e)
        resultsq.put((i, myname, a['qlin'], a['error'], a['final'], exact(X)))
      except Exception as e:
        print X, e
        resultsq.put((i, myname, 0.0, 0.0, 0.0, 0.0))

  if options.verbose:
    print "%s: exiting" % myname
```

: ../research/bin/minion.py

```
#!/usr/bin/env python

import sys
import time

from   interp.cluster   import QueueManager, get_qs
from   optparse         import OptionParser

if __name__ == '__main__':
  parser = OptionParser(usage = "usage: %s [options] <status|watch|add|fresult
      #|slay|clear|clearall|clearresults>")

  parser.add_option('-p', '--port',
    type="int", dest="port", default=6666,
    help="specify the port to use on the server (default: %default)")

  parser.add_option('-a', '--auth-key',
    type="str", dest="authkey", default='asdf',
    help="authkey (default: %default)")

  (options, args) = parser.parse_args()

  if len(args) == 0:
    cmd = 'status'
  else:
    cmd = args[0]

m = QueueManager(address=('', options.port), authkey=options.authkey)
m.connect()

tq,rq,mq,sq = get_qs(m)

if cmd.startswith("st"):
    print "interp queue status:"
    print " tasksq    : %d" % tq.qsize()
    print " resultsq  : %d" % rq.qsize()
    print " masterq   : %d" % mq.qsize()
    print " minionsq  : %d" % sq.qsize()

if cmd.startswith("wa"):
    if len(args) == 2:
        sleeptime = float(args[1])
    else:
        sleeptime = 1

    i = 0
    while True:
        time.sleep(sleeptime)
        if i % 20 == 0:
            print "tasksq resultsq masterq minionsq"
        print "%d %d %d %d" % \
                (tq.qsize(),
                 rq.qsize(),
                 mq.qsize(),
                 sq.qsize(),)
        i += 1

if cmd == 'add':
    for i in xrange(int(args[1])):
        mq.put('jane%d' % i)

```

```
  if cmd == 'fresult':
    for i in xrange(int(args[1])):
      rq.put('fake.%d' % i)

  if cmd == 'slay':
    if len(args) == 1:
      for i in xrange(mq.qsize()):
        print i, "killing", mq.get()
        sq.put("slay")
    elif len(args) == 2:
      for i in xrange(int(args[1])):
        print i, "killing", mq.get()
        sq.put("slay")


  if cmd == 'clear':
    for i in xrange(tq.qsize()): print tq.get()
    for i in xrange(rq.qsize()): print rq.get()

  if cmd == 'clearall':
    for i in xrange(tq.qsize()): print tq.get()
    for i in xrange(rq.qsize()): print rq.get()
    for i in xrange(mq.qsize()): print mq.get()
    for i in xrange(sq.qsize()): print sq.get()

  if cmd == 'clearresults':
    for i in xrange(rq.qsize()): print rq.get()
```

: ../research/bin/iqmgr.py

```
#!/usr/bin/env python

from    interp.cluster import QueueManager
from    optparse       import OptionParser

if __name__ == '__main__':
  parser = OptionParser(usage = "usage: %s [options] <server> <interp count>")

  parser.add_option('-p', '--port',
    type="int", dest="port", default=6666,
    help="specify the port to use on the server (default: %default)")

  parser.add_option('-a', '--auth-key',
    type="str", dest="authkey", default='asdf',
    help="authkey (default: %default)")

  (options, args) = parser.parse_args()

  m = QueueManager(address=('', options.port), authkey=options.authkey)
  s = m.get_server()
  s.serve_forever()
```

: ../research/bin/server.py

A.4 Gmsh Mesh Generation Scripts

```
a = 0.0049;
Point(0) = {0, 0, 0, a};
Point(1) = {1, 0, 0, a};
Point(2) = {1, 1, 0, a};
Point(3) = {0, 1, 0, a};
Line(1) = {0, 1};
Line(2) = {1, 2};
Line(3) = {2, 3};
Line(4) = {3, 0};
Line Loop(6) = {3, 4, 1, 2};
Plane Surface(6) = {6};
```

: ../research/gmsh/gmsh.2D.geo

```
a = 0.885;
Point(0) = {0, 0, 0, a};
Point(1) = {1, 0, 0, a};
Point(2) = {1, 1, 0, a};
Point(3) = {0, 1, 0, a};
Line(1) = {0, 1};
Line(2) = {1, 2};
Line(3) = {2, 3};
Line(4) = {3, 0};
Line Loop(6) = {3, 4, 1, 2};
Plane Surface(6) = {6};
Extrude {0, 0, 1} {
  Surface{6};
}
```

: ../research/gmsh/gmsh.3D.geo

A.5 General-purpose Utilities

```
from interp import config
import sys
sys.path.append(config['pypath'])

import bpy

import pickle
points = pickle.load(open('/tmp/points.p', 'r'))
faces  = pickle.load(open('/tmp/cells.p', 'r'))
# faces = [faces[i] for i in faces]
me = bpy.data.meshes.new('points')
me.verts.extend(points)
me.faces.extend(faces)
scn = bpy.data.scenes.active
ob  = scn.objects.new(me, 'points_obj')
```

: ../research/tools/blender/plot.py

```
#!/bin/bash

```

```
export LD_LIBRARY_PATH=/usr/mpi/fsl_openmpi_gcc-1.4.2/lib:/opt/intel/mkl/10.2.5.035/lib/em64t:/opt/intel/Compiler/11.1/072/lib/intel64:/opt/intel/Compiler/11.1/072/ipp/em64t/sharedlib:/opt/intel/Compiler/11.1/072/tbb/intel64/cc4.1.0_libc2.4_kernel2.6.16.21/lib:/opt/intel/Compiler/11.1/072/lib/intel64:/opt/intel/Compiler/11.1/072/ipp/em64t/sharedlib:/opt/intel/Compiler/11.1/072/tbb/intel64/cc4.1.0_libc2.4_kernel2.6.16.21/lib:/usr/local/cuda/lib64

PYTHON=/fslhome/smm58/research/bin/python
SCRIPT=/fslhome/smm58/src/research/bin/minion.py
OPTIONS=''
SERVER=bigmemssh.fsl.byu.edu

# approximately 1e6 tets is gmsh.3D.2.msh
INPUT=/fslhome/smm58/compute/gmsh/gmsh.3D.2.msh

$PYTHON $SCRIPT $OPTIONS $SERVER $INPUT
```

: ../research/bin/minion.sh

```
#!/bin/bash

#PBS -l procs=512,pmem=2gb,walltime=00:40:00,feature='!harpertown'
#PBS -N scalability-512
#PBS -m bea
#PBS -M stephen@mcquay.me

# ,qos=test

# for parallel submission:
/usr/bin/pbsdsh  /fslhome/smm58/src/research/bin/minion.sh

# serial:
# /fslhome/smm58/bin/slave.sh

exit 0
```

: ../research/bin/submit.sh

A.6 Unit Tests

```python
#!/usr/bin/env python

import unittest

import baker2dorder
import baker2d
import baker3d
import cubic2d
import pattern
import qhull
import quadratic2d


if __name__ == '__main__':
  tests = [
            unittest.TestLoader().loadTestsFromTestCase(baker2dorder.Test),
            unittest.TestLoader().loadTestsFromTestCase(baker2d.Test),
            unittest.TestLoader().loadTestsFromTestCase(baker3d.Test),
            unittest.TestLoader().loadTestsFromTestCase(cubic2d.Test),
            unittest.TestLoader().loadTestsFromTestCase(pattern.Test),
            unittest.TestLoader().loadTestsFromTestCase(qhull.Test),
            unittest.TestLoader().loadTestsFromTestCase(quadratic2d.Test),
  ]

  for test in tests:
    unittest.TextTestRunner(verbosity=3).run(test)
```

: ../research/test/all.py

```python
#!/usr/bin/env python

```

```
import unittest

from interp import baker
from interp.grid import grid
import numpy as np

from interp.grid import contains

def exact_func(point):
  x = point[0]
  y = point[1]
  return 0.5 + x*x + y

def calculate_error_term(self, a,b,c,d,e,f):
    B = np.array([
            self.p1[a] * self.p1[b], self.p1[c] * self.p1[d], self.p1[e] * self.
               p1[f],
            self.p2[a] * self.p2[b], self.p2[c] * self.p2[d], self.p2[e] * self.
               p2[f],
            self.p3[a] * self.p3[b], self.p3[c] * self.p3[d], self.p3[e] * self.
               p3[f],
            self.p4[a] * self.p4[b], self.p4[c] * self.p4[d], self.p4[e] * self.
               p4[f],
        ])
    B.shape = (4,3)

    A = np.dot(B.T, B)
    rhs = np.dot(B.T, self.w)
    abc = np.linalg.solve(A,rhs)

    err = \
      abc[0] * self.phis[a] * self.phis[b] + \
      abc[1] * self.phis[c] * self.phis[d] + \
      abc[2] * self.phis[e] * self.phis[f]
    return err

```

```python
class Test(unittest.TestCase):
  def setUp(self):
    self.verts = [
                 [ 2,  3], # 0
                 [ 7,  4], # 1
                 [ 4,  8], # 2
                 [ 0,  7], # 3, 1
                 [ 5,  0], # 4, 2
                 [10,  5], # 5, 3
                 [ 8,  9], # 6, 4
               ]


    self.q = [exact_func(v) for v in self.verts]

    self.g = grid(self.verts,      self.q)
    self.R = grid(self.verts[:3], self.q[:3])
    self.S = grid(self.verts[3:], self.q[3:])

    self.p1, self.ql1 = baker.qlinear(self.verts[3], self.R)
    self.p2, self.ql2 = baker.qlinear(self.verts[4], self.R)
    self.p3, self.ql3 = baker.qlinear(self.verts[5], self.R)
    self.p4, self.ql4 = baker.qlinear(self.verts[6], self.R)

    self.q1 = exact_func(self.verts[3])
    self.q2 = exact_func(self.verts[4])
    self.q3 = exact_func(self.verts[5])
    self.q4 = exact_func(self.verts[6])


    self.w = np.array([
      self.q1 - self.ql1,
      self.q2 - self.ql2,
      self.q3 - self.ql3,
      self.q4 - self.ql4,
    ])

    self.X = [4,5]

    self.g = grid(self.verts, self.q)

    self.phis, self.qlin  = baker.qlinear(self.X, self.R)
    self.exact = exact_func(self.X)
    self.answer = baker.run_baker(self.X,self.R,self.S)


  def test_R_contains_X(self):
    self.assertTrue(contains(self.X, self.R.verts))

  def test_1(self):
   a,b,c,d,e,f = (0,1,   1,2,   2,0)
    err = calculate_error_term(self, a,b,c,d,e,f)
    self.assertAlmostEqual(err, self.answer['error'])
  def test_swap_first_elements(self):
   a,b,c,d,e,f = (1,0,   1,2,   2,0)
    err = calculate_error_term(self, a,b,c,d,e,f)
    self.assertAlmostEqual(err, self.answer['error'])
  def test_swap_two_pairs(self):
   a,b,c,d,e,f = (1,2,   0,1,   2,0)
    err = calculate_error_term(self, a,b,c,d,e,f)
    self.assertAlmostEqual(err, self.answer['error'])
  def test_swap_all_pairs(self):
   a,b,c,d,e,f = (0,2,   0,1,   2,1)
    err = calculate_error_term(self, a,b,c,d,e,f)
    self.assertAlmostEqual(err, self.answer['error'])


if __name__ == '__main__':
  suite = unittest.TestLoader().loadTestsFromTestCase(Test)
  unittest.TextTestRunner(verbosity=3).run(suite)
```

: ../research/test/baker2dorder.py

```
#!/usr/bin/env python

import unittest

from  interp import baker
from  interp import grid

import numpy as np
import scipy.spatial

class Test(unittest.TestCase):
  def setUp(self):
    self.l = [[-1, 1], [-1, 0], [-1, 1], [0, -1], [0, 0], [0, 1], [1, -1], [1,
        0], [1, 1]]
    self.all_points = [
                       [ 0, 0], # 0
                       [ 1, 0], # 1
                       [ 1, 1], # 2
                       [ 0, 1], # 3
                       [ 1,-1], # 4
                       [ 0,-1], # 5
                       [-1, 1], # 6
                       [-1, 0], # 7
                       [-1,-1], # 8
                     ]
    self.q = [1, 0, 0, 0, 0, 0, 0, 0, 0]
    self.X = [0.5, 0.25]
    self.accuracy = 8

  def testImports(self):
    import numpy
    import scipy
    import interp.grid
    import interp.baker

  def testGetPhis(self):

    X = [0,0]
    r = [[-1, -1], [0, 2], [1, -1]]

    result = baker.get_phis(X, r)

    right_answer = [1/3.0, 1/3.0, 1/3.0]

    for a,b in zip(result, right_answer):
      self.assertAlmostEqual(a,b)

  def testGetPhis2(self):

    X = [0.5,0.25]
    r = [[0, 0], [1, 0], [1, 1]]

    result = baker.get_phis(X, r)

    right_answer = [0.5, 0.25, 0.25]

    for a,b in zip(result, right_answer):
      self.assertEqual(a,b)

  def testQlinear(self):
    X = [0.5, 0.25]
    r = [[0, 0], [1, 0], [1, 1]]
    q = [1, 0, 0]

    phi, result = baker.qlinear(X, grid.grid(r,q))

    right_answer = 0.5

    self.assertAlmostEqual(result, right_answer)

  def testRunBaker_1(self):
    size_of_simplex = 3
    extra_points    = 3

    R = grid.grid(self.all_points[:size_of_simplex],
                              self.q[:size_of_simplex])

    S = grid.grid(self.all_points[size_of_simplex:size_of_simplex +
        extra_points],
                              self.q[size_of_simplex:size_of_simplex +
                                  extra_points])


    answer = baker.run_baker(self.X, R, S)

    a = answer['abc'][0]
    b = answer['abc'][1]
    c = answer['abc'][2]

    self.assertEqual(sorted((a,b,c)), sorted((0,0.0,1/3.)))

  def testRunBaker_2(self):
    size_of_simplex = 3
    extra_points    = 4

    R = grid.grid(self.all_points[:size_of_simplex],
                              self.q[:size_of_simplex])

    S = grid.grid(self.all_points[size_of_simplex:size_of_simplex +
        extra_points],
                              self.q[size_of_simplex:size_of_simplex +
                                  extra_points])

    answer = baker.run_baker(self.X, R, S)

    a, b, c  = sorted(answer['abc'])
    aa,bb,cc = sorted((2/3.0, 2/3.0, 1/3.0))
```

```

        self.assertAlmostEqual(a,aa)
        self.assertAlmostEqual(b,bb)
        self.assertAlmostEqual(c,cc)

    def testRunBaker_3(self):
        size_of_simplex = 3
        extra_points    = 5

        R = grid.grid(self.all_points[:size_of_simplex],
                              self.q[:size_of_simplex])

        S = grid.grid(self.all_points[size_of_simplex:size_of_simplex +
            extra_points],
                              self.q[size_of_simplex:size_of_simplex +
                                 extra_points])

        answer = baker.run_baker(self.X, R, S)

        a = answer['abc'][0]
        b = answer['abc'][1]
        c = answer['abc'][2]

        a,b,c = sorted((a,b,c))
        aa, bb, cc = sorted((13/14., 2/7., 15/14.))

        self.assertAlmostEqual(a,aa)
        self.assertAlmostEqual(b,bb)
        self.assertAlmostEqual(c,cc)

    def testRunBaker_4(self):
        size_of_simplex = 3
        extra_points    = 6

        R = grid.grid(self.all_points[:size_of_simplex],
                              self.q[:size_of_simplex])

        S = grid.grid(self.all_points[size_of_simplex:size_of_simplex +
            extra_points],
                              self.q[size_of_simplex:size_of_simplex +
                                 extra_points])

        answer = baker.run_baker(self.X, R, S)
        a = answer['abc'][0]
        b = answer['abc'][1]
        c = answer['abc'][2]

        a,b,c = sorted((a,b,c))
        aa,bb,cc = sorted((48/53.0, 15/53.0, 54/53.0))

        self.assertAlmostEqual(a, aa)
        self.assertAlmostEqual(b, bb)
        self.assertAlmostEqual(c, cc)

if __name__ == '__main__':
    suite = unittest.TestLoader().loadTestsFromTestCase(Test)
    unittest.TextTestRunner(verbosity=3).run(suite)
```

: ../research/test/baker2d.py

```
#!/usr/bin/env python

import unittest
from interp.baker import get_phis, qlinear
from interp.grid  import grid

import numpy as np
import scipy.spatial

class Test(unittest.TestCase):
  def setUp(self):
    self.X = [0.0, 0.0, 0.0]
    self.r = [
                [0.0, 0.0, 1.0],
                [0.94280904333606508, 0.0, -0.3333333283722672],
                [-0.47140452166803232, 0.81649658244673617,
                    -0.3333333283722672],
                [-0.47140452166803298, -0.81649658244673584,
                    -0.3333333283722672],
              ]
    self.q = [0.0, 0.0, 0.0, 4]


  def testGetPhis(self):
    result       = get_phis(self.X, self.r)
    right_answer = [0.25, 0.25, 0.25, 0.25]

    for a,b in zip(result, right_answer):
      self.assertAlmostEqual(a,b)


  def testQlinear(self):
    phi, result  = qlinear(self.X, grid(self.r, self.q))
    result       = result
    right_answer = 1.0
    self.assertAlmostEqual(result, right_answer)


if __name__ == '__main__':
  suite = unittest.TestLoader().loadTestsFromTestCase(Test)
  unittest.TextTestRunner(verbosity=2).run(suite)
```

: ../research/test/baker3d.py

```
#!/usr/bin/env python

import unittest

from interp.baker import run_baker

from interp.grid import grid
from interp.grid import contains

def exact_func(X):
  x = X[0]
  y = X[0]
  return 1 + x + y

class Test(unittest.TestCase):
  def setUp(self):
    self.verts = [
                        [ 0.25, 0.40], # 0
                        [ 0.60, 0.80], # 1
                        [ 0.65, 0.28], # 2
                        [ 0.28, 0.65], # 3
                        [ 1.00, 0.75], # 4
                        [ 0.30, 0.95], # 5
                        [ 0.80, 0.50], # 6
                        [ 0.35, 0.15], # 7
                      ]
    self.q = [exact_func(p) for p in self.verts]

    self.X = [0.55, 0.45]

    self.g = grid(self.verts, self.q)
    # self.g.construct_connectivity()
    self.R = self.g.create_mesh(range(3))

    self.exact = exact_func(self.X)


  def test_R_contains_X(self):
    self.assertTrue(contains(self.X, self.R.verts))

  def test_RunBaker_1_extra_point(self, extra=1):
    S = self.g.create_mesh(range(3, 3 + extra))
    answer = run_baker(self.X, self.R, S, order=3)
    lin_err   = abs(self.exact - answer['qlin'])
    final_err = abs(self.exact - answer['final'])
    self.assertTrue(lin_err >= final_err)
  def test_RunBaker_2_extra_point(self, extra=2):
    S = self.g.create_mesh(range(3, 3 + extra))
    answer = run_baker(self.X, self.R, S, order=3)
    lin_err   = abs(self.exact - answer['qlin'])
    final_err = abs(self.exact - answer['final'])
    self.assertTrue(lin_err >= final_err)
  def test_RunBaker_3_extra_point(self, extra=3):
    S = self.g.create_mesh(range(3, 3 + extra))
    answer = run_baker(self.X, self.R, S, order=3)
    lin_err   = abs(self.exact - answer['qlin'])
    final_err = abs(self.exact - answer['final'])
    self.assertTrue(lin_err >= final_err)
  def test_RunBaker_4_extra_point(self, extra=4):
    S = self.g.create_mesh(range(3, 3 + extra))
    answer = run_baker(self.X, self.R, S, order=3)
    lin_err   = abs(self.exact - answer['qlin'])
    final_err = abs(self.exact - answer['final'])
    self.assertTrue(lin_err >= final_err)
  def test_RunBaker_5_extra_point(self, extra=5):
    S = self.g.create_mesh(range(3, 3 + extra))
    answer = run_baker(self.X, self.R, S, order=3)
    lin_err   = abs(self.exact - answer['qlin'])
    final_err = abs(self.exact - answer['final'])
    self.assertTrue(lin_err >= final_err)

if __name__ == '__main__':
  suite = unittest.TestLoader().loadTestsFromTestCase(Test)
  unittest.TextTestRunner(verbosity=3).run(suite)
```

: ../research/test/cubic2d.py

```
#!/usr/bin/env python

```

```python
import unittest
from interp.baker import pattern



class Test(unittest.TestCase):
  def setUp(self):
    pass

  def testImports(self):
    from interp.baker import pattern

  def test_baker_eq_8(self):
    b = sorted([tuple(sorted(i)) for i in ((0,1),(1,2),(2,0))])
    p = sorted(pattern(3,2))
    self.assertEqual(b,p)

  def test_baker_eq_17(self):
    b = sorted([tuple(sorted(i)) for i in ((0,1,1), (0,2,2), (1,0,0), (1,2,2),
         (2,0,0), (2,1,1), (0,1,2))])
    p = sorted(pattern(3,3))
    self.assertEqual(b,p)

  def test_baker_eq_15(self):
    b = sorted([tuple(sorted(i)) for i in (
          (0,1), (0,2), (0,3),
          (1,2), (1,3), (2,3))])

    p = sorted(pattern(4,2))

    self.assertEqual(b,p)

  def test_smcquay_(self):
    b = sorted([tuple(sorted(i)) for i in (
          (0,1,2), (1,2,3), (0,1,3), (0,2,3),
          (0,0,1), (0,1,1),
          (1,2,2), (1,1,2),
          (0,2,2), (0,0,2),
          (1,3,3), (1,1,3),
          (2,2,3), (2,3,3),
          (0,3,3), (0,0,3))])

    p = sorted(pattern(4,3))
    self.assertEqual(b,p)




if __name__ == '__main__':
  suite = unittest.TestLoader().loadTestsFromTestCase(Test)
  unittest.TextTestRunner(verbosity=3).run(suite)
```

: ../research/test/pattern.py

```
#!/usr/bin/env python

import unittest

class Test(unittest.TestCase):
   def setUp(self):
      self.l = [[-1, 1], [-1, 0], [-1, 1], [0, -1], [0, 0], [0, 1], [1, -1], [1,
            0], [1, 1]]


   def testQhull(self):
      import delaunay
      dt = delaunay.Triangulation(self.l)
      answer = [
                  [4,1,3],
                  [1,5,0],
                  [5,1,4],
                  [7,3,6],
                  [7,4,3],
              [7,5,4],
              [5,7,8],
            ]

    self.assertEqual(dt.indices, answer)

if __name__ == '__main__':
  suite = unittest.TestLoader().loadTestsFromTestCase(Test)
  unittest.TextTestRunner(verbosity=5).run(suite)
```

: ../research/test/qhull.py

```
#!/usr/bin/env python

import unittest

from interp.baker import run_baker

from interp.grid  import grid
from interp.grid  import contains

def exact_func(X):
  x = X[0]
  y = X[0]
  return 1 - x*x + y*y

class Test(unittest.TestCase):
  def setUp(self):
    self.points = [
                        [ 0.25, 0.40], # 0
                        [ 0.60, 0.80], # 1
                        [ 0.65, 0.28], # 2
                        [ 0.28, 0.65], # 3
                        [ 1.00, 0.75], # 4
                        [ 0.30, 0.95], # 5
                        [ 0.80, 0.50], # 6
                        [ 0.35, 0.15], # 7
                        ]
    self.q = [exact_func(p) for p in self.points]

    self.X = [0.25, 0.4001]
    self.X = [0.55, 0.45]

    self.g = grid(self.points, self.q)
    self.R = self.g.create_mesh(range(3))

    self.exact = exact_func(self.X)


    self.accuracy = 8

  def test_R_contains_X(self):
    self.assertTrue(contains(self.X, self.R.verts))

  def test_RunBaker_1_extra_point(self, extra=1):
    S = self.g.create_mesh(range(3, 3 + extra))
    answer = run_baker(self.X, self.R, S)
    lin_err   = abs(self.exact - answer['qlin'])
    final_err = abs(self.exact - answer['final'])

    # I expect this one to be bad:
    # self.assertTrue(lin_err >= final_err)

  def test_RunBaker_2_extra_point(self, extra=2):
    S = self.g.create_mesh(range(3, 3 + extra))
    answer = run_baker(self.X, self.R, S)
    lin_err   = abs(self.exact - answer['qlin'])
    final_err = abs(self.exact - answer['final'])
    self.assertTrue(lin_err >= final_err)
  def test_RunBaker_3_extra_point(self, extra=3):
    S = self.g.create_mesh(range(3, 3 + extra))
    answer = run_baker(self.X, self.R, S)
    lin_err   = abs(self.exact - answer['qlin'])
    final_err = abs(self.exact - answer['final'])
    self.assertTrue(lin_err >= final_err)
  def test_RunBaker_4_extra_point(self, extra=4):
    S = self.g.create_mesh(range(3, 3 + extra))
    answer = run_baker(self.X, self.R, S)
    lin_err   = abs(self.exact - answer['qlin'])
    final_err = abs(self.exact - answer['final'])
    self.assertTrue(lin_err >= final_err)
  def test_RunBaker_5_extra_point(self, extra=5):
    S = self.g.create_mesh(range(3, 3 + extra))
    answer = run_baker(self.X, self.R, S)
    lin_err   = abs(self.exact - answer['qlin'])
    final_err = abs(self.exact - answer['final'])
    self.assertTrue(lin_err >= final_err)

if __name__ == '__main__':
  suite = unittest.TestLoader().loadTestsFromTestCase(Test)
  unittest.TextTestRunner(verbosity=3).run(suite)
```

: ../research/test/quadratic2d.py