Bigg Mann++ PDE Solver. A Complete Framework for Taylor Made Finite Difference Numerical Analysis

Ch. Orestis Papas

June 14, 2023

1 Abstract

This report presents the development and implementation of BiggMann++, an object-oriented C++ software specifically designed to solve second order partial differential equations (PDEs) in their most generalized form. The extensive utility of the software extends across numerous fields and disciplines, highlighting the universal presence of second order PDEs in a wide array of scientific phenomena. BiggMann++ embodies the developer's perspective on the physical world, the transition from continuum to discrete, from the beautiful mathematical symbols to computer memory pointers, and encapsulates a shift towards a more abstract interpretation of engineering problems. This perspective acknowledges that beneath the myriad physical phenomena and their varied coefficient nomenclatures reside identical core mathematical structures.

BiggMann++ was originally devised as a $C\#$ project for the "Mesh Generation" course during my master's program, with its primary focus being the generation of structured node grids by solving elliptic PDEs in parametric space. As the software evolved, so did my perspective, particularly during my research in the area of tumor development modeling. An insightful observation dawned upon me: the core mathematical constructs remained essentially unchanged, regardless of whether the coefficient represented the oxygen diffusivity in an oxygen mass transport equation, the g_{ij} component of the contravariant tensor in mesh generation. This epiphany underscored the universality of second order PDEs, revealing their beauty and pervasive presence across diverse scientific disciplines. Recognizing this pattern was a significant shift in perspective and ultimately inspired the transformation of BiggMann++ into a more generalized and parametrized tool for that offers control over the complete simulation process. The framework has transitioned to C++ over the course of the last 6 months after the need for custom memory management emerged. It allows user friendly manipulation of every aspect of the simulation from computational domain definition and mesh metrics to node and direction dependent PDE coefficients. Performance is also considered, but

2 Framework Description

The program aims to provide a complete simulation package for the user. It starts with an analysis that solves the Poisson equation for the structured grid creation, then proceeds to the actual engineering application. The underlying mechanisms for each analysis type are exactly the same.

2.1 Core Development Principles

The primary objective of this development effort was to construct a codebase that is easy to understand, readable, and indicative of the thought process embedded in its creation. The code employs terms familiar to any engineer for clarity. The adoption of object-oriented principles was an essential strategy to boost maintainability and circumvent error susceptibility, specifically by avoiding code repetition and casuistry. The key concepts are presented below:

- Complete Avoidance of External Libraries: The only non-custom utility used is the standard library. All numerical methods such as matrix-matrix and matrix-vector multiplication, as well as matrix decomposition, are based on the custom $std :: vector < T >$ container class `Array<T>`.

This approach improves understanding and control of the data structures used and leads to a better understanding of C++.

- Descriptive Naming and Avoidance of "Master Classes": Each variable, function, and class is named in a highly readable and explanatory manner. Classes are organized in such a way that their actions align with user intuition.

- Space Description: The computational space where the PDE is applied, and the relative positioning of two random points, is described with enumerations (enums).

  - Coordinate System: enum Direction (One, Two, Three, Time, None)
    This encapsulates any coordinate system that can be described using three orthonormal axes (1,2,3).
  - Relative Positions: enum Position (TopLeft, Top, TopRight, Left, Center,...,LeftTopBack).
    The consistent identification of nodes, combined with the properties of the isoparametric curves of the mesh, leads to an intuitive and robust description of the relation between any nodes with $O(1)$ complexity for all three Directions.
  - Coordinates: enum CoordinateType (Natural, Parametric, Template).
    This provides information about the spatial representation of each node, aiding in the calculation of mesh metrics.

- Schemes for Uniform and Non-uniform Node Distribution: The program exhibits the versatility to compute both the textbook, hardcoded Finite Difference (FD) schemes with errors of order $O(\Delta x^6)$ for uniformly distributed nodes, and scheme weight for nodes distributed in an arbitrary manner, maintaining an error order of $O(\Delta x^n)$.

- Consistent Cut-off Error Scheme: To apply a scheme with consistent accuracy across the entire domain, Finite Difference Schemes of different types (Forward, Backward, Central) and the same accuracy are applied according to the number of available neighbors. The algorithm successfully determines the graph of each node without considering special cases, leading to accurate higher order schemes.

- Space Decomposition: The implementation of enums and maps allows for the decomposition of the domain into Directions. The contribution of each Direction in all parts of the analysis is treated in loops without taking special cases under consideration.

- Variety of Numerical Solvers: A wide range of solvers is developed or under development.

  - Stationary Iterative Methods (Single / Multi Thread) : Jacobi, Gauss-Seidel, SOR
  - Krylov Subspace Methods (Single / Multi Thread) : Steepest Descent, CG, Preconditioned CG GMRES, Preconditioned GMRES (all under development)

- High Performance Computing Options : All Stationary Iterative Methods have parallelized version with `std::thread`. GPU Implementation with `openCL` under development.

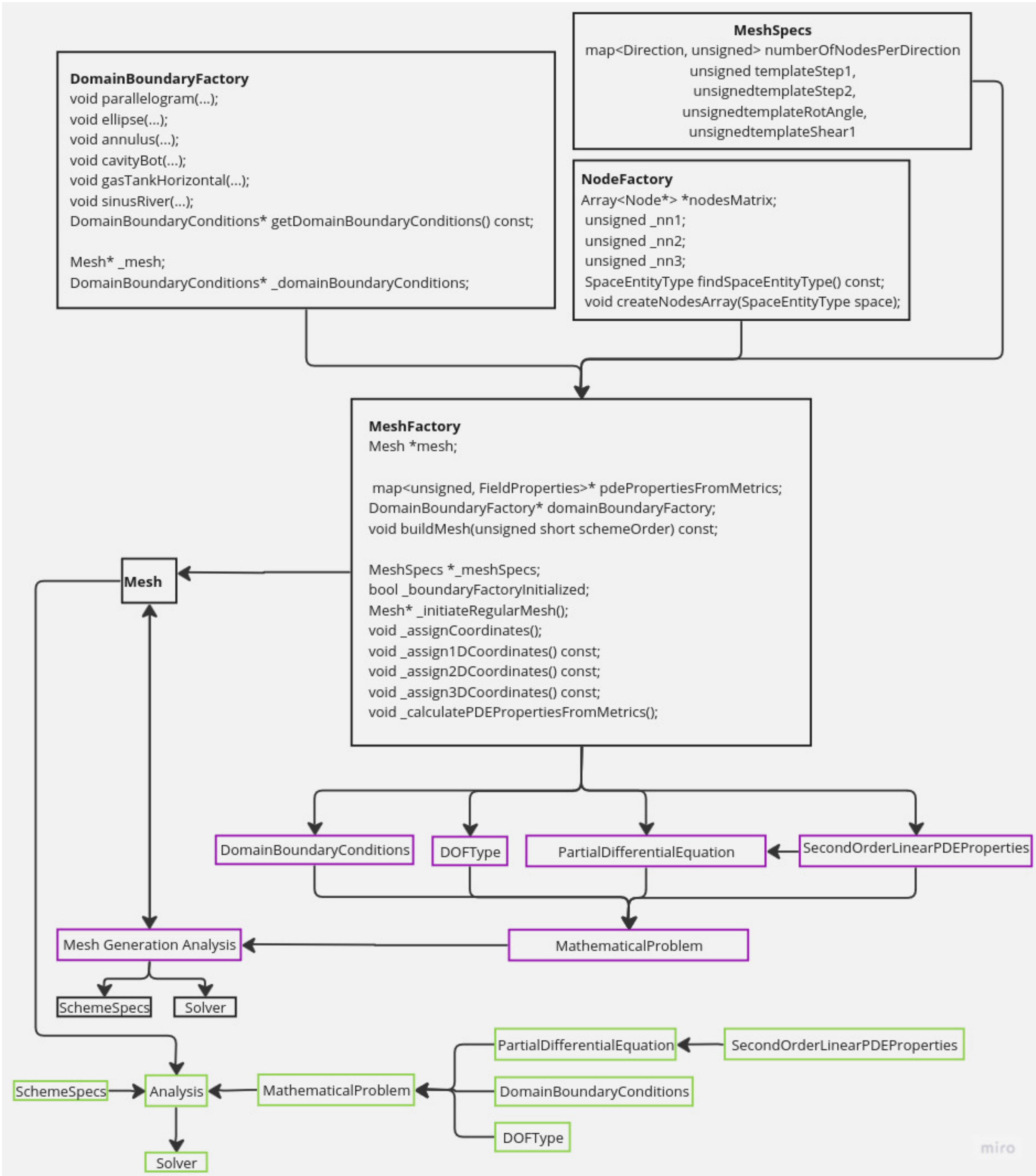

Figure 1: UML Diagram of the BigMann analysis workflow. Green and purple frames correspond to the mesh generation and the physics simulation components correspondingly

2.2 Numerical Analysis

The entire framework is built around the concept that a numerical analysis consists of a well-defined mathematical problem, a computational domain, and a solver.

2.3 Mathematical Problem

2.3.1 Mathematical Problem - PDE and PDE Properties

The mathematical problem comprises a Second Order Linear Partial Differential Equation with its corresponding coefficients, Degrees of Freedom, and Boundary Conditions. The coefficients and degrees of freedom define the scientific domain of the analysis. The PDE is mostly defined by its coefficients. The user can choose between isotropic and anisotropic properties that apply in the whole domain or vary for each node. For a scalar or vector function $\phi = \phi(x_1, x_2, x_3, t)$ the general second order PDE takes the form: $\boldsymbol{A} = Aij$:

- A_{ij} for $i, j < 3$ are the coefficients of the second order partial derivatives at each space direction.
  - When $A_{11} = A_{22} = A_{33}$ and $A_{ij} = A_{ji}$ the host medium is considered as isotropic meaning that it is symmetrical and homogeneous and has the same physical properties in all directions and it does not have any preferred direction of orientation. Examples of isotropic materials include most liquids and gases, as well as some solids such as glass and certain types of metals.
  - When $A_{11} \neq A_{22} \neq A_{33}$ and $A_{ij} \neq A_{ji}$ the host medium is anisotropic meaning that it is non-symmetrical and non-homogeneous. It does not have the same physical properties in all directions and it has preferred directions of orientation. Examples of anisotropic materials some metals and ceramics
- A_{i4}, A_{4j} are the coefficients of the second order partial derivatives with respect to time.
  - When $A_{i4}, A_{4j} \neq 0$ the equation is transient.
  - When $A_{i4}, A_{4j} = 0$ the equation is in steady state.

2.3.2 Mathematical Problem - Boundary Conditions

Boundary conditions are defined using the Position enums, $std :: map$, and lambda functions. The user can impose boundary conditions in the following ways:

- By setting a constant value across the boundary.
- By using a user-defined function of the boundary nodes coordinates $f(x_b, y_b)$.
- By assigning a different value at each boundary node.

At the time of writing, only Dirichlet Boundary Conditions are applied, but the implementation of Neumann conditions with improved accuracy is under development.

2.3.3 Mathematical Problem - Degree Of Freedom Types

The nature of the degrees of freedom in the framework are defined by the DOFType enum, and they are categorized in the DegreesOfFreedom vector field of the `Field_DOFType` struct. For example, the DOF vector for the structured mesh generation problem is $\{Position1, Position2\}$.

2.4 Analysis Degrees of Freedom

The `DegreeOfFreedom` class embodies the essential attributes and functions related to degrees of freedom (DOFs) in the computational problem. This class provides the basic functionality to manipulate these DOFs, such as setting the value and retrieving the constraint type or parent node ID. It also provides comparison operators to compare two DOF instances. The Degrees of Freedom are applied on the mesh nodes. They are described by the following properties:

- `DOFType` : Enum describing tha type of the dof (Position1, Displacement2, Velocity3, Temperature, UnknownScalarVariable, UnknownVectorFieldVariableComponent1 etc.
- `ConstraintType` : Enum defyining whether a dof is Fixed or Free
- `id` : unsigned ID based on the Constraint Type. DOF ordering complies with nodal ordering (Left to Right, Bottom to Top, Back To Front) for both Fixed and Free DOFs.
- `value` : double that is initialized with NaN for Free dof and the boundary condition value for Fixed (Dirichlet) dof.
- `parentNode` : unsigned that corelates the dof with the host Node ID.

The `DOFInitializer` class is responsible for the initialization of the DOFs for all the nodes in the mesh. The class constructor takes a mesh object, domain boundary conditions, and a list of DOFs types to be initialized. Depending on the nature of the boundary conditions (homogeneous or non-homogeneous), the initializer assigns fixed or free DOFs to the boundary nodes. Dirichlet and Neumann dofs are taged as Fixed or Free correspondingly, while all the internal dofs are Free. The class also provides functionality to assign unique IDs to each DOF depending on the `ConstraintType`, construct total DOF lists, and create additional data structures for efficient access to DOFs information.

Finally, the `AnalysisDegreesOfFreedom` class leverages the functionality of the `DOFInitializer` to generate and store DOFs for a numerical analysis. This class maintains different vectors and maps to hold free, fixed, flux, and total DOFs for the analysis. It also records the count of different types of DOFs. The class destructor is responsible for the deallocation of the generated DOFs, ensuring that the memory management is handled effectively.

2.5 Nodes

The `Node` class encapsulates the fundamental attributes and behaviors associated with a discrete point (node) in a discretized domain. It contains the following properties:

- `ID` : Unique Global ID
- `NodalCoordinates` : Class that manages the position vectors representing the coordinates of a node. The primary attribute of this class is a map between `NodalCoordinates` and vectors of doubles. These vectors represent the node's position in various coordinate systems (Natural, Parametric, Template). The class constructor initializes the position vectors map. During destruction, all dynamically allocated vectors inside the map are deleted to prevent memory leaks. `NodalCoordinates` provides methods to add, set, or remove position vectors corresponding to different coordinate types as well access to these vectors by constant reference or as pointers.
- `vector<DegreeOfFreedom*>*` : Vector with the Degrees of Freedom that belong to the Node.

`Node` can be initialized with an empty ID and coordinates, and an empty vector for its degrees of freedom. The class exposes methods to fetch the degree of freedom pointer and reference for a given DOFType.

2.6 Mesh

The `Mesh` object contains all the information necessary to describe the computational domain and is a key component of the analysis. The Node* are stored in a custom `Array<T>` class in row-major format, but are also housed in other data structures that allow easy access to boundary and internal nodes. It also offers some useful utilities.

Figure 2: UML Diagram of the Mesh class

2.6.1 Mesh Generation - Initialization

The `MeshFactory` class is responsible for creating and configuring a mesh.

- When a `MeshFactory` object is instantiated, it takes in a `MeshSpecs` object which contains details such as the number of nodes in each direction, the template step size in all Directions as well as the rotation and shear angles for the template mesh.
- With the `MeshSpecs` object, the `MeshFactory` initializes the mesh by invoking the `_initiateRegularMesh()` method. This method determines the type of the mesh (1D, 2D, or 3D) based on the number of nodes per direction, and creates a corresponding `Mesh` object.
- The `NodeFactory` class, which resides in the `StructuredMeshGenerator` namespace, is responsible for allocating the `Node*` objects and storing them in an `Array<Node*>*`. It also gives the nodes IDs that ascend from Left to Right, Bottom to Top, Back to Front.
- The parametric and template coordinate vectors are calculated and added to the `PositionVectors` . by invoking the `_assignCoordinates()` method. The appropriate rotation and shear transformations are applied at the template coordinates Depending on the type of space entity, it calls one of the methods `_assign1DCoordinates()`, `_assign2DCoordinates()`, or `_assign3DCoordinates()`.
- After assigning the coordinates, `MeshFactory` calculates the mesh metrics and uses them to compute the PDE properties, which are then stored in `pdePropertiesFromMetrics`.
- Finally, the `MeshFactory` constructs a `DomainBoundaryFactory` object for the mesh. The geometry of the boundaries is set from the user, where the programm is called.

2.6.2 Mesh Generation - Metric Tensors Calculation

The calculation of the metrics are performed by the `calculateMetrics` function. The input argument allows for calculation on the template or natural coordinates (only after solution). The process is summurized as follows:

- The function first checks whether the mesh has been initialized. If not, it raises a runtime error. If the mesh is initialized, it proceeds by creating a map to store the mesh metrics.
- A `FiniteDifferenceSchemeBuilder` object is created. This object provides access to utility functions needed to construct the finite difference scheme. The function also generates a `GhostPseudoMesh`, which is an auxiliary mesh that includes additional layers of ghost nodes, facilitating calculations at the boundary. The ghost nodes are initialized in the same way as the rest of the nodes by giving them coordinates that match the template.
- The function iterates over each node in the mesh. For each node, it generates an `IsoParametricNodeGraph` object. This object represents the neighbourhood of the node in the mesh, including the number of ghost nodes and diagonal neighbours.
- For each node, the function initializes a `Metrics` object to store the computed metrics. It obtains colinear nodal coordinates for both the parametric and template coordinate systems.
- The function enters a loop that traverses all spatial directions (I). For each direction, it computes the covariant and contravariant base vectors. These vectors represent the derivatives of real space coordinates with respect to curvilinear coordinates, and vice versa. The function calculates these vectors using appropriate weights and stores them in the `Metrics` object for the node. The weight calculation can be performed on uniform and non-uniform grids.
- After processing each direction, the function calculates the covariant and contravariant tensors. These tensors represent the metric tensors in covariant and contravariant forms, respectively. It then adds the `Metrics` object to the `metrics` map.
- Deallocate Memory

2.6.3 Mesh Generation - Poisson PDE Equation Solution

In the case of Structured Mesh Generation the `MathematicalProblem` the unknown quantities are the natural coordinates of the internal nodes and they are calculated by solving Laplace or Poison PDE in the parametric space. The numerical simulation is orchestrated by the `MeshFactory::buildMesh` function :

- The `MathematicalProblem` is defined as follows :
  - The contravariant metric tensor components of each node will be imposed as `SecondOrderLinearPDEProperties`
  - The boundary shape is defined by the `DomainBoundaryFactory` class. The value of the Natural coordinates at the boundaries will act as Dirichlet conditions for the Fixed DOF during the simulation. The available boundary shapes are:
    - `parallelogram` : Parallelogram with input step size, rotation and shear angles.
    - `ellipse` : Ellipse with input a, b.
    - `annulus` : Annulus / Ampitheatre with input inner and outer radii as well as start and end corners
    - `cavityBot` : Parallelogram with input step size, and half-circle cavity on the Bottom boundary.
    - `gasTankHorizontal` : Parallelogram with input step size, and half-circle cavities on the Right and Left Boundaries
    - `sinusRiver` : Top and Bottom boundaries are shapes as sinus curves with the same input A and ω varying by an input distance

- The analysis is performed on the parametric space so all the information regarding the nodal coorinates is provided in the initialized `Mesh`

- The `buildMesh` function checks if the boundary conditions have been initialized, creates a `SteadyStateMathematicalProblem` object which includes the PDE, the boundary conditions, and the DOF, and sets up a solver for this mathematical problem.
  A `SteadyStateFiniteDifferenceAnalysis` object is then created. The constructor takes the aformentioned `SteadyStateMathematicalProblem`, `Mesh` and user defined `Solver`, `SchemeSpecs`, and Parametric coordinate system as input.

- When `SteadyStateFiniteDifferenceAnalysis` is created it calls `LinearSystemAnalyzer`. The class constructor accepts several pointers to objects including AnalysisDegreesOfFreedom, Mesh, MathematicalProblem, FDSchemeSpecs, and a CoordinateType enum. The constructor subsequently initializes a number of member variables and data structures, including the right-hand side vector, the system matrix, and a map that correlates parametric coordinates with their corresponding nodes in the mesh. The class contains the following functions:
  - `_initiatePositionsAndPointsMap()`: This function initializes a map to keep track of positions and points corresponding to different derivative orders and directions.
  - `_getQualifiedFromAvailable()`: This function returns the positions and points that satisfy template conditions from a pool of available positions and points.
  - `_getPDECoefficient()`: This function retrieves the partial differential equation coefficient for a given derivative order, node, and direction.
  - `createLinearSystem()`: This function constructs a finite difference scheme builder, and defines template positions and points that are needed for all scheme types (forward etc) that correspond to the input error order. It then iterates over all degrees of freedom, defining the node, identifying the maximum number of points needed for a scheme with consistent order and calculates the and PDE coefficients at the node. Next, it marches through all the non-zero derivative orders. For each order and each spatial direction, the method checks available positions, filters the node graph, calculates colinear coordinates and degrees of freedom and finally and calculates scheme weights. The Linear System components are calculated as follows:

    * The loop iterates over all colinear degrees of freedom (colinearDOF). Colinear DOFs are those which are along the same isoparametric line in the particular spatial direction being considered.
    * For each colinear DOF, it identifies the neighboring DOF (neighbourDOF).
    * It calculates a 'weight' which is the product of the scheme weight for that DOF (schemeWeights[iDof]), the coefficient of the i-th derivative of the PDE (iThDerivativePDECoefficient), and the reciprocal of the denominator. This weight represents the contribution of the neighbor node to the specific entry of the matrix or vector being assembled.
    * Depending on the constraint type of the neighboring DOF, it contributes differently to the matrix or vector:
        - If the neighbor DOF is free, it contributes to the matrix. Specifically, the contribution is added to the matrix entry corresponding to the current DOF (thisDOFPosition) and the neighbor DOF (neighbourDOFPosition). The position is based on the global identification of DOF.
        - If the neighbor DOF is fixed, it contributes to the right-hand side vector. The contribution, termed as 'dirichletContribution', is calculated as the product of the fixed value of the DOF and the weight. This contribution is subtracted from the vector entry corresponding to the examined Free DOF, leading to proper implementation of the Dirichlet boundary conditions into the Linear System
    * A new `LinearSystem` instance is created with the constructed matrix and rhs

- `buildMesh` calls the `solve` function. LUP Decomposition is performed on the matrix (The result is stored in the already existing matrix), and the linear system is solved with backward and forward substitution.

- The solution of the system is applied to the nodes as their corresponding Natural Coordinates vector.

- Analysis components except for the mesh are deallocated.

- Mesh is optionally stored in a VTK file.

3 Results

In the following section some exported meshes are showm: It should be noted that the code has been benchmarked with COMSOL MultiPhysics in the Steady State Laplace equation with 0% errors.

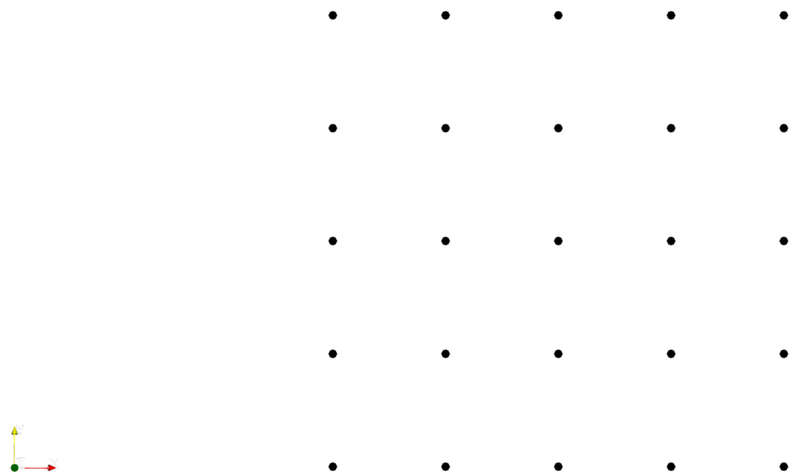

Figure 3: 5 x 5 Nodes, $L_x = 1$, $L_y = 1$

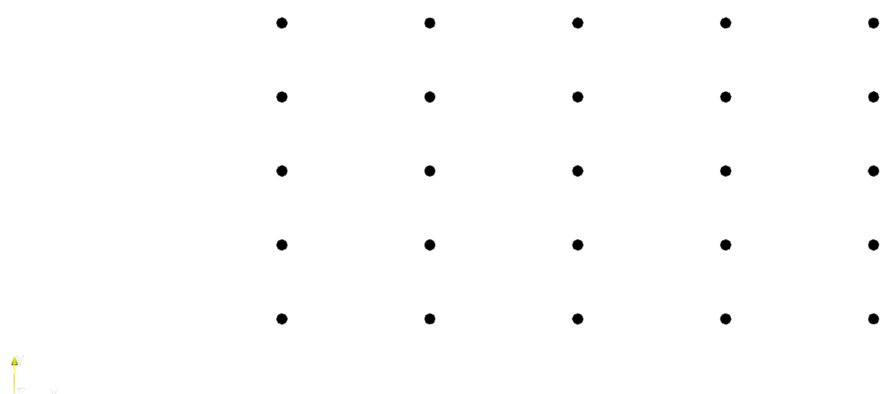

Figure 4: 5 x 5 Nodes, $L_x = 2$, $L_y = 1$

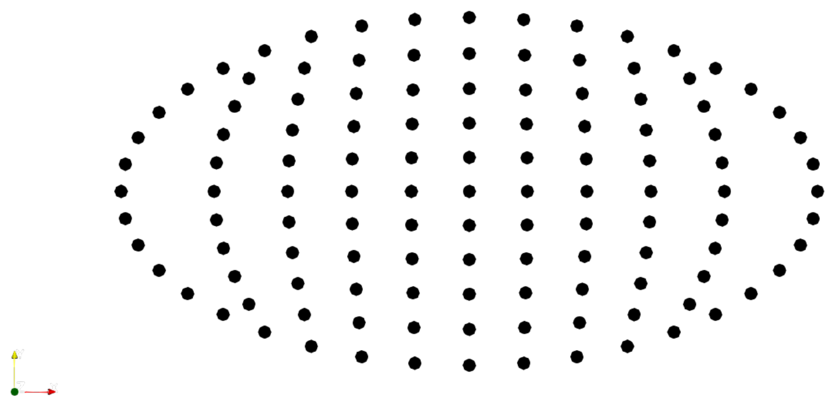

Figure 5: 11 x 11 Nodes, $\alpha = 2$, $\beta = 1$

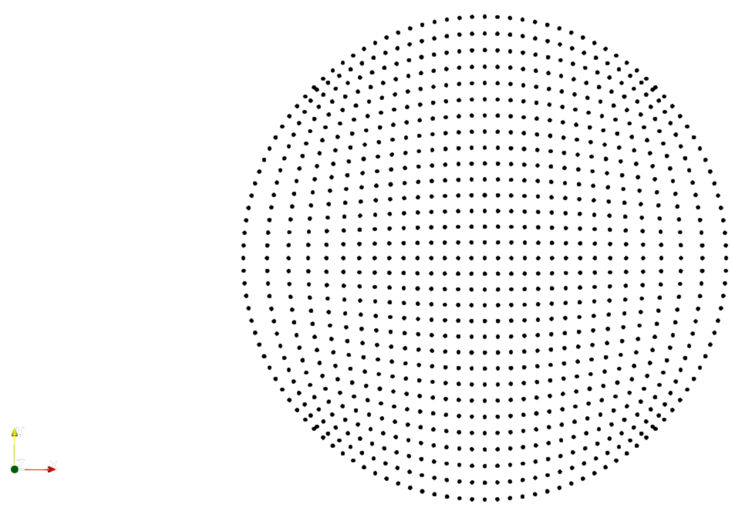

Figure 6: 31 x 31 Nodes, $\alpha = 1$, $\beta = 1$

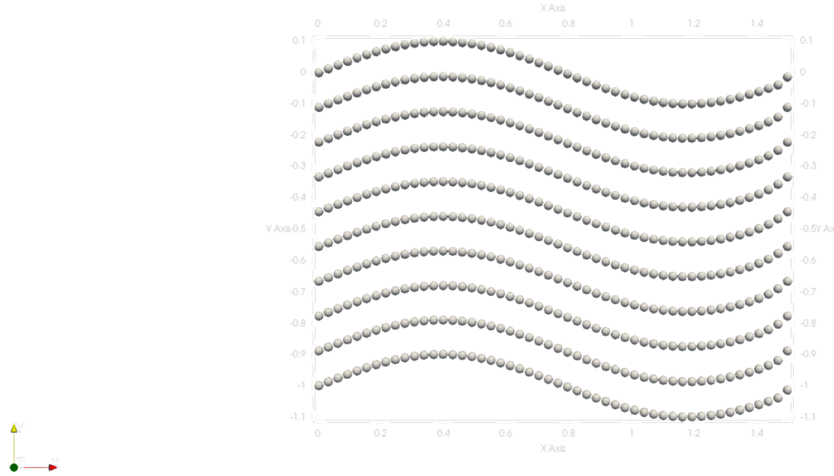

Figure 7: Windows Logo

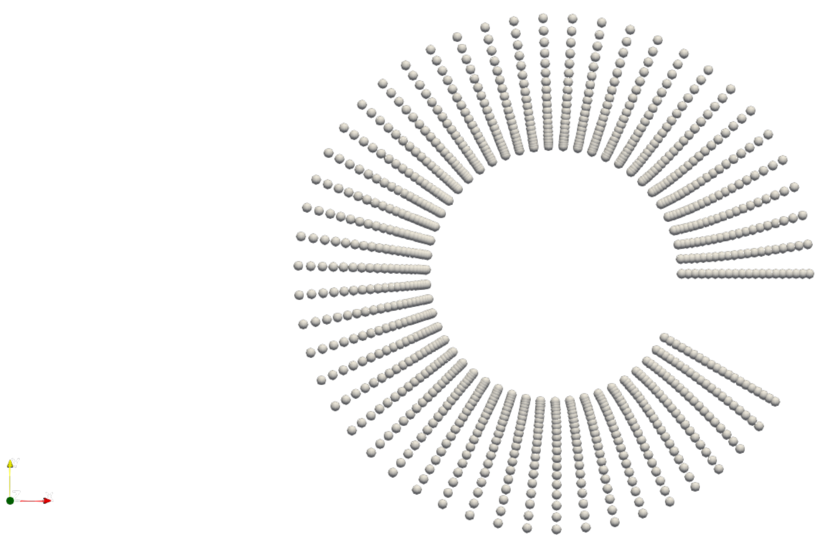

Figure 8: Annulus 51 x 21 Nodes, $r_{in} = 0.5$, $r_{in} = 1$