An Isogeometric Solver for the Laplace-Beltrami Operator

Corey Nelson

1. Introduction

Over the past decade, Isogeometric Analysis (IGA) has grown in popularity 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, from fluid dynamics to structural mechanics. Isogeometric Analysis lies at the interface of computational analysis and computational geometry, and this project project intends to explore the relationship between these two fields.

The goal of this project is to implement an Isogeometric Analysis (IGA) solver of the Laplace-Beltrami Operator defined over NURBS (Non-Uniform Rational B-Spline) surfaces in $\mathbb{R}^3$. This is accomplished in C++ using the PETSc library for completing the linear algebra computations.

2. Mathematical Preliminaries

2.1. The Laplace-Beltrami Operator

The Laplace-Beltrami operator is similar to the standard Laplacian as it is defined as the divergence of the gradient of a function. However, in the context of the Laplace-Beltrami operator, the function in question is defined over a surface living in space. Thus, gradients and divergences are taken over the surface as opposed to in the strict cardinal directions, we will denote gradients over the surface as ∇_Ω and the Laplace-Beltrami operator as Δ_Ω [2].

Given a surface Ω in $\mathbb{R}^3$ with boundary Γ which has outward facing normal $\mathbf{n}$, along with source term $f \in L^2(\Omega)$, the Laplace-Beltrami problem seeks to find $u : \Omega \to \mathbb{R}$ such that

$$-\Delta_\Omega u = f \in \Omega, \tag{1}$$

$$u = g \in \Gamma_D, \tag{2}$$

$$\mathbf{n}_\Gamma \cdot \nabla_\Omega u = 0 \in \Gamma_N. \tag{3}$$

We convert this to a symmetric weak form which takes the form: find $u \in \mathcal{V}$ such that

$$(\nabla_\Omega u, \nabla_\Omega v) = (f, v)\,, \forall v \in \mathcal{V}_0. \tag{4}$$

The space $\mathcal{V}$ is the space of all functions in H^1 defined over the surface Ω, and the space $\mathcal{V}_0$ is the subset of functions in $\mathcal{V}$ with boundary values equal to zero.

In order to compute the integral above, we can pull it back to the parametric domain. We define

$$\alpha\left(\hat{u},\hat{v}\right)=\int_{\hat{\Omega}}D\mathbf{F}^{+}\hat{\nabla}\hat{u}\cdot\left(D\mathbf{F}^{+}\hat{\nabla}\hat{v}\right)Jd\xi \tag{5}$$

Where $D\mathbf{F}$ is the Jacobian of the mapping F between parametric and physical space, and J is the determinant of the Jacobian. This presents a challenge for parametrically 2D surfaces in $\mathbb{R}^3$, as the Jacobian is not square, i.e.:

$$D\mathbf{F}=\begin{bmatrix}\frac{\partial x}{\partial\xi} & \frac{\partial x}{\partial\eta}\\ \frac{\partial y}{\partial\xi} & \frac{\partial y}{\partial\eta}\\ \frac{\partial z}{\partial\xi} & \frac{\partial z}{\partial\eta}\end{bmatrix} \tag{6}$$

We will utilize the Monroe-Penrose pseudo-inverse for this mapping [3]

$$D\mathbf{F}^{+}=(D\mathbf{F}^{T}D\mathbf{F})^{-1}D\mathbf{F}^{T}. \tag{7}$$

And we will take the determinant of the Jacobian as

$$J=\sqrt{\left(\frac{\partial y}{\partial\xi}\frac{\partial z}{\partial\eta}-\frac{\partial z}{\partial\xi}\frac{\partial y}{\partial\eta}\right)^{2}+\left(\frac{\partial z}{\partial\xi}\frac{\partial x}{\partial\eta}-\frac{\partial x}{\partial\xi}\frac{\partial z}{\partial\eta}\right)^{2}+\left(\frac{\partial x}{\partial\xi}\frac{\partial y}{\partial\eta}-\frac{\partial y}{\partial\xi}\frac{\partial x}{\partial\eta}\right)^{2}}. \tag{8}$$

2.2. Bernstein Polynomials

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

$$B_{a,p}(\xi)=(1-\xi)B_{a,p-1}(\xi)+\xi B_{a-1,p-1}(\xi), \tag{9}$$
$$B_{1,0}(\xi)=1, \tag{10}$$
$$B_{a,p}(\xi)=0 \text{ if } a<1 \text{ or } a>p+1. \tag{11}$$

We will use Bernstein polynomials as the local description of elements over our surface, and relate them to the global basis functions via the process of *Bèzier Extraction*. 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.:

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

2.3. 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 [4]. 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 [5, 6].

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 $\boldsymbol{\xi} = \{\xi_1, \ldots, \xi_{d_p}\}$:

$$N_{\mathbf{i},\mathbf{p}}(\boldsymbol{\xi}) = \prod_{d=1}^{d_p} N^d_{i_d,p_d}(\xi_d) \tag{12}$$

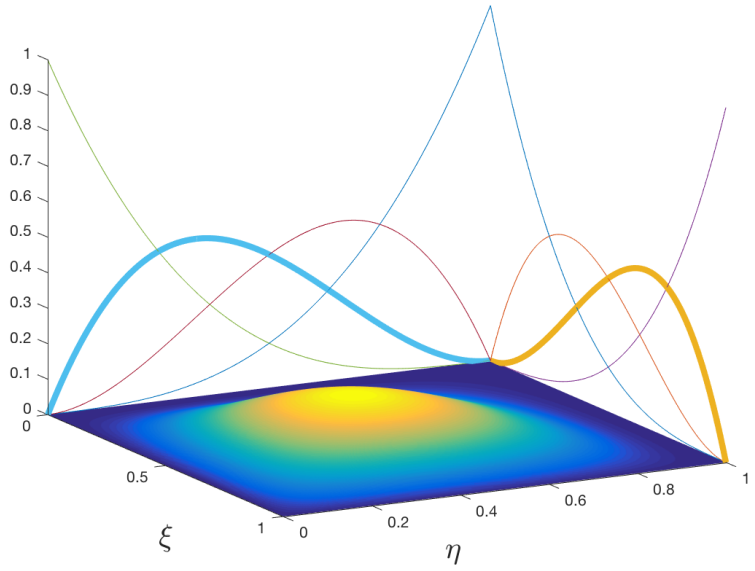

Figure 1: Typical 2D B-spline basis function

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:

$$\phi(\xi) = \sum_{i=1}^{n} d_i^\phi N_{i,p}(\boldsymbol{\xi}) \tag{13}$$

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

$$\mathbf{x} = \sum_{\mathbf{i}} \mathbf{P_i} N_{\mathbf{i,p}}(\boldsymbol{\xi}) \tag{14}$$

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.

2.4. NURBS

Finally, the functions we will use to describe our surface along with being the basis for analysis are the Non-Uniform Rational B-Splines (NURBS). In one dimension, NURBS are defined as ratios of of B-spline bases:

$$R_i(\xi) = \frac{w_i N_{i,p}(\xi)}{\sum_{b=1}^{n} w_b N_{b,p}(\xi)}, i = 1, \ldots, n. \tag{15}$$

NURBS can be built up into higher dimensions in exactly the same tensor product fashion as B-spline bases.

NURBS are used extensively in computer graphics, as they provide a means of exactly producing conic sections such as circles and ellipses [5]. In Figure 2, we use the 3D modeling program Blender to generate a cylinder. The set of vertices surrounding the cylinder are the control points making up the *control mesh* of the surface.

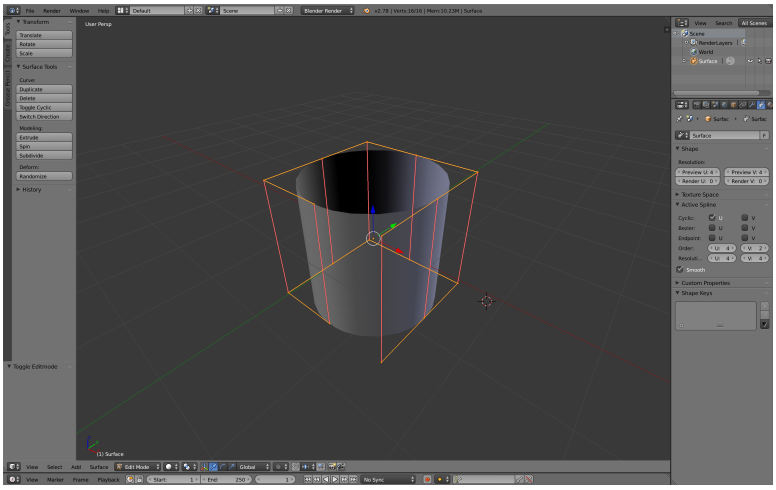

Figure 2: NURBS cylinder created in Blender

2.5. Bezier Extraction

Bèzier extraction, discussed at length in [7] fundamentally links the the NURBS basis technology of IGA with the data structures prevalent in classical Finite Elements. As

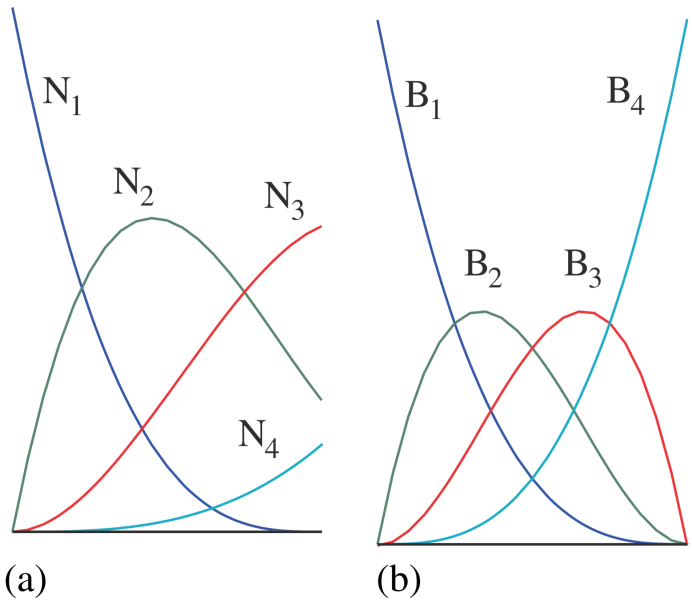

Figure 3: [7] B-Spline bases over a knot span (as in (a), $[0, 1]$) can be written as linear combination of Bernstein bases (b) over their unit domain. The extraction operator stores the coefficients of these linear combinations.

illustrated above, B-spline bases (and NURBS) 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. This presents a problem for our bases derived from B-spline technology, as B-splines and NURBS do not have local support, we would have to store information about each basis over every element in the domain. Fortunately, through the extraction operator, we may represent B-spline bases as a linear combination of Bernstein bases with local 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 3. This operator is constructed by mapping the effect of increasing the multiplicity of each knot through the interior of the domain until high order continuity of the bases is broken across knots, leaving C^0 continuous bases across knots. This expanded basis becomes Bernstein polynomials over each knot span. The effect of knot insertion is stored in matrix form in the extraction operator.

Bèzier extraction is possible over NURBS bases as well by projecting the NURBS bases onto a set of B-spline bases one dimension higher [5].

3. Implementation

3.1. Building Geometry in Blender

The surfaces used for analysis are built in the 3D modeling system Blender. Blender provides simple tools for creating and editing NURBS surfaces, such as the cylinder in FIgure 2 or the surface bump in Figure 4. Blender provides a rather robust scripting feature where python scripts can be utilized to manipulate the 3D scene. The custom blender script `BlenderExportNURBS.py` (found in `Shoreline/Scripts/`) was written to export NURBS geometry from Blender into a simple format which can be read into the application code.

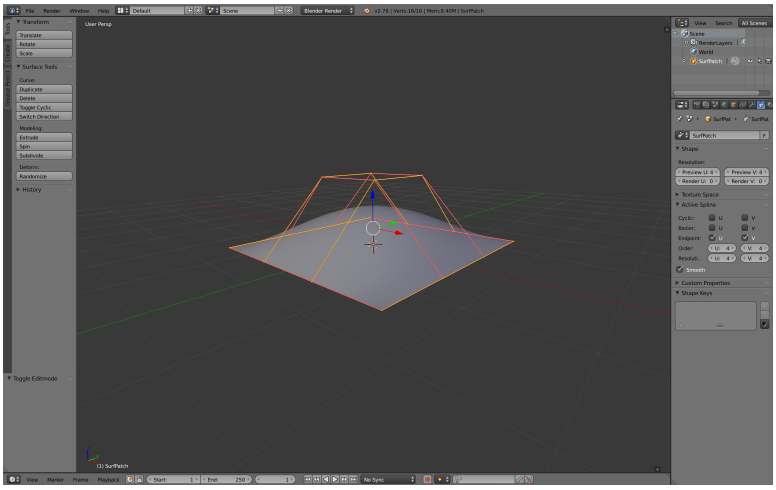

Figure 4: NURBS surface bump created in Blender

3.2. Surface Class

The `Surface` class is responsible for storing information about the imported surface, and prepares it for analysis. A `Surface` object is initialized with the file path of a surface exported from Blender. This file is read in with `import_Geometry(string geomFile)` which parses the polynomial order, the number of basis functions and the list of control points from this file. Knot vectors are not stored directly in Blender, but are computed, so the process for building knot vectors in Blender has been adapted for this application code within the `compute_knots()` set of functions. Generally, Blender will export single element geometries, which is generally not suitable for analysis. As such, a surface refinement tool can be run on the surface `refine()` which performs knot insertion, doubling the number of knot spans in each direction. The `Surface` object is made analysis ready with the command `ready_analysis()`, which builds the IEN array and computes the extraction operator along with the extracted control points (control points defining each element individually). The IEN array is built via a call to `ien_2D()` which calls `ien_1D(int uv)` for each direction, and then tensor products together the output of those calls. The extraction operator is similarly built via a call to `extraction_2D()` which calls `extraction_1D(int uv)` in each direction and tensor products together the output of each element's 1D extraction operator in the u and v directions.

3.3. Problem Class

The `Problem` class is responsible for building boundary conditions and forcing functions for analysis over the imported surface. Its constructor `Problem(Surface *srf, string problemType)` takes in a pointer to the `Surface` object, along with a string denoting the problem to be solved. That string is compared to the tags on each of the implemented problems. The `Problem` object then computes a vector of booleans `BC` which indicates which basis function is associated with a boundary condition. A corresponding vector of doubles `g` is computed which stores the value of the boundary conditions. Similarly, a vector of doubles `forcing` is computed which returns the value of the forcing function. We will keep the forcing to zero for now, but this will become useful moving forward.

3.4. Solver Class

The final component of the application code is the `Solver` class. A `Solver` object is created with the constructor `Solver_LaplaceBeltrami(Surface *srf, Problem *prb)`, taking in a pointer to a `Surface` and a `Problem` object. The `Solver` object is responsible for managing all of the PETSc data structures, and builds and solves the numerical solution to the problem at hand. It is also responsible for printing the solution and various data structures to files which can be read into and analyzed or visualized in MATLAB.

PETSc objects are built with the method `petsc_setup()`, which allocates the tangent matrix `Tangent`, the right hand side vector `rhs` and the solution vector `sol_vec`. The sparsity of the tangent matrix is determined by the number of degrees of freedom within elements on the surface. the solver uses the `KSPGMRES` solver with a `PCLU` preconditioner in order to do an exact solution to the matrix system in one iteration.

The other important class methods in the `Solver` class are `element_formation(int e)` and `element_assembly()`. The formation routine forms the local element stiffness matrix and right hand side over element `e`. This is done with a loop over quadrature points nested with a loop over basis functions defined over the element. The assembly step loops over each element, calling `element_formation()` and assembles those into the global system matrix. This process is nearly identical to typical element formation and assembly steps found in classical finite element methods.

4. Results

Unfortunately, results have been elusive. There is clearly an implementation error, We believe in applying boundary conditions because the solutions generated are rather odd! In Figure 5, we see the results of applying Dirichlet boundary conditions of 1 to the left side, and 0 on all other sides. This should result in a smooth transition of the solution from 1 to zero across the domain, with at worst some strange behavior at the corners where the boundary condition is discontinuous. However, it appears that only the elements on the left side are seeing the boundary condition, and that effect is not transfered throughout the domain. This solution was generated on a flat plate of unit side lengths with a 32x32 mesh. A similar situation occurs when applying uniform forcing. In

Figure **??**, we have applied homogeneous Dirichlet boundary conditions of zero around the entire boundary, and applied a uniform forcing of 1 throughout the interior. We expect the solution to be uniform near 1 throughout the interior, and for the solution to drop off significantly at the boundaries. What we see is several apparently random peaks. This is confusing, and certainly a bug in the solver.

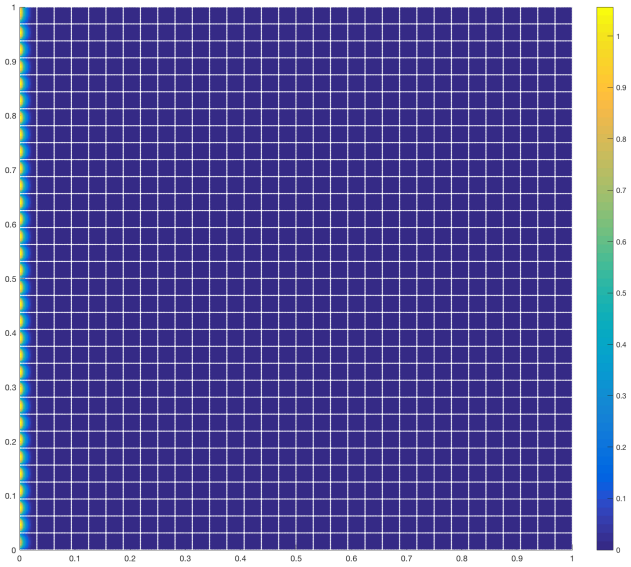

Figure 5: Odd behavior when applying Dirichlet boundary conditions

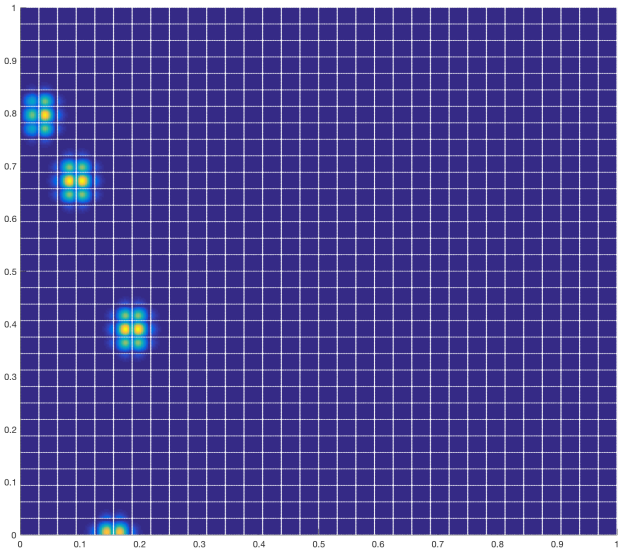

Figure 6: Odd behavior when applying uniform forcing

5. Future Work

Obviously, first order of business is to sort out the issues with applying forcing and boundary conditions.

The eventual goal of this work will be to build a solver capable of modifying metrics of the surface. For instance, the Laplace-Beltrami operator can be extended to solve over metric tensors such as curvature tensors. This becomes useful in modifying the parametrization of a surface. With this tool, the hope is to build a tool in which engineers can modify the geometry of a simulation in vitro without destroying the analysis suitable parametrization.

6. 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] Mimitrios Kamilis, NUMERICAL METHODS FOR PDES ON CURVES AND SURFACES, Master's Thesis, Umea University, 2013.

[3] R. N. Simpson, Z. Liu, R. Vazquez, J. A. Evans, An isogeometric boundary element method for electromagnetic scattering with compatible B-spline discretizations, arXiv:1704.07128 [math] (2017). ArXiv: 1704.07128.

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

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

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

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