

Efficient basis derivatives evaluations for high-order discontinuous Galerkin schemes

Micaiah Smith-Pierce^{*1} and Aliraza Mazaheri^{†2}

¹*Daniel Guggenheim School of Aerospace Engineering, Georgia Institute of Techonology, Atlanta, GA 30332*

²*Aerothermodynamics Branch, Research Directorate, NASA Langley Research Center, Hampton, VA 23681*

Computational methods of evaluating a basis of Lagrange polynomials are developed for the purpose of implementing efficient discontinuous Galerkin conservation laws solvers. Special attention is payed to the computation of higher-order partial derivatives which may be required for certain applications. Two different approaches are considered; one involves hardcoding explicit one-line formulae into the source code in order to make it as simple as possible, whereas the other uses algorithms designed to minimize the asymptotic order of growth with respect to the order of the scheme. Timing experiments show that either approach can perform well if implemented effectively, and identify advantages and disadvantages of each.

Nomenclature

ξ^i	The i th interpolation node (i.e., Lagrange point) in local element coordinates
$\mathbf{p}^{(i)}$	The position index function
$\mathbf{x}^i$	The i th interpolation node (i.e., Lagrange point) in Cartesian coordinates
δ_{mn}	The Kronecker delta tensor
η	Index of a monomial term
Ω_h	Computational domain
Φ	The vector of basis functions
ϕ^i	Shape function associated with the i th degree of freedom
$\xi_1, \dots, \xi_{N_{coord}}$	Element-specific coordinates (as defined in Table 2)
C	The matrix of coefficients to represent the ϕ in the monomial basis
$d\Phi$	The matrix of basis derivatives
$D^{\mathbf{r}}$	The partial differential operator $\frac{\partial^{r_1+\dots+r_{N_{dim}}}}{\partial x_1^{r_1} \dots \partial x_{N_{dim}}^{r_{N_{dim}}}}$ for the multiindex $\mathbf{r} = (r_1, r_2, \dots, r_{N_{dim}})$
i	Index of degrees of freedom
I_m	The $m \times m$ Identity matrix
K	Degree of the polynomial basis
n	Element's index
N_e	Number of elements
N_{coord}	The number of ξ -coordinates
N_{dim}	The number of spatial dimensions
N_{dof}	Number of degrees of freedom of u_h within each element
u_h	Approximate solution
u_n^i	Coefficient of ϕ^i to represent u_h in the n th element (I_n)
X	The vector of monomials $\mathbf{x}^{\mathbf{q}}$

Boldface characters will be used to denote vectors, and their components will be identified with subscripts; e.g., $\mathbf{x} = (x_1, x_2, \dots, x_{N_{dim}})$. In addition, an element name written as a subscript to a variable will indicate

^{*}Corresponding author, Undergraduate student, micaiah@gatech.edu

[†]alireza.mazaheri@nasa.gov

that variable is evaluated in the context of the named reference element (and may have a different value in a different element type). Furthermore, because some indices are placed in superscripts, when a variable is raised to a power it will be wrapped in round brackets (“”).

Finally, the following convention will be observed. If $\mathbf{a} = (a_1, \dots, a_m)$ and $\mathbf{b} = (b_1, \dots, b_m)$, then

$$\mathbf{a}^{\mathbf{b}} := (a_1)^{b_1} (a_2)^{b_2} \dots (a_m)^{b_m}.$$

I. Introduction

Discontinuous Galerkin (DG) methods have shown the potential to enhance the speed and accuracy of solutions to conservation laws on both structured and unstructured grids (Zhou, Li, and Shu,¹ Antoniadis et al.²). DG is a type of finite element method, where the computational domain, Ω_h , is divided into many smaller domains known as “elements”. Within each element, the approximate solution, u_h , is represented as

$$u_h = \sum_{i=1}^{N_{dof}} u_n^i \phi^i, \quad (1)$$

where the coefficients u_n^i depend only on time, and ϕ^i are polynomials, known as basis functions that are not time dependent. The constant $N_{dof} \in \mathbb{N}$ is referred to as the number of “degrees of freedom” of the approximation. Although much effort has been expended discovering the fastest and most accurate manner in which to advance the coefficients u_n^i , there is little published material describing the preferable means to numerically evaluate the basis functions ϕ^i . To address that deficiency, this study develops methods to compute the basis functions and documents their performance.

Although many factors affect the overall efficiency of a DG code, the speed of basis evaluation may have a larger impact, for example, in evaluating the smoothness indicators for high-order weighted essentially nonoscillatory schemes,³ which requires all the nonzero partial derivatives of the basis functions. In general, the number of required derivatives could far exceed that of the actual basis functions. So, the ability to evaluate these derivatives efficiently may be important in order to minimize the computational cost.

The objective is to compute the following as efficiently as possible; These quantities must be computed:

- ϕ^i for $1 \leq i \leq N_{dof}$,
- $D^{\mathbf{r}} \phi^i$ for $1 \leq i \leq N_{dof}$ and any multiindex $\mathbf{r} = (r_1, r_2, \dots, r_{N_{dim}})$ such that the resulting expression does not identically vanish.

Here, N_{dim} is the number of spatial dimensions, and $D^{\mathbf{r}}$ is the partial differential operator $\frac{\partial^{r_1 + \dots + r_{N_{dim}}}}{\partial x_1^{r_1} \dots \partial x_{N_{dim}}^{r_{N_{dim}}}}$. Explicitly describing the appropriate range of values for the multiindex $\mathbf{r}$ is not entirely trivial; this is explained in Section II. In this study, the basis functions, ϕ , are chosen to be the Lagrange interpolating polynomials (see Section II), but the outcome is also relevant for other basis functions.

With the exception of Pyramids, the standard elements (i.e., Line, Triangle, Quadrilateral, Tetrahedral, Hexahedral, and Prism) are only considered in this study. Pyramids are excluded because there is no basis consisting purely of polynomials defined on pyramid elements. Instead, rational functions must be used,⁴ and optimizing their numerical evaluation is an area for further research.

Two approaches to solve the stated objective are considered; “explicit” and “abstract” approaches. In the explicit approach, the finite element basis and their derivatives are expressed directly into the source, while both the finite element basis and their derivatives are evaluated on the fly in the “abstract” approach. These two approaches are compared by measuring the corresponding execution times.

Despite the scarcity of published material on basis and derivative evaluation, Berutt and Trefethen⁵ describe an efficient method for univariate interpolation by Lagrange polynomials, along with first and second derivatives. Although these results are related to our objective, we develop more general methods which apply to multidimensional elements and higher derivatives. In addition, Hesthaven and Warburton⁶ provide code for evaluating Lagrange polynomials on simplices, but do not discuss higher-order derivatives nor do they discuss computational efficiency.

This paper is organized as follows. Section II describes the definition of the Lagrange basis for various elements, followed by the explicit procedure of computing the basis and their derivatives in Section III. Section IV outlines details of the abstract implementation for on the fly computations of the basis and their

derivatives. The performance of these two approaches for various polynomial orders are presented in Section V. Finally, some general conclusions are given in Section VI.

II. Definition of the basis

The goal of this study is to develop a fast method for computing finite element basis functions and their derivatives. The functions are used to form a solution approximation to a partial differential equation (PDE), while their derivatives are used in flux integrations and/or defining the smoothness of the polynomials. In this study, the Lagrange polynomials are used but the outcome is applicable to other polynomial bases.

Here, the reference elements are defined with a vertex at the origin and the rest at different corners of the unit cube as illustrated in Fig. 1. The vertices are given in Table 1.

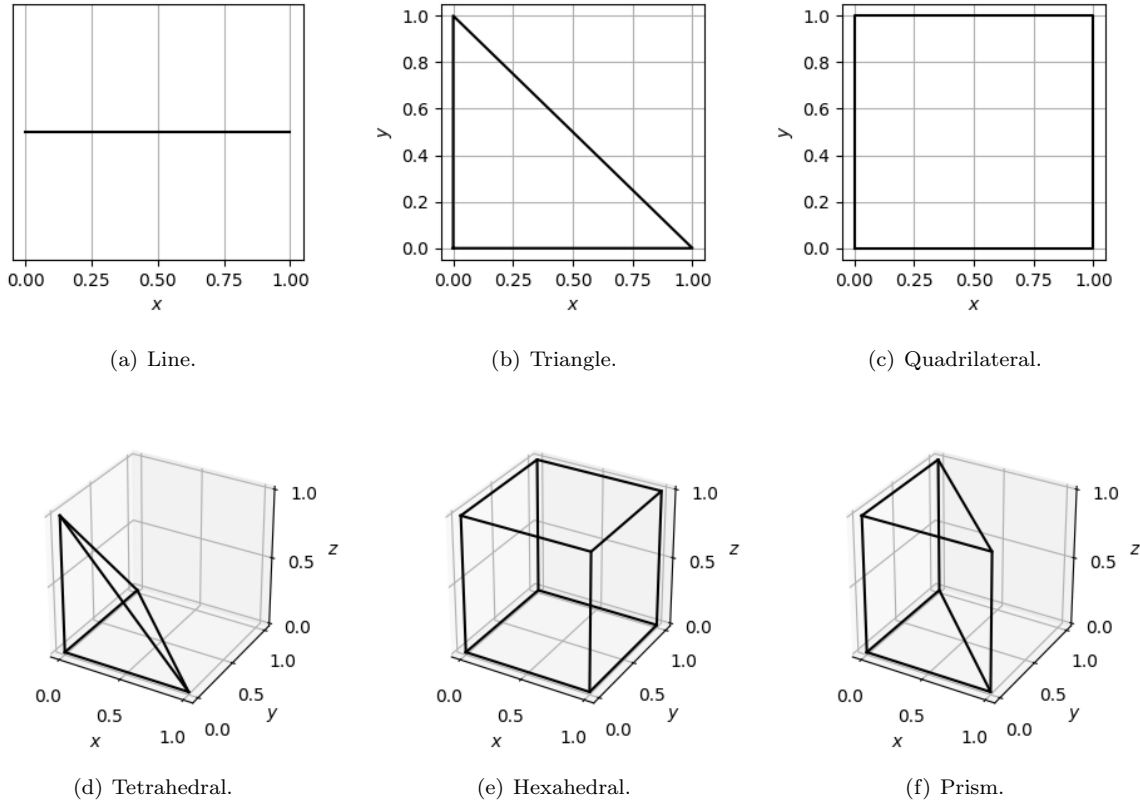


Figure 1. Reference elements.

Table 1. Vertices of the reference elements in Cartesian coordinates.

Element	Vertices coordinates $(x_1, x_2, x_3) = (x, y, z)$							
	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7
Line	(0, 0, 0)	(1, 0, 0)						
Triangle	(0, 0, 0)	(1, 0, 0)	(0, 1, 0)					
Quadrilateral	(0, 0, 0)	(1, 0, 0)	(1, 1, 0)	(0, 1, 0)				
Tetrahedral	(0, 0, 0)	(1, 0, 0)	(0, 1, 0)	(0, 0, 1)				
Hexahedral	(0, 0, 0)	(1, 0, 0)	(1, 1, 0)	(0, 1, 0)	(0, 0, 1)	(1, 0, 1)	(1, 1, 1)	(0, 1, 1)
Prism	(0, 0, 0)	(1, 0, 0)	(0, 1, 0)	(0, 0, 1)	(1, 0, 1)	(0, 1, 1)		

The bases are defined by employing local coordinates for each element. The standard Cartesian coordinates are used for line, quadrilateral, and hexahedral elements, while for simplices (triangles and tetrahedra),

barycentric coordinates are employed for convenience. For prisms, barycentric coordinates are used to represent the xy -position while the z -position is defined with a Cartesian coordinate.

For completeness, we describe the barycentric coordinates as used in our study. Consider a point Q inside a simplex element, with sides numbered 1, 2, and 3 (for triangle) and 1, 2, 3, and 4 (for tetrahedral); see Fig. 2. The barycentric coordinates ξ_1 , ξ_2 , and ξ_3 in the trinagle are then defined as the ratio of the area of

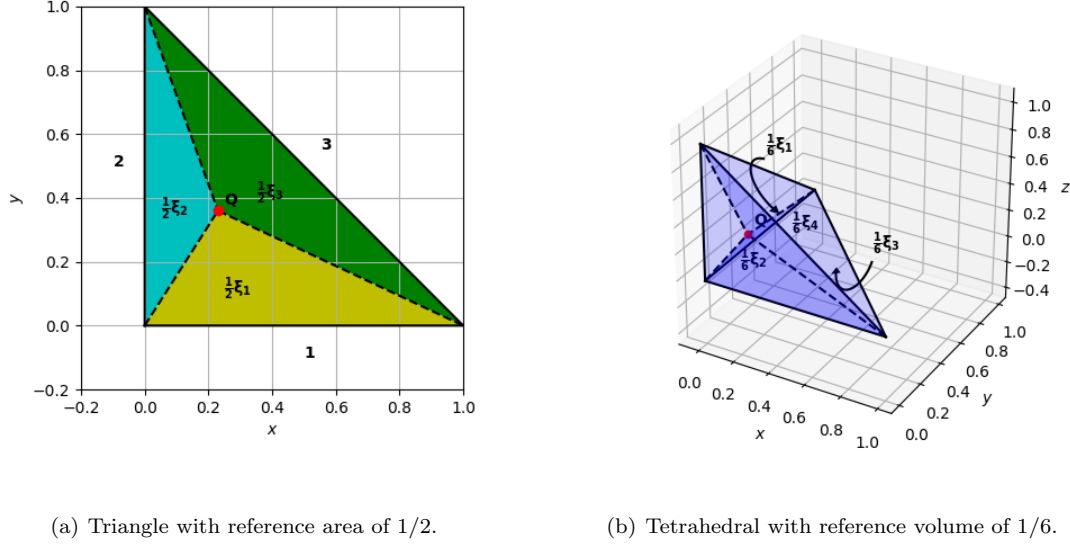


Figure 2. Barycentric coordinates.

the smaller triangles bounded by sides 1, 2, and 3 to the area of the original triangle. Similarly, barycentric coordinates for a Tetrahedral can be defined as the volume of each smaller Tetrahedral bounded by each respective face, divided by the volume of the entire Tetrahedral. Note that the barycentric coordinates as defined on multidimensional reference elements are subject to the following simple identities:

$$\xi_d = x_d, \quad d = 1, \dots, N_{dim} - 1,$$

$$\xi_{N_{dim}} = 1 - \sum_{d=1}^{N_{dim}} x_d,$$

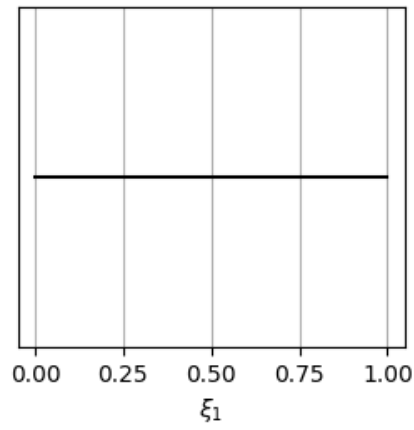
where $\mathbf{x} = (x_1, x_2, x_3) = (x, y, z)$, and d refers to the element's dimension. Note that the notion of barycentric coordinates is different from the unrelated concept of the barycentric "formula" for Lagrange polynomials.⁵

Table 2 presents the coordinates (barycentric and/or Cartesian) for all the standard elements considered in this study. These coordinates are further illustrated in Fig. 3.

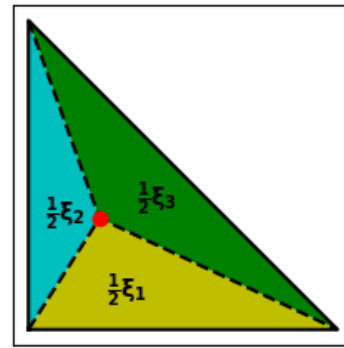
Table 2. Definition of ξ -coordinates.

Element	Coordinate definitions				Number of coordinates
	ξ_1	ξ_2	ξ_3	ξ_4	N_c
Line	x	—	—	—	1
Triangle	x	y	$1 - x - y$	—	3
Quadrilateral	x	y	—	—	2
Tetrahedral	x	y	z	$1 - x - y - z$	4
Hexahedral	x	y	z	—	3
Prism	x	y	$1 - x - y$	z	4

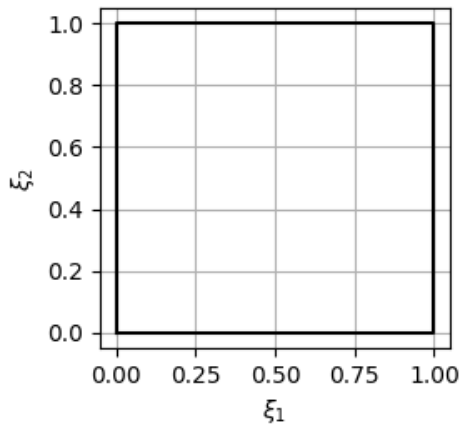
The finite element basis can now be defined in terms of reference elements coordinates. The basis used in this study consists of Lagrange polynomials, which were originally invented for the purpose of interpolating



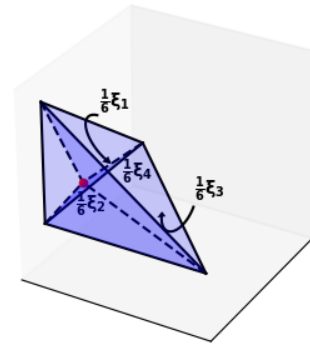
(a) Line.



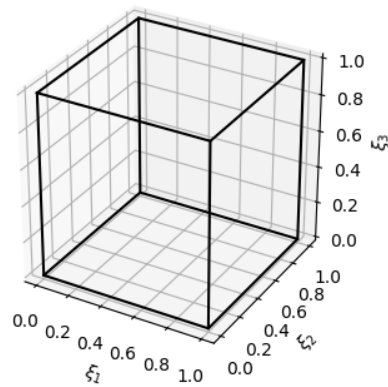
(b) Triangle.



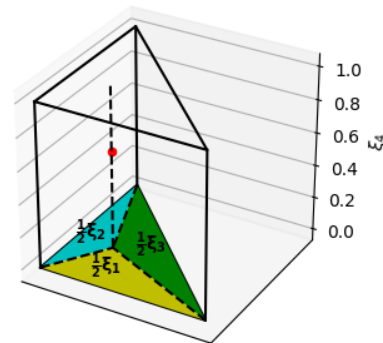
(c) Quadrilateral.



(d) Tetrahedral.



(e) Hexahedral.



(f) Prism.

Figure 3. Visualization of the ξ -coordinates for each reference element.

between discrete data points.⁷ Lagrange polynomials, by definition, are each equal to 1 at one point within the element and 0 at certain other points. All of these points are collectively referred to as Lagrange points (aka nodes). In other words, the following identity holds for a set of nodes $\{\xi^i : 1 \leq i \leq N_{dof}\}$:

$$\phi^i(\xi^j) = \delta_{ij}. \quad (2)$$

For a complete basis to be formed, there must be exactly N_{dof} nodes. Figure 4 shows the Lagrange points for the Lagrange polynomial, P_K , of order three ($K = 3$) for standard elements. A set of Lagrange basis

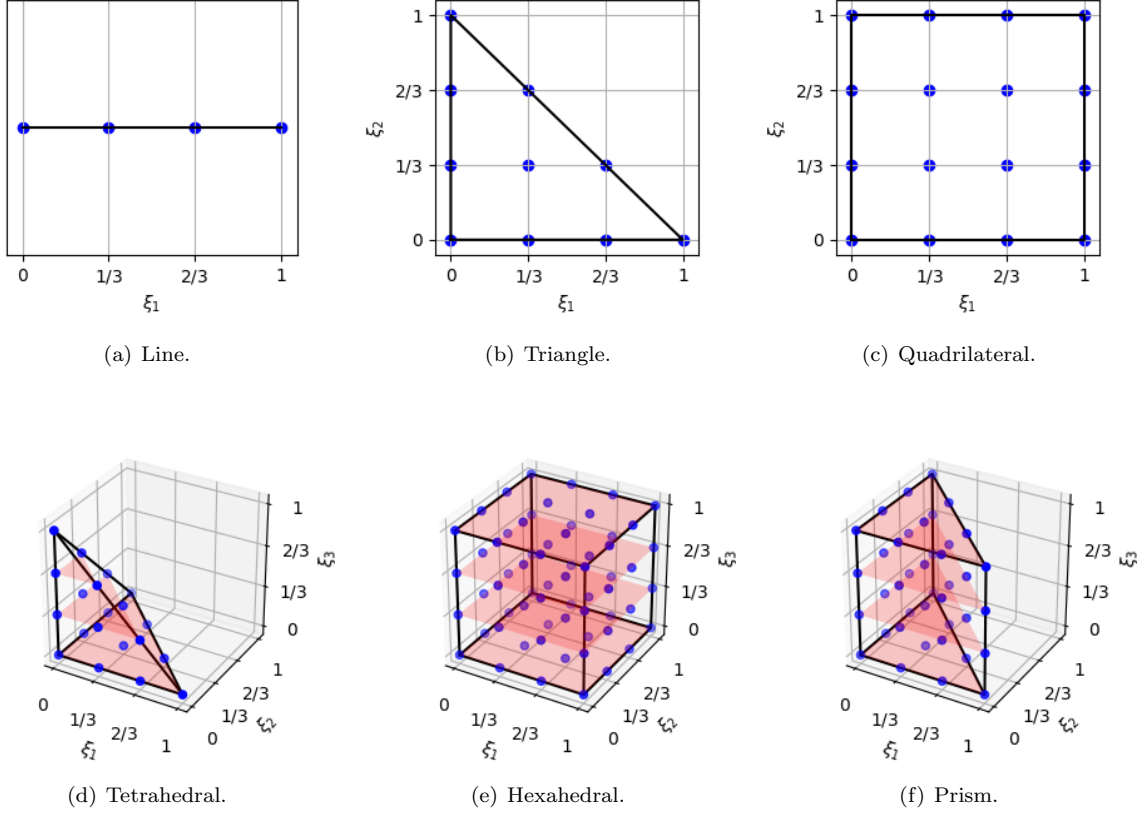


Figure 4. Nodes for each element where $K = 3$ (when interpreting the coordinate axes, recall that the first N_{dim} barycentric coordinates are always equal to the Cartesian coordinates).

polynomials for the third-order Lagrange polynomial on line and triangle is shown in Fig. 5.

In order to construct the Lagrange basis, a set of nodes must be defined. The nodes were chosen to be a set of points, which are evenly distributed in each dimension, with a spacing of $\frac{1}{K}$. To state this precisely,

$$\xi^i = \frac{1}{K} \mathbf{p}(i) = \frac{1}{K} (p_1(i), \dots, p_{N_{coord}}(i)), \quad (3)$$

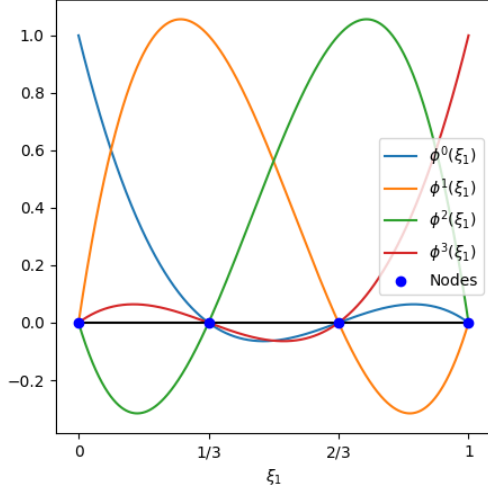
for some function

$$\mathbf{p} : \{1, \dots, N_{dof}\} \rightarrow \mathbb{N}^{N_{coord}},$$

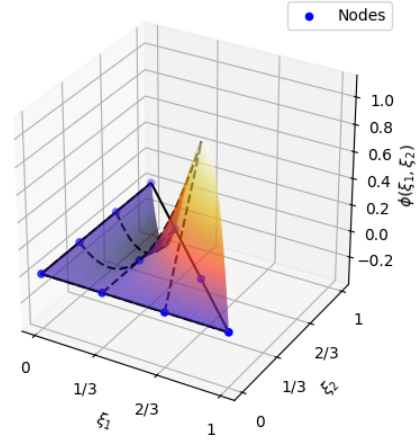
such that $\mathbf{p}$ is one-to-one, and

$$p_d(i) \leq K \quad : \quad 1 \leq d \leq N_{coord}.$$

The function $\mathbf{p}$, which will be referred to as the “position index” function, uniquely maps the index of a node to a vector of integers not greater than the polynomial order. The actual definition of $\mathbf{p}$ is arbitrary, provided that the mapping is one to one; i.e., it maps the entire set of indices to the entire set of nodes.



(a) A set of Lagrange basis on Line.



(b) A single Lagrange polynomial for a Triangle

Figure 5. Lagrange polynomial of order three.

Now that the nodes have been defined, N_{dof} , which is equal to both the number of nodes and the number of basis functions (see Table 3), can be computed. If an alternative distribution of nodes is desired, the results presented in this paper for line, quadrilateral, and hexahedral elements can easily be generalized by substituting the new node coordinate values in place of $\mathbf{p}(i)/K$. By contrast, this cannot be done for simplex elements because the expressions for ϕ given in Eq. (5) will be in violation of the property given in Eq. (2).

Table 3. Number of degrees of freedom (N_{DOF}) of the interpolating polynomials for various element types. Here K is the order of interpolating polynomial, and N_{dim} is the element dimension.

Line	Triangle or Tetrahedral	Quadrilateral or Hexahedral	Prism
$K + 1$	$(K + N_{dim})! / (K N_{dim}!)$	$(K + 1)^{N_{dim}}$	$(K + 1)^2 (K + 2) / 2$

Since the reference elements, coordinates, and nodes have been defined, it is now possible to state the expressions for the basis functions. Essentially, they are formed by multiplying factors of

$$\frac{\xi_d - m/K}{p_d(i)/K - m/K} \quad : \quad 1 \leq d \leq N_{coord},$$

for some integer m . Because this expression becomes unity at ξ^i , and vanishes at any node, where p_d evaluates to n , the interpolation property, Eq. (2), can be satisfied.

The value ϕ for each element type will now be stated. Note that the name of an element type may be written as a subscript to indicate that a quantity must be evaluated in the context of that reference element.

For line elements, the expression is

$$\phi_{line}^i(\xi_1) = \prod_{\substack{m=0 \\ m \neq p_1(i)}}^K \frac{\xi_1 - m/K}{p_1(i)/K - m/K}. \quad (4)$$

In this case, $p_1(i) := i - 1$ is a valid definition for p . To simplify the notation in the following equations, and without a loss of generality, the same results may be recovered by using $\phi_{line}^{p_1^{-1}(i)}$ in place of ϕ_{line}^{i+1} in the following expressions.

For simplices (triangles and tetrahedra), the expression is:

$$\phi_{simplex, N_{coord}}^i(\xi_1, \dots, \xi_{N_{coord}}) = \prod_{d=1}^{N_{coord}} \prod_{m=0}^{p_d(i)-1} \frac{\xi_d - m/K}{p_d(i)/K - m/K}. \quad (5)$$

For example, we have $\phi_{triangle} = \phi_{simplex,3}$ since a triangle has 3 ξ -coordinates. This form is equivalent to that discussed by De Marchi, Cecchi, and Morandi.⁸

For “bricks” (quadrilaterals and hexahedra), we have:

$$\phi_{brick, N_{coord}}^i(\xi_1, \dots, \xi_{N_{coord}}) = \prod_{d=1}^{N_{coord}} \phi_{line}^{p_d(i)+1}(\xi_d). \quad (6)$$

That is, the basis is simply a tensor product of that for line elements. It may seem as though this expression could also be used for simplex and prism elements, as it would still satisfy the interpolation property, Eq. (2). However, this cannot be done because the polynomials ϕ must form a complete basis for the space of polynomials in which they exist. So, when the expressions are expanded into monomial form in xyz -coordinates, the maximum number of monomial terms must be equal to N_{dof} . However, as Table 4 shows (see discussion below), ϕ_{brick} polynomials contain too many terms to be applied to simplex or prism elements.

The form for prisms is similar to that for bricks, but slightly more involved:

$$\phi_{prism}^i(\xi_1, \xi_2, \xi_3, \xi_4) = \phi_{triangle}^j(\xi_1, \xi_2, \xi_3) \phi_{line}^{p_d(i)+1}(\xi_4), \quad (7)$$

where j is defined implicitly by

$$(p_d)_{triangle}(j) = (p_d)_{prism}(i) \quad : \quad 1 \leq d \leq 3 = N_{C,triangle}.$$

This may appear confusing, as the new variable j was introduced spontaneously to link the position indices for prisms to those for triangles. However, the idea is still quite simple. We merely compute Lagrange polynomials for triangles based on the horizontal (xy) position and polynomials for lines based on the vertical (z) position, and take the product.

Now that the basis has been defined, its derivatives will be considered. It was not necessary to develop actual expressions for the derivatives because in the methods presented in this paper, at least some part of the differentiation logic was always left to the computer programs themselves. However, it is important to establish the set of derivatives to be computed.

In Section I, it was mentioned that all of the partial derivatives, which are not identically equal to zero, must be obtained. Consider the monomial representation of any Lagrange polynomial; i.e.,

$$\phi = \sum_{\eta=1}^{N_{mon}} (\mathbf{x})^{\mathbf{q}^\eta}, \quad (8)$$

for some multiindices $\mathbf{q}^\eta = (q_1^\eta, \dots, q_{N_{dim}}^\eta)$, where N_{mon} indicates the maximum number of monomials in the expansion, and

$$(\mathbf{x})^{\mathbf{q}^\eta} := \prod_{d=1}^{N_{dim}} (x_d)^{q_d^\eta}.$$

This representation is obtained by substituting the definitions given in Table 2 into Eqs. (4)–(7), and then expanding the expressions algebraically. By counting the number of factors in which x , y , and z will appear in Eqs. (4)–(7), bounds can be established on the values of $\mathbf{q}$, which are shown in Table 4. If the number of possible combinations $(q_1^\eta, \dots, q_{N_{dim}}^\eta)$, which satisfy that the bounds are counted, it can be verified that $N_{mon} = N_{dof}$, which is a necessary property for the basis to be valid.

Table 4. Bounds on the powers of x , y , and z in the monomial representation of the Lagrange basis for different element types. Here D represents the number of spatial dimensions of the element, and $1 \leq d \leq N_{dim}$ is an index that is used to represent which coordinate (x , y , or z) the power q_d^η belongs to.

Line	Triangle or Tetrahedral	Quadrilateral or Hexahedral	Prism
$q_1^\eta \leq K$	$\sum_{d=1}^{N_{dim}} q_d^\eta \leq K$	$q_d^\eta \leq K$	$(q_1^\eta + q_2^\eta), q_3^\eta \leq K$

Elementary calculus indicates that for some multiindex $\mathbf{r}$, we have

$$D^{\mathbf{r}}(\mathbf{x})^{\mathbf{q}^\eta} = 0,$$

unless

$$r_d \leq q_d^\eta \quad : \quad 1 \leq d \leq N_{dim}.$$

Thus, for the derivatives that must be computed, $\mathbf{r}$ also satisfies the bounds specified in Table 4. The complete set of derivatives that must be evaluated is summarized in Table 5.

Table 5. The derivatives to be computed. Here K is the polynomial order, and N_{dim} is the number of spatial dimensions.

Derivative	Bounds on $\mathbf{r}$
$D^{\mathbf{r}} \phi_{line}$	$r_1 \leq K$
$D^{\mathbf{r}} \phi_{simplex, N_{dim}}$	$\sum_{d=1}^{N_{dim}} r_d \leq K$
$D^{\mathbf{r}} \phi_{brick, N_{dim}}$	$r_d \leq K \quad : \quad 1 \leq d \leq N_{dim}$
$D^{\mathbf{r}} \phi_{prism}$	$(r_1 + r_2), r_3 \leq K$

III. Explicit approach

The closed-form nature of Eqs. (4)–(7) allowed them to be hardcoded directly into the source code. Consequently, the time required to evaluate them was simply that required to perform the floating point arithmetics, which is expected to be fast for low-order cases. We refer to this approach as “explicit” because Eqs. (4)–(7) appear explicitly in the code (as opposed to the abstract approach where they are the final product of algorithms).

In this method, the source code (in C++) was generated by a preprocessor program written in Python, which uses the Sympy library,⁹ to construct the products. One change was made to the expressions. Namely, the denominators, which contain only the constant terms, were precomputed by the Python program and expressed in the C++ output as floating-point literals. For example, the source code for a line element with $K = 3$ was essentially the following, but expressed in C++ syntax:

$$\begin{aligned} \phi_{line}^1 &= -4.5(x - 1/3)(x - 2/3)(x - 1), \\ \phi_{line}^2 &= 13.5x(x - 2/3)(x - 1), \\ \phi_{line}^3 &= -13.5x(x - 1/3)(x - 1), \\ \phi_{line}^4 &= 4.5x(x - 1/3)(x - 2/3). \end{aligned} \tag{9}$$

In order to generate the derivative expressions, Sympy’s automatic symbolic differentiation is used, with the assistance of the following identity:

$$\frac{d^r}{d\xi^r} \prod_{m \in S} (\xi - m/K) = r! \sum_{s \subseteq S: |s|=|S|-r} \left(\prod_{m \in s} (\xi - m/K) \right), \tag{10}$$

where S is some set of integers (e.g., $\{1, \dots, K\} \setminus \{i\}$, as in Eq. (4)). Identity (10) is derived by considering an arbitrary number of applications of the product rule.

Since one of the goals of DG research is to develop high-order-accurate schemes, the asymptotic time required to evaluate the basis as $K \rightarrow \infty$ is of interest. By evaluating the number of factors in Eqs. (4)–(7), it may be seen that evaluating each ϕ^i requires $O(K)$ operations. Therefore, evaluating the entire basis requires $O(N_{dof} K) = O(N_{dof}^{(N_{dim}+1)/N_{dim}})$ operations. By contrast, from Eq. (10), it can be seen that computing the derivatives involves finding all possible combinations of factors from the sets, which have anywhere from 1 to K elements. From that fact, it is shown that the asymptotic order of growth is larger than any polynomial.

IV. Abstract approach

In the explicit approach, although the expression for each basis function is in simplest terms, some operations became redundant when the entire basis was computed at once. For example, when computing Eq. (9), the product $(x - 1/3)(x - 2/3)$ is computed twice; once for ϕ_{line}^1 , and again for ϕ_{line}^3 . It is possible to eliminate this redundancy, but the resulting expressions will no longer be in a closed form. Rather, the basis

evaluation must be implemented as an algorithm, which involves programming abstraction. Indeed, using the algorithms presented in the following, the time required to compute the basis is reduced to $O(N_{dof})$, and the derivatives to $O(N_{dof}^3)$:

A. Basis evaluation

The following algorithm is implemented in C++, and used to compute ϕ_{line} .

Algorithm 1 Compute ϕ_{line}^i for all $1 \leq i \leq N_{dof} = K + 1$

```

minInd  $\leftarrow$  1
minDiff  $\leftarrow$  x
for  $1 \leq p \leq K$  do
    if  $x - p/K < minDiff$  then
        minInd  $\leftarrow$  p
        minDiff  $\leftarrow$   $x - p/K$ 
    end if
end for
prodminInd  $\leftarrow$  1
for  $0 \leq p \leq K$  do
    if  $p \neq minInd$  then
        prodminInd  $\leftarrow$   $(x - p/K)prod^{minInd}$ 
    end if
end for
for  $0 \leq p \leq K$  do
    if  $p \neq minInd$  then
        prodp  $\leftarrow$   $\left(\frac{minDiff}{x - p/K}\right)prod^{minInd}$ 
    end if
end for
for  $1 \leq i \leq N_{dof}$  do
     $\phi_{line}^i \leftarrow \frac{prod^{i-1}}{w^{(i-1)}}$ 
end for

```

Here,

$$w^p := \prod_{\substack{0 \leq m \leq K \\ m \neq p}} (p/K - m/K), \quad (11)$$

is precomputed, and tabulated during an initialization stage. The minimum-finding step at the start prevents division by a vanishingly small quantity when x is very close to one of the nodes.

The numerical result of Algorithm 1 for $\phi_{line}^{minInd+1}$ is equivalent to directly evaluating Eq. (4). Otherwise, for $i \neq minInd + 1$, it is equivalent to the “modified Lagrange” formula described by Higham;¹⁰ i.e.,

$$\phi_{line}^i = \frac{\prod_{m=0}^K (x - m/K)}{w^{p_1(i)} (p_1(i)/K - m/K)}. \quad (12)$$

where

$$w^p = \prod_{m=0}^p (p/K - m/K), \quad (13)$$

which is tabulated during an initialization phase.

Although both Algorithm 1 and the barycentric formula⁵ are $O(N_{dof})$, Algorithm 1 has two advantages. First, according to Higham, the barycentric formula has greater roundoff error than the modified Lagrange formula in certain cases. Second, Algorithm 1 involves primarily multiplication operations, as opposed to the barycentric formula, which involves mainly divisions, and thus, slower on some processors.

Algorithm 2 is developed for simplex elements. The end result is numerically equivalent to directly evaluating Eq. (5). Yet, the most time consuming step is the final loop, so the entire algorithm is still $O(N_{dof})$.

Algorithm 2 Compute $\{\phi_{simplex, N_{coord}}^i : 1 \leq i \leq N_{dof}\}$, where N_{coord} is the number of ξ -coordinates. Recall that for these elements, $N_{coord} = N_{dim} + 1$.

```

 $\xi_{N_{coord}} \leftarrow 1$ 
for  $1 \leq d \leq N_{dim}$  do
     $\xi_d \leftarrow x_d$ 
     $\xi_{N_{coord}} \leftarrow \xi_{N_{coord}} - x_d$ 
end for
for  $1 \leq d \leq N_{coord}$  do
     $prod_d^0 \leftarrow \xi_d$ 
    for  $1 \leq m < K$  do
         $prod_d^m \leftarrow (\xi_d - m/K)prod_d^{m-1}$ 
    end for
end for
for  $1 \leq i \leq N_{dof}$  do
     $\phi_{simplex, N_{coord}}^i \leftarrow 1$ 
    for  $1 \leq d \leq N_{coord}$  do
         $\phi_{simplex, N_{coord}}^i \leftarrow \frac{prod_d^{p_d(i)}}{w^{p_d(i)}} \phi_{simplex, N_{coord}}^i$ 
    end for
end for

```

Since all other element types can be written as tensorial products of ϕ_{line} and $\phi_{simplex,3}$, it was not necessary to develop any additional algorithms. To compute the basis for these elements, first a complete basis for a line and/or triangle is evaluated. Then, each basis function for the desired element is formed as a product of already computed quantities. Since evaluating the basis for a lower-dimensional element is faster than for a higher-dimensional one, the asymptotic efficiency is determined by the final multiplication step, which is $O(N_{dof})$.

B. Derivatives

As effective as Algorithms 1 and 2 may be at eliminating redundancy, such generalizations are found to be very difficult for derivatives. Instead, the basis is represented in terms of monomials, since these can be easily differentiated with the formula:

$$D^{\mathbf{r}}(\mathbf{x})^{\mathbf{q}^\eta} = \prod_{d=1}^{N_{dim}} \left((x_d)^{q_d^\eta - r_d} \prod_{\rho=0}^{r_d-1} (q_d^\eta - \rho) \right), \quad (14)$$

if $\mathbf{r}$ and $\mathbf{q}$ obey the bounds in Tables 4 and 5.

In order to represent the basis in terms of monomials, we require an $N_{dof} \times N_{dof}$ coefficient matrix C such that

$$\Phi = CX, \quad (15)$$

where

$$\Phi = \begin{bmatrix} \phi^1 \\ \vdots \\ \phi^{N_{dof}} \end{bmatrix} \quad ; \quad X = \begin{bmatrix} (\mathbf{x})^{\mathbf{q}^1} \\ \vdots \\ (\mathbf{x})^{\mathbf{q}^{N_{dof}}} \end{bmatrix}.$$

The values of the sequence $\mathbf{q}^1, \dots, \mathbf{q}^{N_{dof}}$ are arbitrary as long as every possible combination of components, which satisfy the appropriate bounds is included exactly once. In order to find C , the first step is to construct a Vandermonde-like matrix:

$$V = \begin{bmatrix} (\mathbf{x}^1)^{\mathbf{q}^1} & \dots & (\mathbf{x}^{N_{dof}})^{\mathbf{q}^1} \\ \vdots & & \vdots \\ (\mathbf{x}^1)^{\mathbf{q}^{N_{dof}}} & \dots & (\mathbf{x}^{N_{dof}})^{\mathbf{q}^{N_{dof}}} \end{bmatrix}. \quad (16)$$

The interpolation property, Eq. (2), requires that $CV = I_{N_{dof}}$. So, the coefficient matrix is given by

$$C = V^{-1}. \quad (17)$$

Although the matrix inversion is expensive, it is only necessary to be performed once, regardless of how many times the basis derivatives are evaluated.

Once C has been computed, the derivatives are given by

$$d\Phi = C(dX), \quad (18)$$

where

$$d\Phi = \begin{bmatrix} D^{\mathbf{q}^1} \phi^1 & \dots & D^{\mathbf{q}^1} \phi^{N_{dof}} \\ \vdots & & \vdots \\ D^{\mathbf{q}^{N_{dof}}} \phi^1 & \dots & D^{\mathbf{q}^{N_{dof}}} \phi^{N_{dof}} \end{bmatrix} ; \quad dX = \begin{bmatrix} D^{\mathbf{q}^1}(\mathbf{x})^{\mathbf{q}^1} & \dots & D^{\mathbf{q}^1}(\mathbf{x})^{\mathbf{q}^{N_{dof}}} \\ \vdots & & \vdots \\ D^{\mathbf{q}^{N_{dof}}}(\mathbf{x})^{\mathbf{q}^1} & \dots & D^{\mathbf{q}^{N_{dof}}}(\mathbf{x})^{\mathbf{q}^{N_{dof}}} \end{bmatrix}.$$

Now, if the values $\{(x_d)^r : 1 \leq r \leq K, 1 \leq d \leq N_{dim}\}$ and $\{\prod_{\rho=0}^{r-1}(q-\rho) : 1 \leq q \leq K, 1 \leq r \leq q\}$ are computed first and tabulated, then dX can be computed in $O(N_{dof})$ time using Eq. (14). For that reason, the most expensive operation, which is performed for every derivative evaluation, is the matrix product CdX , which theoretically is $O(N_{dof}^3)$. That said, the Eigen¹¹ library, which is used in this study, is actually capable of computing the product in slightly less time. In addition, the derivatives for prism and “brick” elements can also be obtained in less time because they can be evaluated as a tensor product of derivatives for line and triangle elements.

C. Implementation

The “abstract” approach was implemented in two different ways. First, it was implemented by defining C++ template classes which use object-oriented paradigms to concisely express the above algorithms. This shall be referred to as the “abstract template” implementation. It was also implemented with a Python script that automatically hardcodes the algorithms into C++ files. Although far less concise than the template approach, this method is nearly as simple as the explicit approach while maintaining the asymptotic complexity of the abstract algorithms. This second implementation shall be referred to as the “abstract hardcoded” implementation.

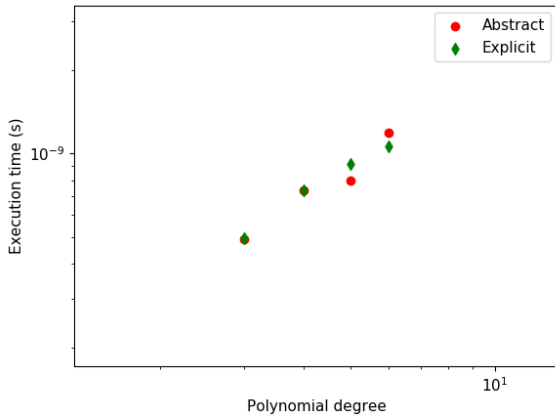
There is one difference between the algorithms used by the template and hardcoded approaches. Namely, Algorithm 1 cannot be used in a hardcoded implementation because it involves runtime branching. Instead, Algorithm 2 was used, as a line can also be considered to be a 1-dimensional simplex! Although this algorithm involves roughly twice as many operations, it is still $O(N_{dof})$.

V. Numerical experiments

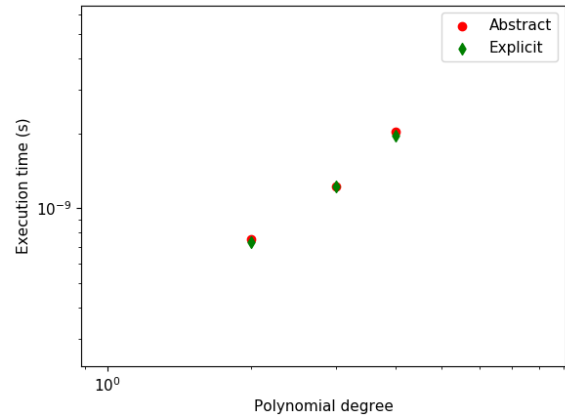
Asymptotic analysis is useful for designing algorithms, but in practice the implementation has a critical effect on performance. So, the execution speeds of all approaches were measured empirically and compared. Each implementation was evaluated 10^6 times in a loop, and the average execution time for each iteration was calculated from the total elapsed time. All implementations were compiled by the GNU C++ compiler with “-O3” optimization and executed in serial.

Figures 6 and 7 compares the execution speed of the explicit approach to the hardcoded abstract approach. These comparisons were limited to low polynomial degree because the explicit code took an inordinately long time to compile for higher degree. For high-order, 3-dimensional elements, the explicit code can involve thousands of lines of complicated arithmetic. The hardcoded abstract approach seems to take less time to compile, although this has not been measured precisely. The abstract templated approach compiles nearly instantaneously by comparison, but its execution time is several orders of magnitude longer. Thus, it is possible to test much higher degree polynomials with the templated approach, but it is not worth comparing its execution speed at modest degree.

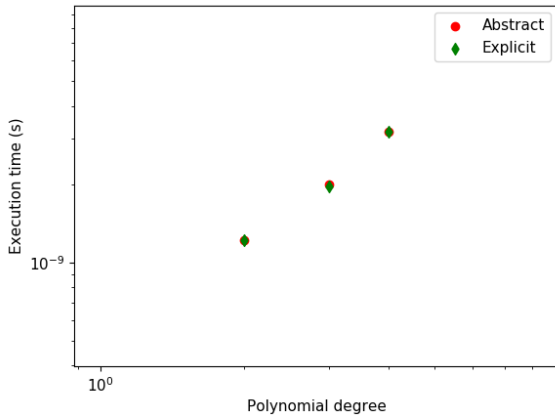
The asymptotic analysis of the previous sections applies only to the floating point operations. It is interesting to check whether these results actually predict the execution time of the program. To that end, Figures 8 - 11 display the timing data for the abstract methods along with power-function curve-fits. The powers obtained thus, which correspond to the asymptotic order of growth, are reported in Table 6. These do not always agree with the predictions in Section IV. Although they are technically more desirable, they suggest that some other factor besides the number of floating point operations may be limiting the execution



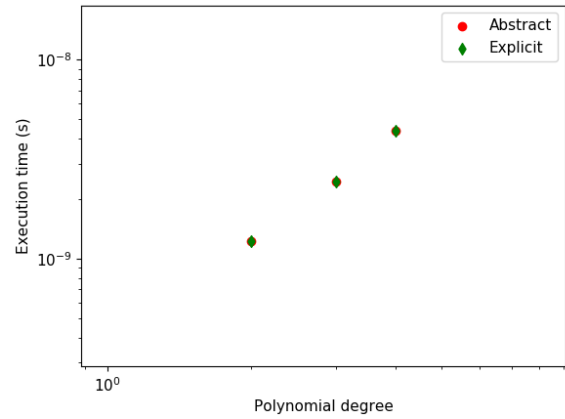
(a) Line.



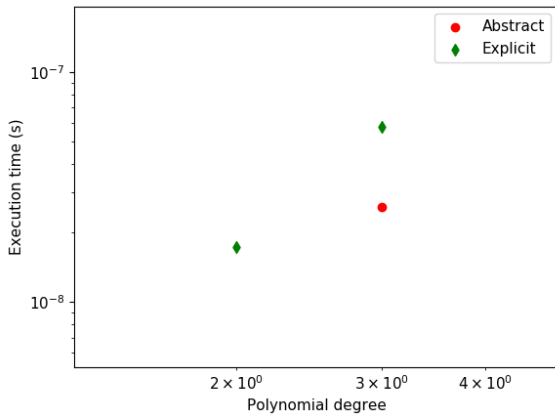
(b) Triangle.



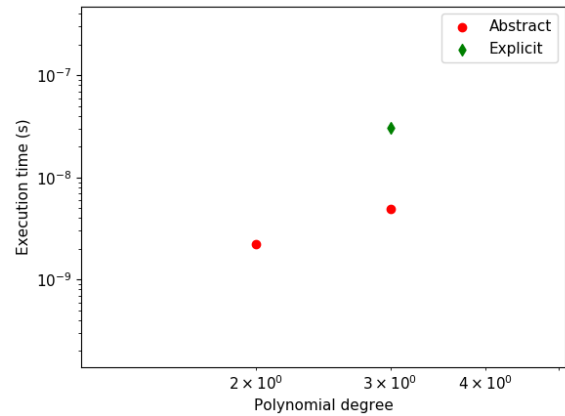
(c) Quadrangle.



(d) Tetrahedral.

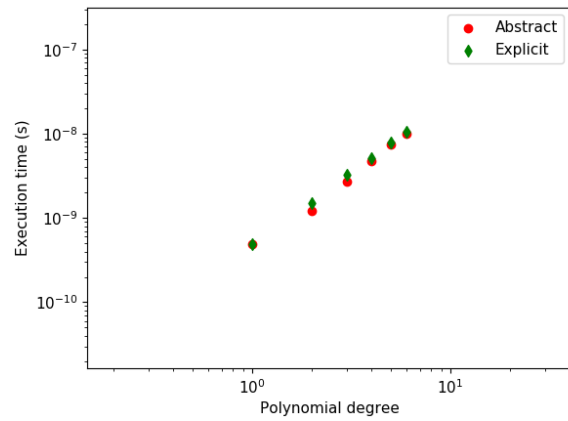


(e) Hexahedral.

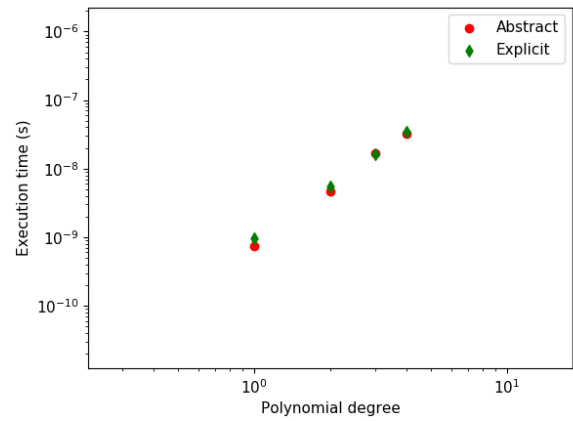


(f) Prism.

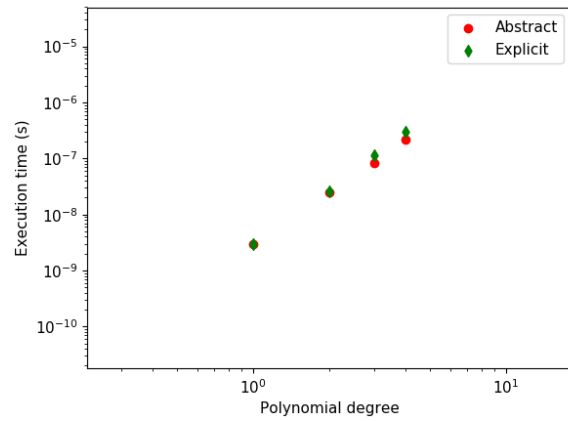
Figure 6. Execution time to compute the complete set of basis functions, without derivatives.



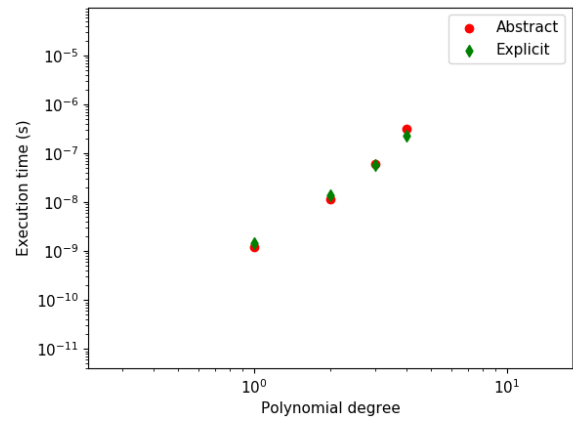
(a) Line.



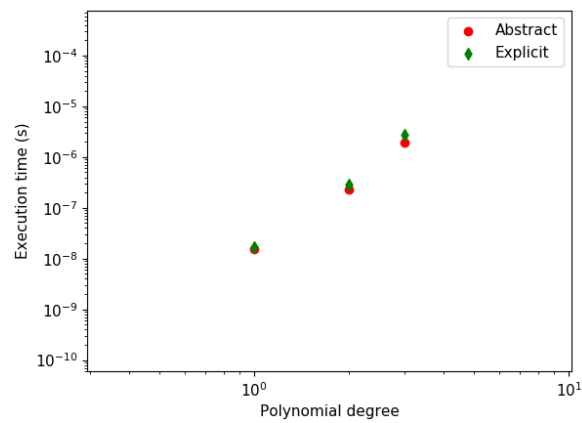
(b) Triangle.



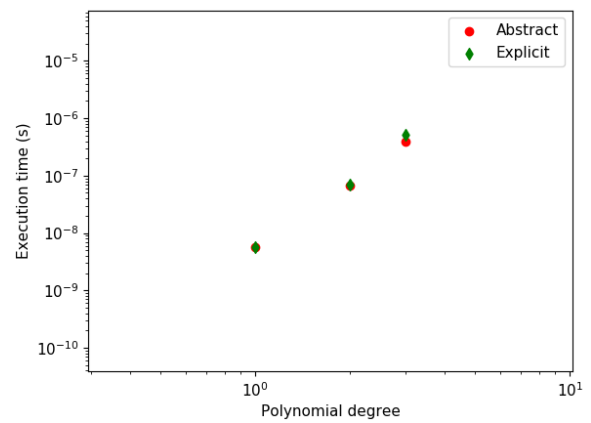
(c) Quadrangle.



(d) Tetrahedral.



(e) Hexahedral.



(f) Prism.

Figure 7. Execution time to compute the complete set of basis derivatives.

speed. This hypothesis is supported by the similarity in the execution times of the explicit and abstract hardcoded implementations, which involve different numbers of operations.

Although the performance of the explicit and abstract hardcoded implementations is generally similar, the abstract approach is modestly faster for quadrangle, hexahedral, and prism elements. The explicit approach is slightly faster for some cases for Triangle and Tetrahedral, although the significance of this minor disparity is questionable. Therefore, differences in compilation time may be the most important factor in distinguishing these two approaches.

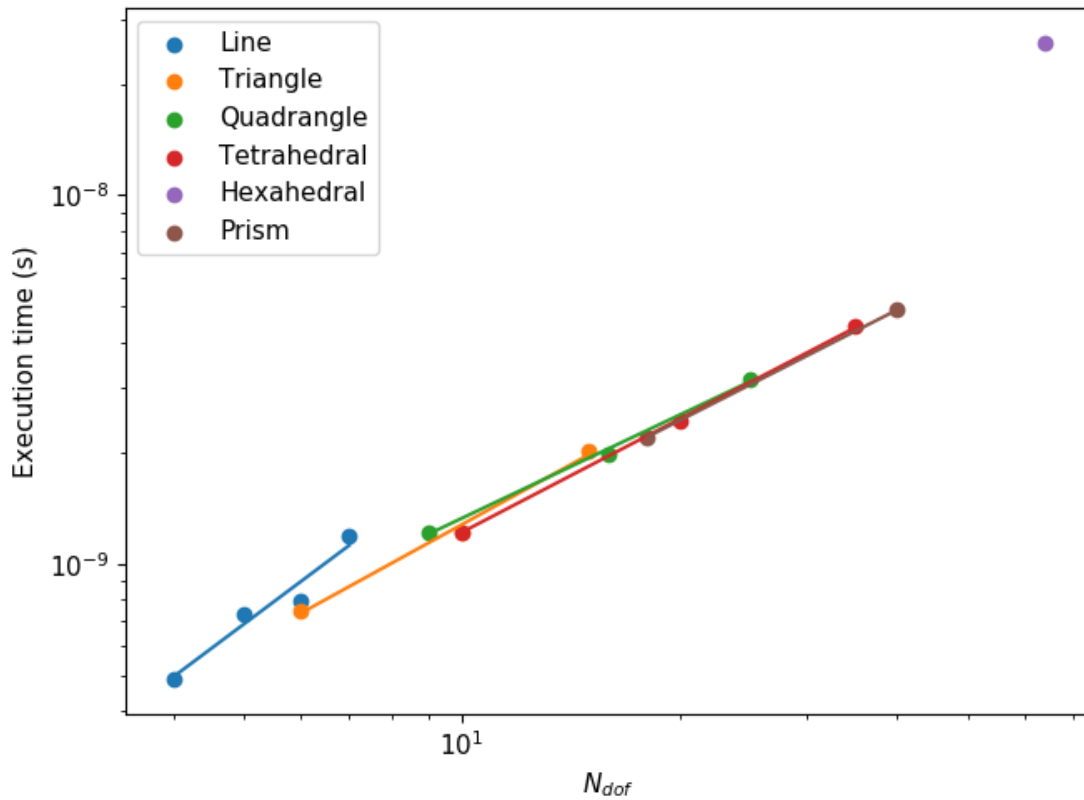


Figure 8. Time required to compute the complete basis (without derivatives) by the abstract hardcoded approach.

Table 6. Order of growth n such that $\approx O(N_{dof}^n)$ time is required to compute the specified quantity. Hex is not shown due to insufficient data.

Element	Basis	Derivatives
Line	1.47	1.47
Triangle	1.08	1.08
Quadrangle	0.93	0.93
Tetrahedral	1.02	1.02
Prism	1.00	1.00

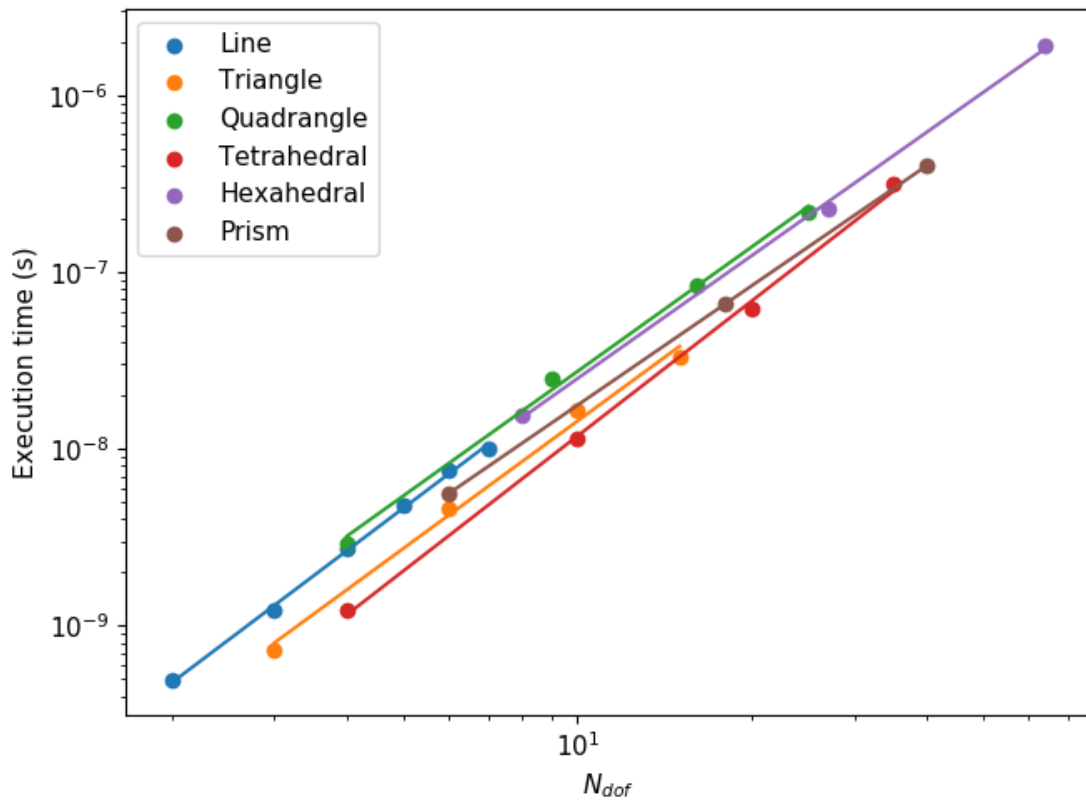


Figure 9. Time required to compute all derivatives by the abstract hardcoded approach.

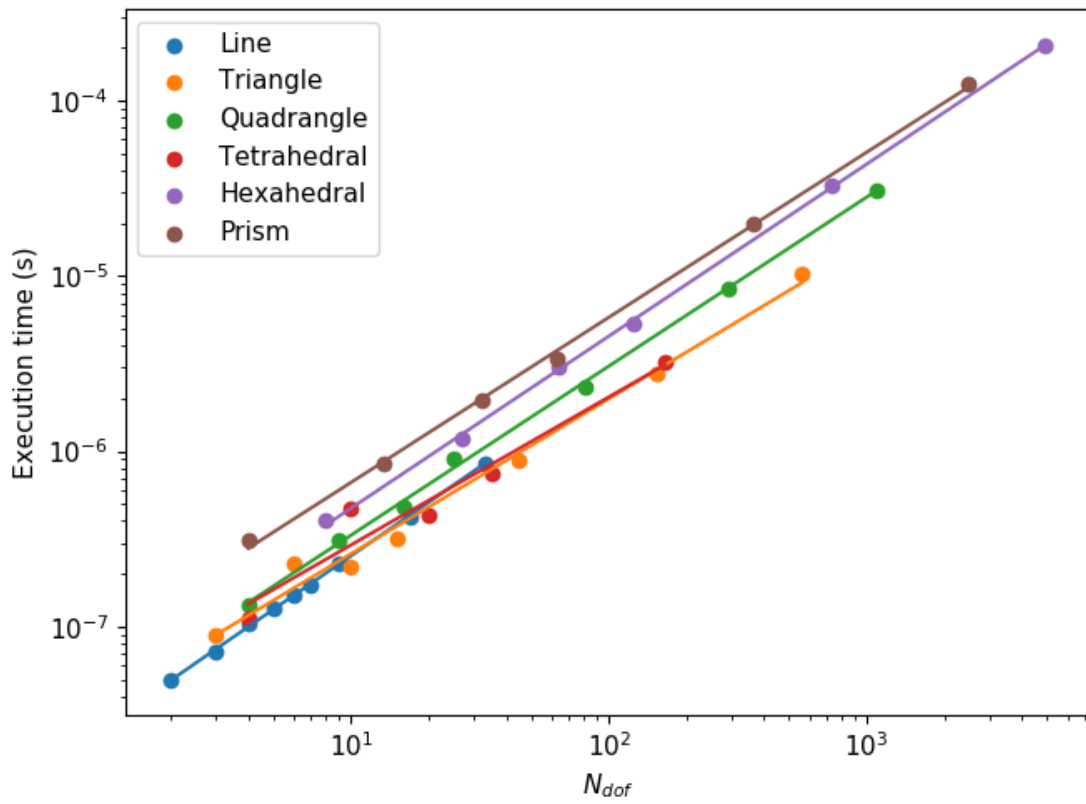


Figure 10. Time required to compute the complete basis (without derivatives) by the abstract templated approach.

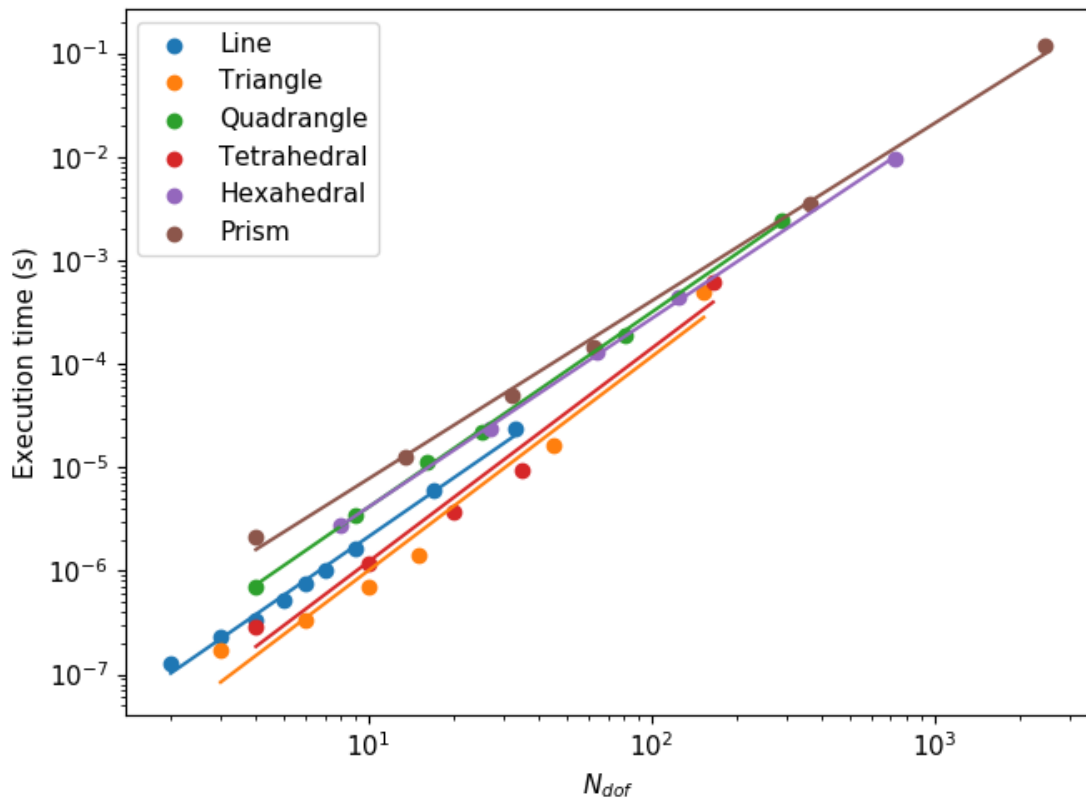


Figure 11. Time required to compute all derivatives by the abstract templated approach.

VI. Conclusions

In this study, two approaches to evaluating a basis of Lagrange polynomials and its derivatives are explored. First, an explicit approach is developed, in which a preprocessing program hardcodes expressions for the desired quantities into the source code. Then, a more involved computational procedure, known as the abstract approach, is formulated in order to eliminate redundant operations. The computational cost of each of these approaches is evaluated both theoretically and experimentally.

The theoretical analysis indicates that the time required to execute the abstract implementations should grow more slowly than the explicit implementations as the degree of the polynomials is increased. However, the timing experiments indicate that the implementation in code is more important than the choice of explicit versus abstract. Furthermore, since the approaches which are fastest to execute take a large amount of time to compile, compilation time can be as important as execution time. Hardcoding the abstract algorithms with a preprocessing program was found to give a good combination of execution and compile time for low to moderately high order. By contrast, the templated abstract approach is the only one of the three which is viable at extremely high order. Thus, these two approaches are complementary.

This study develops and evaluates techniques for computing a Lagrange basis. Although these methods and results may be generalized to other bases, such as orthonormal polynomials, it would be useful for future research to provide a detailed study of such cases. Additionally, a study of the relative importance of the cost of basis evaluation in the overall efficiency of DG codes could help programmers design or select a basis evaluation method. Finally, efficient basis evaluations for pyramid elements should also be studied.

References

- ¹Tie Zhou, Yinfan Li, and Chi-Wang Shu. Numerical comparison of weno finite volume and runge–kutta discontinuous galerkin methods. *Journal of Scientific Computing*, 16(2):145–171, Jun 2001.
- ²A. F. Antoniadis, K. H. Iqbal, E. Shapiro, N. Asproulis, and D. Drikakis. Comparison of high-order finite volume and discontinuous galerkin methods on 3d unstructured grids. In *AIP Conference Proceedings*, volume 1389, pages 1886–1889. American Institute of Physics, 2011.
- ³Alireza Mazaheri, Chi-Wang Shu, and Vincent Perrier. Bounded and compact weighted essentially nonoscillatory limiters for discontinuous galerkin schemes: Triangular elements. *Journal of Computational Physics*, 395:461–488, 2019.
- ⁴G. Bedrosian. Shape functions and integration formulas for three-dimensional finite element analysis. *International Journal for Numerical Methods in Engineering*, 35, 1992.
- ⁵Jean-Paul Berutt and Loyd N. Trefethen. Barycentric lagrange interpolation. *SIAM Review*, 46(3):501–517, 2004.
- ⁶Jan S. Hesthaven and Tim Warburton. *Nodal Discontinuous Galerkin Methods*. Springer, 2008.
- ⁷E. Meijering. A chronology of interpolation: From ancient astronomy to modern signal and image processing. *Proceedings of the IEEE*, 90(3):319–342, 2002.
- ⁸S. De Marchi and M. Morandi Cecchi. The polynomial approximation in the finite element method. *Journal of Computational and Applied Mathematics*, 57, 1995.
- ⁹Aaron Meurer, Christopher P. Smith, Mateusz Paprocki, Ondřej Čertík, Sergey B. Kirpichev, Matthew Rocklin, AMiT Kumar, Sergiu Ivanov, Jason K. Moore, Sartaj Singh, Thilina Rathnayake, Sean Vig, Brian E. Granger, Richard P. Muller, Francesco Bonazzi, Harsh Gupta, Shivam Vats, Fredrik Johansson, Fabian Pedregosa, Matthew J. Curry, Andy R. Terrel, Štěpán Roučka, Ashutosh Saboo, Isuru Fernando, Sumith Kulal, Robert Cimrman, and Anthony Scopatz. Sympy: symbolic computing in python. *PeerJ Computer Science*, 3:e103, January 2017.
- ¹⁰Nicholas Higham. The numerical stability of barycentric lagrange interpolation. *IMA Journal of Numerical Analysis*, 24, 10 2004.
- ¹¹Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. <http://eigen.tuxfamily.org>, 2010.