A Phasta Implementation of Isogeometric B-Spline Basis Functions for Structured CFD

Corey Nelson

1. Introduction

Over the last decade, Isogeometric Analysis (IGA) has rapidly matured as a high order numerical method for solving problems in solid mechanics and fluid dynamics. Since the seminal paper was published in 2005 [1], dozens of fields have enjoyed the high-order accurate benefits of IGA, and computational fluid dynamics is certainly among them. In this project, we hope to bring some IGA capability to the legacy CFD code known as Phasta.

The goal of this project is to implement a B-spline hexahedral element into the Phasta framework. The element will be defined in the context of a structured cube domain, so much of the complexity of mesh generation will be avoided in this first pass. The efficacy of the proposed element will be evaluated with a Taylor-Green Vortex test case. We anticipate the Isogeometric element will produce more accurate result than the current classical Finite Element bases employed currently in Phasta, though at a higher computational cost.

The Isogeometric element will take the form of a set of local Bernstein polynomial basis functions which will be mapped to global B-spline bases through a process known as *Bèzier Extraction*. This extraction process is a complexity not employed in classical Finite elements, and is thus the major complexity to implement in Phasta.

The thrust of this project will be to provide the extraction operators and basis functions, and Arvind Raghunath Dudi will develop the infrastructure throughout the code base to pass around the extraction operator. Thus, the two of us are working closely to provide a cohesive implementation of the IGA technology.

2. Mathematical Preliminaries

2.1. Bernstein Polynomials

Bernstein polynomials are defined over $\xi \in [0, 1]$ as

$$B_{a,p}(\xi) = (1-\xi)B_{a,p-1}(\xi) + \xi B_{a-1,p-1}(\xi), \tag{1}$$

$$B_{1,0}(\xi) = 1, \tag{2}$$

$$B_{a,p}(\xi) = 0 \text{ if } a < 1 \text{ or } a > p+1. \tag{3}$$

These polynomials have several handy properties with regard to their implementation as a basis for analysis. First, they are interpolatory at the endpoints of the domain $[0,1]$, and are a partition of unity, i.e.:

$$\sum_{a=1}^{p+1} B_{a,p}(\xi) = 1.$$

2.2. B-Splines

Univariate B-splines are defined as piecewise polynomial curves comprised of linear combinations of B-spline basis functions. The basis functions for B-splines can be produced through a recursion relation utilizing three pieces of information. These are p, the polynomial degree, n, the number of basis functions and Ξ, the so called *knot vector*.

The knot vector Ξ encodes information about the B-spline's continuity. It is a vector of nondecreasing real numbers ξ_i such that $\xi_i \leq \xi_{i+1}$. To define a B-spline basis, we require $n+p+1$ knots in our knot vector. For our purposes, we will impose two more constraints on the knot vector, that the knot vector is *open* such that the first and last knot are repeated $p+1$ times, and that the knot vector is normalized such that $\xi_1 = 0$ and $\xi_{n+p+1} = 1$.

B-spline basis functions can be computed in a number of ways, but we utilize the Cox-De Boor recursion relation detailed in [2]. The basis function $N_{i,p}$ is the *ith* basis function of degree p. For a detailed explanation of efficient algorithms for computing b-spline bases and their derivatives, we point to any number of books and articles, including [3, 4].

Multi-dimensional B-spline bases can be built in a tensor product fashion. We can define the multi-index $\mathbf{i} = \{i_1, \ldots, i_{d_p}\}$ which defines the index of the basis within the tensor product, and $\mathbf{p} = \{p_1, \ldots, p_{d_p}\}$ which defines the polynomial degree in direction d. Here, d_p is the dimension of the parametric domain. Then, the basis function in d_p dimensions is defined over the parametric domain with coordinates $\boldsymbol{\xi} = \{\xi_1, \ldots, \xi_{d_p}\}$:

$$N_{\mathbf{i},\mathbf{p}}(\boldsymbol{\xi}) = \prod_{d=1}^{d_p} N^d_{i_d,p_d}(\xi_d) \tag{4}$$

We define quantities in terms of B-splines by summing the product of B-spline bases and control variables. For instance, the quantity $\phi(\xi)$ which could be the solution to the advection-diffusion equation over the domain Ω. We approximate ϕ with B-splines defined over Ω and control variables d_i^{ϕ}:

$$\phi(\xi) = \sum_{i=1}^{n} d_i^{\phi} N_{i,p}(\boldsymbol{\xi}) \tag{5}$$

This formulation is useful for simple rectangular domains, and is how we will construct solutions over square domains in this paper, however, we could extend this to more

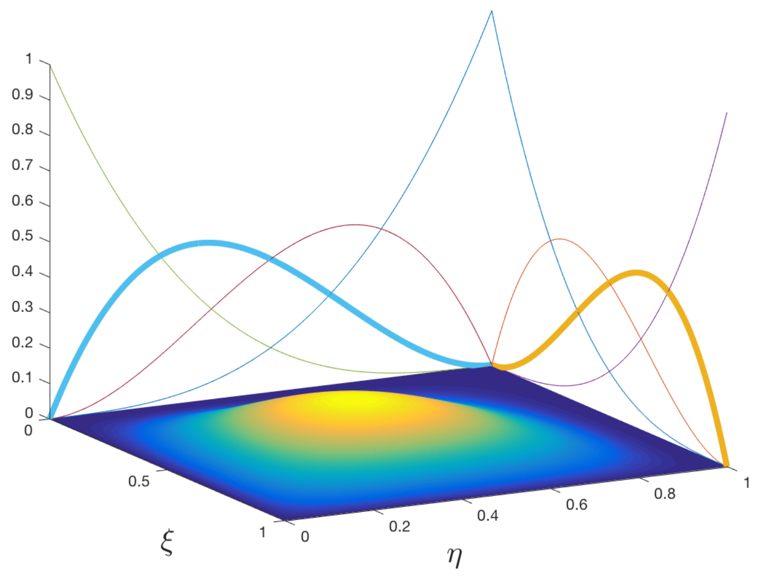

Figure 1: Typical 2D B-spline basis function

interesting domains by mapping our parametric coordinates $\boldsymbol{\xi}$ to physical coordinates $\mathbf{x}$ via the mapping:

$$\mathbf{x} = \sum_{\mathbf{i}} \mathbf{P_i} N_{\mathbf{i},\mathbf{p}}(\boldsymbol{\xi}) \tag{6}$$

where $\mathbf{P_i}$ are known as control points, and live in physical space $\mathbf{P_i} \in \mathbb{R}^{d_s}$ where d_s is the dimension of physical space. Setting $d_p = 1$, and $d_s = 3$ allows us to construct curves in 3D space. Similarly, setting $d_p = 2$ and $d_s = 3$ produces surfaces, and so on.

2.3. Bezier Extraction

Bèzier extraction, discussed at length in [5] fundamentally links the the B-spline basis technology of IGA with the data structures prevalent in classical Finite Elements. As illustrated above, B-spline bases have support spanning several elements (knot spans). This extended support is partially responsible for the improved accuracy of IGA over typical Finite element methods, though it is also fundamentally incompatible with current discretization technologies widely employed for performing Finite Element Analysis. Specifically, classical Finite elements relies on a straightforward pull back from an element in the parametric domain to a parent element. This mapping is simple because typical bases utilized in classical Finite Elements are designed strictly with local support over an element in parametric space. The concept of a parent element is important to the implementation of classical Finite Elements, as information about basis functions and quadrature must only be stored for a single element, as opposed to having bases evaluated at quadrature points over every element in the domain. The parent element dramatically reduces the memory cost of Finite Elements and is thus ubiquitous in Finite Element packages. This presents a problem for our B-spline bases, as they do not have local support, we must store basis information over each element. Fortunately, through the extraction operator, we may convert our B-spline bases to Bernstein bases with local

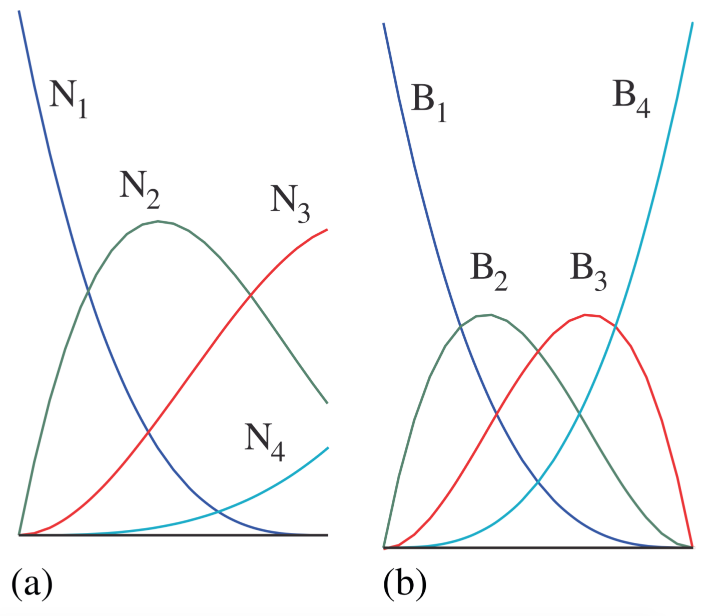

Figure 2: [5] B-Spline bases over a knot span (as in (a), $[0, 1]$) can be written as linear combination of Bernstein bases (b) over their unit domain. The extraction operator stores the coefficients of these linear combinations.

support over each element (knot span). This mapping is performed through the *extraction operator*.

The extraction operator maps Bernstein polynomials to B-spline bases via a simple linear combination. As such, a single Bernstein parent element can be employed to integrate B-spline bases into a typical Finite Element data structure. These two sets of bases are shown side by side in Figure 2.

3. Proposed Implementation

3.1. New Subroutines

The major subroutines to implement surround the construction of the Extraction operator and the definitions of the inputs for that subroutine.

1. `knotVecCreate()` creates a knot vector suitable for the the definition of the extraction operator and the basis functions. This subroutine takes in a start and end point for a domain in 1D (often 0 and 1, but these are parameters) and a number of elements, and returns an open knot vector with the number of open knots relating to the number of elements desired across the domain in 1D.
2. `BasisFuncsAndDers()` returns the value of all the basis functions an their derivatives defined at a point in the domain. These can be Bernstein polynomials or full B-spline bases depending on the knot vector passed in. An open knot vector with no interior knots will return Bernstein bases of order p - the desired polynomial order.
   - `FindSpan()` is a subroutine that finds the knot span in which the selected point lives. it is requried for computing B-spline bases.
3. `Extraction_1D()` returns the extraction operator in 1D for a given knot span and

polynomial order p.

4. `shphexIGA()` packages all these functions up into a fortran interface which returns the bases, their derivatives and the extraction operator. This function will be largely based on the `shphex()` routine currently in Phasta. The key difference is that `shphexIGA()` will also return the extraction operator. This leads to the major implementation change needed to be made in Phasta.

3.2. Phasta Subroutines to Modify

Because the extraction operator must be passed around the solver, we require adding the extraction operator array to the list of arrays already passed around. This will be accomplished by adding another array to the top of `genshp()`: `extrct(nel1D, p+1, p+1)`. The extraction operator per element is a square matrix of size $p+1$, and can be stored in each spatial direction for a tensor product domain. Thus, this 1D extraction operator is all that is required for storage.

Several other minor changes are being made to the code to hardcode our Taylor Green vortex problem, and interface our data structure with the rest of the code. Primarily, this involves adding global variables to `common.h` and the corresponding `common_c.h`.

1. ADD `HexIGAShapeAndDrv()` function primitive to `topo_shapedefs.h`, the header file containing function primitives for the element shape functions.
2. ADD `HexIGAShapeAndDrv()` definition to `newshape.cc`, the source code for element shape functions.
3. ADD `switch` case (6) to `genshp.f` to call `shphexIGA()`. `genshp()` is responsible for building the list of element blocks of each different topologies via a switch statement. We set the switch condition for our implementation to only hit the block of IGA elements. That way, we can ensure our test case is built only of our IGA elements.
4. CREATE new function `shphexIGA()` as a fortran wrapper to get IGA shape functions from `HexIGAShapeAndDrv()`
5. EDIT `genshp` to take in parameter `num_elem_1D` to define the number of elements along one side of the cube domain
6. CHANGE in `common.h` line 18, `NSD = 1` instead of 3. Our implementation involves a tensor product of 1D basis functions, so setting this to 1 should ensure we only build our bases once.
7. CHANGE in `common_c.h` line 64, `NSD = 1` instead of 3
8. in `genshp.f` set `lcsyst = 6` to ensure only IGA hex elements are built
9. in `genshp.f` set `nshl = ipord+1` to ensure number of shape functions are locally always $p+1$code
10. in `common_c.h #define num_elem_1D 16` to fix the domain of our test case to be 16^3 elements.
11. in `common.h` line 18 add `num_elem_1D = 16`

It is largely the responsibility of Arvind to pipe the `extrct` data structure through the solver. He will write a function `shpIGA(extrct, shp, indices)` which computes the global bases for each element in the domain. This function will be called throughout the solver where currently the value of `shp()` is evaluated. Arvind is also writing a similar routine for computing global basis gradients.

4. Descriptions of Phasta Functions

Here, we describe the functionality of the major subroutines found in `phSolver`. This section is divided by file. Some of the `Sclr` versions of various subroutines are not described, as they have identical responsibilities to their non `Sclr` counterparts though for a scalar problem, as opposed to the full Navier-Stokes problem.

4.1. asbflx.f

`AsBFlx()` is responsible for the assembly of flux terms at boundary elements into the global system. The residual term is computed via a call to `e3b()`. The LHS matrix is assembled via a call to `f3lhs()` which also computes the boundary normal of each element.

4.2. asbmfg.f

`AsBMFG()` is responsible for assembling all other boundary terms into the global system. This is done through a cll to `e3b()` to compute a element local residual `rl` which is then assembled into `res`.

Similarly, `AsBMFGSclr()` assembles the residual for a scalar problem into the global system via a call to `e3bSclr()` which returns the element level residual `rtl`.

4.3. asigmr.f

`AsIGMR()` is responsible for assembling the residual and LHS matrix for the interior elements through a call to `e3()` over each element. `EGmass` is computed in `e3()` and assembled into a block-diagonal form `BDiagl` which is assembled into the global block diagonal matrix `BDiag`.

4.4. asiq.f

`AsIq()` is responsible for assembling data over interior elements in order to reconstruct the global diffusive flux vector. This is done through a call to `e3q()` on each element to build the local diffusive flux vector `ql` which is then assembled into `qres`

4.5. bc3lhs.f

`bc3LHS()` is responsible for satisfying boundary conditions at the element level for the LHS mass matrix. Effectively, over all Dirichlet boundary conditions, the row of the LHS matrix must be zeroed out, and a 1 placed in the diagonal component of the corresponding row. This subroutine is responsible for performing this operation for all implemented boundary conditions.

4.6. bc3per.f

`bc3per()` is responsible for applying periodic boundary conditions by setting the rows corresponding to the pair of periodic boundary locations to equal eachother.

4.7. bc3res.f

`bc3Res()` is responsible for applying boundary conditions to the residual vector. This is done utilizing the same logic as `bc3lhs()`, but Diriclet BC values are added to the residual vector at BC locations.

4.8. BCprofile.f

`initBCprofileScale()` is responsible for reading in user set boundary condition profiles, such as ramped inlet profiles etc.

4.9. bflux.f

`Bflux()` is responsible for computing boundary fluxes and writing the computed fluxes to a file along with printing the primitive variables to another file. First, the variables stored in `y` are scaled from dimensionless form to dimensional form. Fluxes are computed through a call to `AsIFlx()` and `AsBFlx()` which compute `flxLHS`, `flxres` and `flxnrm`, the flux LHS, flux residual and outward normal vector of boundary elements. These are then used to solve for fluxes through boundaries.

4.10. e3.f

`e3()` is responsible for for computing the RHS residual of a 3D element, and is also capable of computing a modified residual and a block-diagonal preconditioner for an element. This is accomplished via a slough of calls to functions each of which computes a component of the local system:

1. `e3ivar()` computes the values of all the integration variables at a single point within the element including all components of velocity, temperature, pressure, enthalpy, internal energy etc (see code for full list)
2. `e3mtrx()` computes variable transform matrices `A_0`, `A_1`, `A_2` and `A_3` at a point in an element.
3. `e3conv()` computes the convection contribution to the standard Galerkin form of the Navier-Stokes.
4. `e3visc()` computes the viscous contribution to the standard Galerkin form of the Navier-Stokes.
5. `e3source()` computes body force contributions.
6. `e3LS()` computes the Least-Squares components to the stabilized Galerkin form. This computes the SUPG terms.
7. `e3massr()` computes element level contribution to the mass residual vector.
8. `e3massl()` computes the element level contribution to the mass LHS matrix.
9. `e3bdg()` computes the element level contribution to the block diagonal preconditioner matrix.

4.11. e3b.f

`e3b()` is responsible for putting together the RHS residual of the boundary elements. This is done with similar logic to the RHS computation done over interior elements in `e3()`, though with fewer external function calls. For instance, terms related to the pressure, convective, viscous and heat fluxes are computed within `e3b()`. Variable quantities are found via a call to `e3bvar()`. Material properties of the fluid are needed, and are found in a call to `getDiff()`.

4.12. e3bvar.f

`e3bvar()` is responsible for computing variable values at a point on a surface of a boundary element. Here, temperature, pressure and velocity components are computed, along with kinetic energy and several thermodynamic properties (via a call to `getthrm()`). Along with variable values, element metrics such as normal vectors and parametric derivatives are computed.

4.13. e3dc.f

`e3dc()` is responsible for computing terms for the residual vector related to discontinuity capturing. This process allows for shocks to be computed within the fluid volume.

4.14. e3q.f

`e3q()` is responsible for computing the element level contribution to the diffusive flux vector and the lumped mass matrix. This is accomplished through a call to `e3qvar()` to compute basis function gradients, element metrics and state variable values, and a call to `getDiff()` for material properties (ρ, c_p, etc.).

4.15. e3ql.f

`e3ql()` performs the same job as `e3q()` in computing the diffusive flux vector, but this uses a different method. This utilizes a local projection of the vector, rather than mapping from parametric to physical space via multiplication by the determinant of the element Jacobian.

4.16. e3qvar.f

`e3qvar()` is similar to `e3ivar()` and `e3bvar()` in that it computes variable values required for its corresponding element construction subroutine (`e3q()` in this case).

4.17. elmgmr.f

`ElmGMRs()` is responsible for putting together the various data structures required for the GMRes solver. These are the LHS matrix, the RHS residual vector and the block diagonal preconditioner. First, a number of parameters are setup pertaining to the blocks of element topologies living in the fluid domain discretization such as `nenl` the number of vertices per element in the block, `ndofl` the number of degrees of freedom in an element, `ngauss` the number of quadrature points on the element, etc. Then, assembly operations are called. `AsIq()` assembles the diffusive flux vector components in the interior of the domain, `AsIGMR()` assembles the residual and tangent matrix of interior elements, `AsBMFG` takes care of boundary integral assembly. A number of boundary condition handling functions are also called to ensure that the desired boundary conditions are satisfied for each assembled component of the global system. These are: `bc3LHS()` which handles boundary conditions on the LHS matrix, `qpbc()` which handles periodic boundary conditions on the diffusive flux term and `rotabc()` which is responsible for handling axisymmetric boundary conditions via periodicity. Along with these, a call is made to `bc3Res()` which applies boundary conditions to the global residual vector, and finally a call is made to `bc3BDg()` to apply boundary conditions to the block diagonal preconditioner matrix.

4.18. itrbc.f

`iterBC()` is responsible for satisfying boundary conditions on the Y set of variables. This is done in a straightforward fashion, utilizing the `BC` and `iBC` arrays to compute boundary conditions. These conditions are cast back to the Y vector.

4.19. itrdrv.f

`itrdrv()` is responsible for time stepping via the Generalized-α method. This function is essentially the top of the solver hierarchy. This function makes a call per time step inner iteration to a GMRes solver via the `SolGMR()` family of functions for the linear system solution.

4.20. itrPC.f

This file is responsible for the subroutines needed for the predictor-multicorector method within the Generalized-α method.

1. `itrSetup()` is responsible for setting the parameters used in the predictor-multicorrector method.
2. `itrPredict()` performs the prediction of the solution at time step `n+1`, updating the variables `y` and `yc` along with their past values `yold` and `acold` with the next prediction on the inner loop.
3. `itrCorrect` is responsible for the correction step of the predictor-multicorrector method. This is done by computing $Y_{n+1} = Y_n - \Delta Y$.
4. `itrUpdate()` updates the solution variables at the end of the given time step.

4.21. solgmr.f

This file contains subroutines for completing a GMRes solve of the global system. There are two methods contained in this file. One GMRes solver for a sparse matrix solve, and another for an element-by-element solve.

1. `SolGMRe()` computes a GMRes solve on an element by element basis. This is done via a local assembly of the variable `EGmass` which is multiplied successively by a local vector to build up a local Krylov space which is used to build the linear system solution globally.

2. `SolGMRs()` performs a global system solve via a sparse matrix solve. element level mass matrices have been assembled into a global sparse matrix, and then that sparse matrix is successively multiplied by vectors to build up the Krylov space of the global system, which is then used to build the solution to the global linear system.

5. References

[1] T. Hughes, J. Cottrell, Y. Bazilevs, Isogeometric analysis: CAD, finite elements, NURBS, exact geometry and mesh refinement, Computer Methods in Applied Mechanics and Engineering 194 (2005) 4135–4195.

[2] C. de Boor, On calculating with B-splines, Journal of Approximation Theory 6 (1972) 50–62.

[3] L. Piegl, W. Tiller, The NURBS book, Springer Science & Business Media, 2012.

[4] J. Austin Cottrell, T. J. R. Hughes, Y. Bazilevs, Isogeometric Analysis: Toward integration of CAD and FEA, 2009. DOI: 10.1002/9780470749081.ch7.

[5] M. J. Borden, M. A. Scott, J. A. Evans, T. J. R. Hughes, Isogeometric finite element data structures based on Bzier extraction of NURBS, International Journal for Numerical Methods in Engineering 87 (2011) 15–47.