# N-th-order Accurate, Distributed Interpolation Library

Stephen M. McQuay‡*, Steven E. Gorrell‡

**Abstract**—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, yields an interpolation to an arbitrary degree of accuracy. The smbinterp module 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. The library includes modules that allow for its use in high-performance parallel computing environments. These modules were implemented using built-in Python modules to simplify deployment. This implementation was found to scale linearly to approximately 180 participating compute processes.

**Index Terms**—n-th-order accurate general interpolation, distributed calculation schemes, multiphysics simulation

## Introduction and Background

As engineers attempt to find numeric solutions to large physical problems, simulations involving multiple physical models or phenomena, known as multiphysics simulations, must be employed. This type of simulation often involves the coupling of disparate computer codes. 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 [Sha01], or the coupling of Reynolds-Averaged Navier-Stokes and Large Eddy Simulation (RANS/LES) codes in Computational Fluid Dynamics (CFD) [Med06]. 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 [Hah06]. When this is the case, and the mesh elements do not align, the engineer must perform interpolation from the upstream code to the downstream code.

Frameworks exist that perform interpolation for multiphysics simulations. In general, frameworks of this variety try to solve two problems. First, the framework should rapidly calculate the interpolation. Secondly, the interpolation should be accurate.

CHIMPS (*Coupler for High-performance Integrated Multi-Physics Simulations*) is a Fortran API (with Python bindings) that implements an efficient Distributed Alternating Digital Tree for rapid distributed data lookup [Alo06], [Hah09]. By default, CHIMPS can only provide the user with linear (second-order accurate) interpolations. While CHIMPS can provide third-order and higher accurate interpolations, it is not automatic; higher-order interpolations are only performed if the engineer supplies the CHIMPS API with higher-order terms. If this information is unavailable, then CHIMPS can only yield linear interpolations.

Another interpolation framework exists that can perform automatic higher-order interpolation. AVUSINTERP [Gal06] (*Air Vehicles Unstructured/Structured Interpolation Tool*) is a tool that provides linear and quadratic interpolations requiring only the physical values at points in a donor mesh, i.e. no a priori knowledge of higher-order terms. While this framework implements a superior interpolation scheme to the tri-linear interpolation found in CHIMPS, AVUSINTERP was not implemented in a parallel fashion, nor does it allow for the engineer to arbitrarily choose the order of the interpolation past third-order accuracy.

The research presented herein describes the development of a library that is a union of the best parts of the aforementioned tools. Namely, this research provides a library, named smbinterp, that implements the interpolation of a physical value from a collection of donor points to a destination point and performs this interpolation to an arbitrary degree of accuracy. The library can perform this interpolation in both two- and three-dimensional space. Also, the library was designed and implemented to be used in a high-performance parallel computing environment. The smbinterp library is implemented as a python module that builds upon the numpy and scipy libraries and presents an API for use in multiphysics simulation integration. The library is released under the GPL, and project is available on github [smbinterp].

## Method

The numerical method implemented in smbinterp was first proposed by Baker [Bak03]. This interpolation method comprises the adjustment of a linear interpolation by a least squares estimate of higher-order terms. The Baker interpolation of the physical value of interest (denoted $q$) to the point $\Xi$ is defined by:

$$q(\Xi) = q_{linear}(\Xi) + f(\Xi), \tag{1}$$

where $q_{linear}$ is the linear interpolation, and $f(\Xi)$ is an estimation of the higher-order error terms. The following explanation is specific to two-dimensional space; three-dimensional space is treated in [McQ11].

The participating geometry required to implement this method in two spatial dimensions is shown in figure 1. The blue points ($R$) and green points ($S$) represent points in a source mesh, and the red point $\Xi$ is the point to which an interpolation is desired. $\triangle R$ represents a simplex that surrounds the destination point $\Xi$, and $S_{1..m}$ is a collection of extra points surrounding the simplex $R$. The triangles $A_1 - A_3$ represent the areas formed by $\Xi$ and $\triangle R$.

* *Corresponding author: stephen@mcquay.me*
‡ *Brigham Young University*

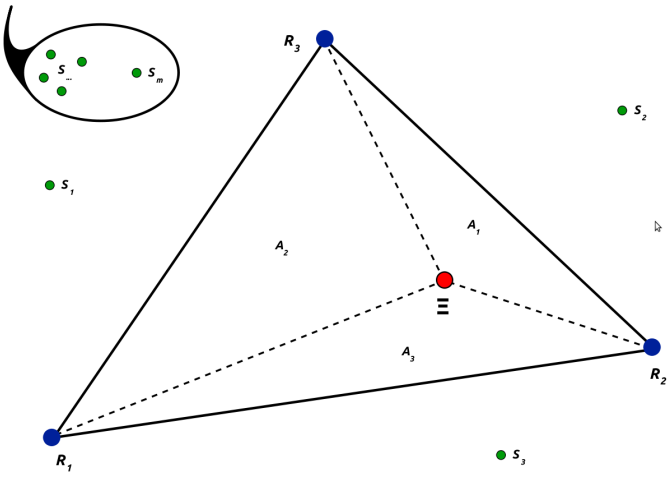


***Fig. 1:*** *Planar Simplex used in Baker's Interpolation Scheme*

Barycentric coordinates , denoted $\phi_j(\Xi)$, are used to perform the linear interpolation. In geometric terms, the barycentric coordinates 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 $\triangle R$.

The barycentric coordinates define the influence that each point in the simplex $\triangle R$ contributes to 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. If $\Xi = R_j$, the value of $q_{linear}(\Xi)$ should then be influenced entirely by the known value of $q(R_j)$. If $\Xi$ is placed in such a way as to give $\frac{A_1}{A_{total}} = \frac{A_2}{A_{total}} = \frac{A_3}{A_{total}}$, the value $q(R_j)$ at each point $R_j$ contributes equally to the calculated value of $q_{linear}(\Xi)$.

The linear interpolant, which requires the simplex $\triangle R$ and $\Xi$ as inputs, is defined as

$$q_{linear}(\triangle R, \Xi) = \sum_{j=1}^{N+1} q(R_j)\phi_j(\Xi), \tag{2}$$

where $N+1$ is the number of points in a simplex (3 in two-dimensional space, and 4 in three-dimensional space). The values of the basis functions $\phi_j(\Xi)$ is the only unknown in equation 2.

To solve for $\phi_j(\Xi)$ a system of linear equations will be defined involving the points in the simplex $R_j$, $\Xi$, and equation 2. If $q(\Xi)$ is a constant, $q_1 = q_2 = q_3 = q_{linear} = q_{constant}$, and equation 2 can be modified by dividing by $q_{constant}$, that is:

$$\phi_1 + \phi_2 + \phi_3 = 1. \tag{3}$$

Furthermore, the basis functions must be calculated so that equation 2 also interpolates geometric location of the point $\Xi$, hence

$$R_{1x}\phi_1(\Xi) + R_{2x}\phi_2(\Xi) + R_{3x}\phi_3(\Xi) = \Xi_x \tag{4}$$
$$R_{1y}\phi_1(\Xi) + R_{2y}\phi_2(\Xi) + R_{3y}\phi_3(\Xi) = \Xi_y. \tag{5}$$

The values of the basis functions $\phi_j(\Xi)$ can be found by solving the following system of linear equations involving equations 3, 4 and 5:

$$\begin{bmatrix} 1 & 1 & 1 \\ R_{1x} & R_{2x} & R_{3x} \\ R_{1y} & R_{2y} & R_{3y} \end{bmatrix} \begin{bmatrix} \phi_1(\Xi) \\ \phi_2(\Xi) \\ \phi_3(\Xi) \end{bmatrix} = \begin{bmatrix} 1 \\ \Xi_x \\ \Xi_y \end{bmatrix}, \tag{6}$$

which yields the values for $\phi_j(\Xi)$, providing a solution for equation 2.

At this point the first of two unknowns in equation 1 have been solved, however the least squares approximation of error terms $f(\Xi)$ remains unknown. 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 $R_i$ of the simplex $\triangle R$, the error term should have no influence from $\phi_i(\Xi)$, as $A_i = 0$. This condition is satisfied when expressing the error terms using the linear basis functions as

$$f(\Xi) = a\phi_1(\Xi)\phi_2(\Xi) + b\phi_2(\Xi)\phi_3(\Xi) + c\phi_3(\Xi)\phi_1(\Xi). \tag{7}$$

In equation 7 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 verticies 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 $\Xi$ that are not in the simplex $R_j$. A least squares system of equations is 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 7. Define $A$ as $(a, b, c)^T$. Applying least squares theory $a$, $b$, and $c$ are found by inverting the following $3 \times 3$ matrix:

$$B^T A = B^T w. \tag{8}$$

The matrix $B$ is defined using the identical basis function pattern as in equation 7. Denote $\phi_j(S_k)$ as the value of $\phi_j$ evaluated using equation 2 and the data point $S_k$ (in lieu of $\Xi$). The matrix $B$ in equation 8 is thus defined:

$$B = \begin{bmatrix} \phi_1(S_1)\phi_2(S_1) & \phi_2(S_1)\phi_3(S_1) & \phi_1(S_1)\phi_3(S_1) \\ \phi_1(S_2)\phi_2(S_2) & \phi_2(S_2)\phi_3(S_2) & \phi_1(S_2)\phi_3(S_2) \\ \vdots & \vdots & \vdots \\ \phi_1(S_m)\phi_2(S_m) & \phi_2(S_m)\phi_3(S_m) & \phi_1(S_m)\phi_3(S_m) \end{bmatrix}. \tag{9}$$

The value of $q(S_k)$ is known a priori (values of $q$ at each point $S_k$ in the donor mesh). The value of $q_{linear}(S_k)$ (the linear extrapolant) can also be calculated using equation 2. Define $w$ in equation 8 as

$$w = \begin{bmatrix} q(S_1) - q_{linear}(\triangle R, S_1) \\ q(S_2) - q_{linear}(\triangle R, S_3) \\ \vdots \\ q(S_m) - q_{linear}(\triangle R, S_m) \end{bmatrix}. \tag{10}$$

Equation 8 is populated with the information from each of the surrounding points in $S_k$, then the unknown $A$ can be calculated. Knowing $A$, equation 7 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 1 for $q(\Xi)$.

There exist known limitations to this least squares-based interpolation method. First a change in vertex stencil will generally yield a discontinuity in interpolation results. While this property makes this method insufficient for graphical applications, it has been shown to yield sufficiently accurate results to be used in engineering applications [Bak03], [Gal06].

Secondly, while solutions to the linear system in equation 2 are well-behaved, certain vertex configurations can lead to a singular system of equations in equation 7. These pathological vertex configurations occur when more than $n-1$ of the extra points lie on one extended edge of the simplex $\triangle R$ [Bak03]. 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.

Extension of this method into three dimensions is non-trivial, and is explained in depth in [McQ11]. 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 $\nu$ as the desired order of accuracy less one (i.e. for cubic interpolation $\nu$ is 3). As defined above, $N$ is the spatial degree. The pattern for the combinations of basis functions that are used to define $f(\Xi)$ is collection of $\nu$-th ordered combinations of $N+1$ basis functions $\phi_j$ that are unique and non-duplicate, triplicate, etc. The following code implements this pattern:

```
from itertools import product

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

The dynamic calculation of the basis function pattern in this fashion is powerful, in that it can be calculated for any arbitrary $\nu$, and for any spatial dimension (although only $N$ of 2 and 3 are dealt with herein). However, for each point $\Xi$ 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 $\nu$, 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 smbinterp that only calculates the pattern if it has not been previously calculated. This concept is known as memoization, and is implemented using the following function wrapper:

```
from functools import wraps

def memoize(f):
  cache = {}
  @wraps(f)
  def memf(simplex_size, nu):
    x = (simplex_size, nu)
    if x not in cache:
      cache[x] = f(simplex_size, nu)
    return cache[x]
  return memf
```

Baker's method gives a reasonable interpolation solution for a general cloud of points. However, the method suggested by Baker for the vertex selection algorithm for the terms $\triangle R$ and $S_k$ consists of simply selecting the points nearest $\Xi$. While this is the most general point selection algorithm, it can lead to the aforementioned pathological vertex configurations. This configuration is prevalent when the source mesh is composed of a regular grid of vertices, and must be addressed if the method is to yield a good interpolation.

Furthermore a mesh may have been designed to capture the gradient information, and therefore the mesh topology should be respected. Simply selecting the closest points to $\Xi$ would yield inferior results. By selecting the more topologically (according to the mesh) adjacent points the information intended to be captured in the mesh's design will be preserved.

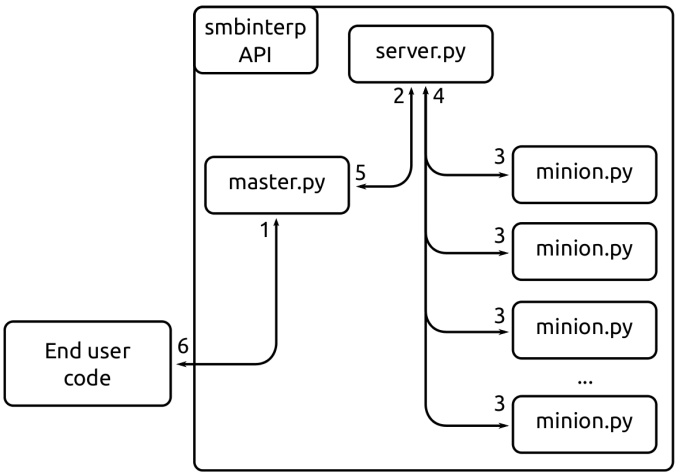


***Fig. 2:*** *Flowchart of the Parallelization Architecture*

A plugin architecture was implemented in smbinterp which yields the requisite flexibility needed to avoid the pathological grid configurations and gives the engineers complete control over the point selection algorithms. The base class for all grid objects that desire to use the interpolation methods is defined as follows:

```
class grid(object):
  def __init__(self, verts, q):
    self.verts = np.array(verts)
    self.tree = KDTree(self.verts)

    self.q = np.array(q)

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

  def get_containing_simplex(self, Xi):
    # ...
    return simplex

  def get_simplex_and_nearest_points(self,
                          Xi, extra_points = 3):
    # ...
    return simplex, extra_points
```

The cells and cells_for_verts data structures are used when searching for a containing simplex. The structures are populated with connectivity information before a round of interpolations. The method employed in the default implementation for the location of the containing simplex in an upstream mesh is straight forward: first the spatial tree structure is used to find the location of the nearest vertex to the point of interest, then the cells are recursively visited in topologically adjacent order and tested for inclusion of the point $\Xi$.

The selection of the extra points $S_k$ is also implemented in the base grid class. The default algorithm simply queries the kdtree structure for $(N+1)+m$ points and discards the points that are already in the simplex $\triangle R$.

Plugins are defined as classes that inherit from the base grid object, and that implement the requisite functionality to populate the cells and cells_for_vert data structures. If either of the default simplex and vertex selection methods do not provide the desired functionality they could be overridden in the derived class to provide a more tuned $\triangle R$ and $S_k$ selection algorithms. This gives engineers complete control over point selection and makes the interpolation library mesh agnostic.

A parallel mechanism for calculating $q(\Xi)$ was implemented

in smbinterp. As is illustrated in figure 2, a stream of requested interpolations are presented to a queuing mechanism that then distributes the task of calculating the interpolations to a set of minions.

The server.py application implements the four queues required to implement this method: a queue for tasks to be performed, a queue for results, and two queues 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 2. The master.py script is responsible for orchestrating the submission of interpolations and events associated with starting and stopping a set of interpolations. Each of the minions has access to the entire domain and are responsible for performing the interpolations requested by the end user.

The crux of the solution lies in providing the minions with a steady stream of work, and a pipeline into which the resultant interpolations can be returned. The mechanism developed in smbinterp uses built-in Python modules to minimize the deployment expense. The multiprocessing module provides a manager class which facilitates the access of general objects to authenticated participants over a network. The built-in Queue objects, which implement a multi-producer, multi-consumer first-in-first-out queue, are presented to the minions and masters using the functionality in the manager class.

### Results and Discussion of Results

The root mean square (RMS) of the errors was used to determine the accuracy of the smbinterp module. A continuous function whose values varied smoothly in the test domain was required to calculate the error; the following equation was used:

$$q(x,y) = \left(\sin(x\pi)\cos(y\pi)\right)^2. \tag{11}$$

A plot of this function is found in figure 3. Each error $\varepsilon_i$ was calculated as the difference between the actual value (from equation 11) and calculated interpolations (at each point in the destination domain using smbinterp), or $\varepsilon_i(\Xi) = q_{exact}(\Xi) - q_{calculated}(\Xi)$.

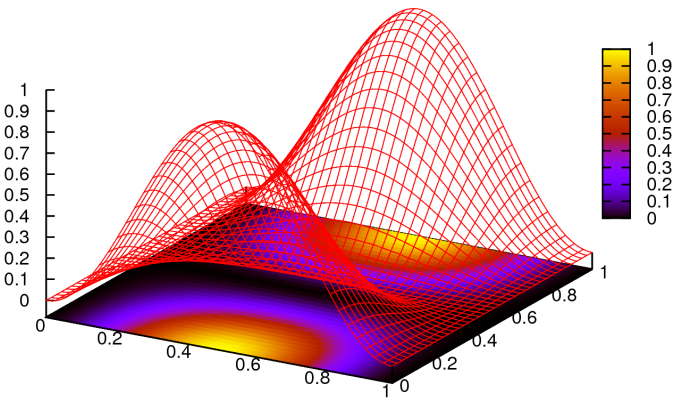


***Fig. 3:*** *Plot of Equation 11*

A mesh resolution study was performed to determine how the RMS of error varied with mesh density. The source mesh was generated using gmsh, and the lowest-resolution mesh is shown in figure 4. The results of this study are show in figure 5. A collection of 1000 random points were used as the destination for interpolation.

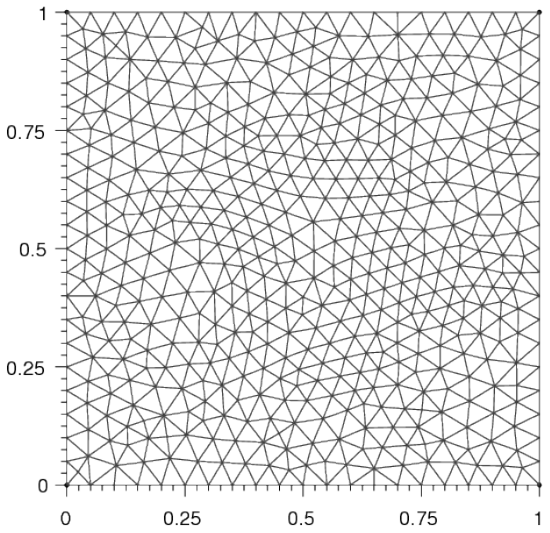


***Fig. 4:*** *Lowest-resolution test mesh*

Figure 5 plots the relationship between mesh spacing and RMS of error of all interpolations in the collection of destination vertexes. The x-axis represents the spacing between the regular mesh elements. The y-axis was calculated by performing interpolation from each resolution of mesh to a static collection of random points. 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 the requested degree of accuracy. As an example, the collection of points for $\nu$ of 2 should be third-order accurate, and should follow a line with slope of 3; this is closely demonstrated in the plots.

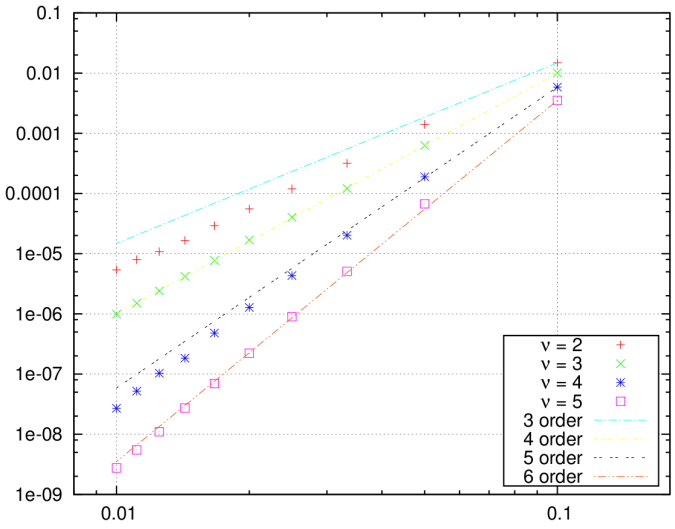


***Fig. 5:*** *RMS of Error vs. Mesh Spacing*

Figure 5 shows the results of the resolution study for the two-dimensional test case meshes. The three dimensional test case meshes yielded similar results and are presented in [McQ11]. As the meshes were refined the RMS of error decreased. The fourth- and sixth-order results ($\nu$ 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.

As mesh element size decreased, the RMS of error decreased

as well. The RMS of error for the highest $\nu$ decreased more than that of the lowest $\nu$. The RMS of error of the most coarse mesh (far right) ranges within a single order of magnitude, whereas the RMS of errors at the most fine spacing (far left) span four orders of magnitude. The results 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 figure 5 is indicative of the order of 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 $\nu$ 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 (i.e. less error). Also, the fine meshes experience a more significant decrease in RMS of error than the coarse meshes while increasing the order of approximation, $\nu$. 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 optimally accurate results.

The parallel algorithm employed by smbinterp was found to scale quasi-linearly to approximately 180 participating minion.py processes. 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 [WSU], or $S_p = \frac{T_1}{T_p}$. A parallel algorithm is considered to have ideal speedup if $S_p = p$.

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 $E_p = \frac{T_p}{p}$. The efficiency of an algorithm ranges from 0 to 1, and is shown for smbinterp in figure 6.

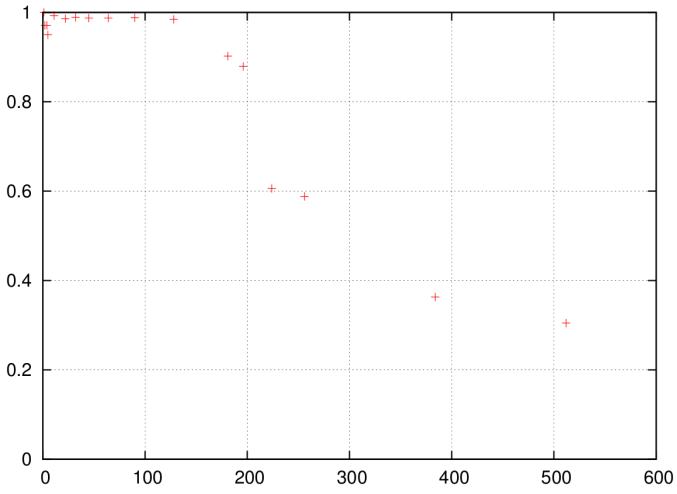


***Fig. 6:*** *Efficiency ($E_p$) of the Parallel Algorithm*

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. If an algorithm does not have an efficiency of 1, it is usually indicative of communication overhead or bottlenecks of some form. It was observed that the cpu utilization of the server.py script increased linearly up to 181 minions (CPU utilization of 200%), but then did not increase past that point. The implementation of the server.py script represents the bottleneck of this implementation.

## Conclusions

The smbinterp module was developed to provide a high-performance interpolation library for use in multiphysics simulations. The smbinterp module 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 interpolation is fifth-order accurate, etc.

The smbinterp module 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.

The smbinterp module 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. It is suggested to replace the queuing mechanism with a more high-performance queuing library (e.g. ØMQ) and a more advanced participant partitioning scheme to allow the library to scale past this point.

## Acknowledgments

The authors thank Marshall Galbraith for his friendly and crucial assistance which helped clarify the implementation of the numerical method used herein. The authors are especially grateful to have performed this research during a time when information is so freely shared and readily available; they are indebted to all of the contributors to the Python and Scipy projects. The authors 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.

## References

[Sha01] S. Shankaran et al., *A Multi-Code-Coupling Interface for Combustor/Turbomachinery Simulations*, AIAA Paper 2001-974, 39th AIAA Aerospace Sciences Meeting and Exhibit January 8–11, 2001.

[Med06] G. Medic et al., *Integrated RANS/LES computations of turbulent flow through a turbofan jet engine*, Center for Turbulence Research Annual Research Briefs, 2006, pp. 275-285.

[Hah06] S. Hahn et al., *Coupled High-Fidelity URANS Simulation for Helicopter Applications}*, Center for Turbulence Research Annual Research Briefs, 2006, pp 263-274.

[Alo06] J. Alonso et al., *CHIMPS: A High-Performance Scalable Module for Multi-Physics Simulations*, AIAA Paper 2006-5274, 42nd AIAA/ASME/SAE/ASEE Joint Propulsion Conference & Exhibit, Sacramento, CA, July 2006.

[Hah09] S. Hahn et al., *Extension of CHIMPS for unstructured overset simulation and higher-order interpolation*, AIAA Paper 2009-3999, 19th AIAA Computational Fluid Dynamics, San Antonio, Texas, June 22-25, 2009

[Gal06] M. Galbraith, J. Miller. *Development and Application of a General Interpolation Algorithm*, AIAA Paper 2006-3854, 24th AIAA Applied Aerodynamics Conference, San Francisco, California, June 5-8, 2006

[Bak03] Baker, T. *Interpolation from a Cloud of Points*, in 12th International Meshing Roundtable, Santa Fe, NM, 2003, pp. 55-63.

[McQ11] S. M. McQuay, *SMBInterp: an Nth-Order Accurate, Distributed Interpolation Library,* M.S. thesis, Mech. Eng., Brigham Young University, Provo, UT, 2011.

[WSU] http://en.wikipedia.org/wiki/Speedup

[smbinterp] https://github.com/smcquay/smbinterp