NEW HIT-SI GRID/GEOMETRY and NIMROD GRID INITIALIZATION

Cihan Akcay

October 29, 2009

MOTIVATION based on previous results

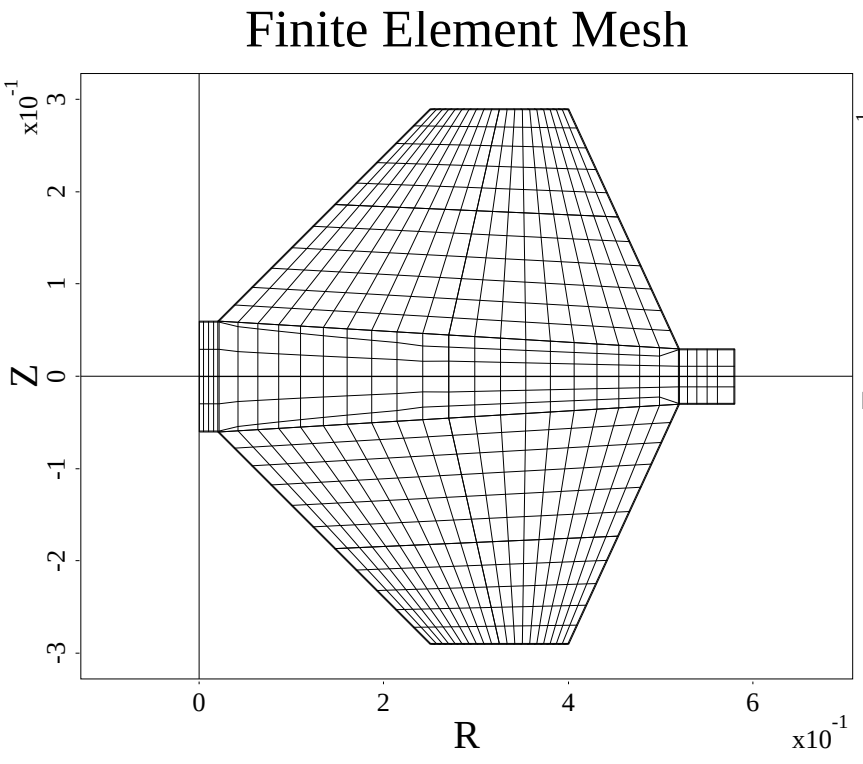
- Come up with a gridding algorithm that easily captures the curvature of the simulated geometry
- A grid packing algorithm different from the existing one.
- non-conforming grid makes the coding very cumbersome and post-processing difficult. nimfl never worked successfully on the old geometry and visIT grids have holes.

Outline

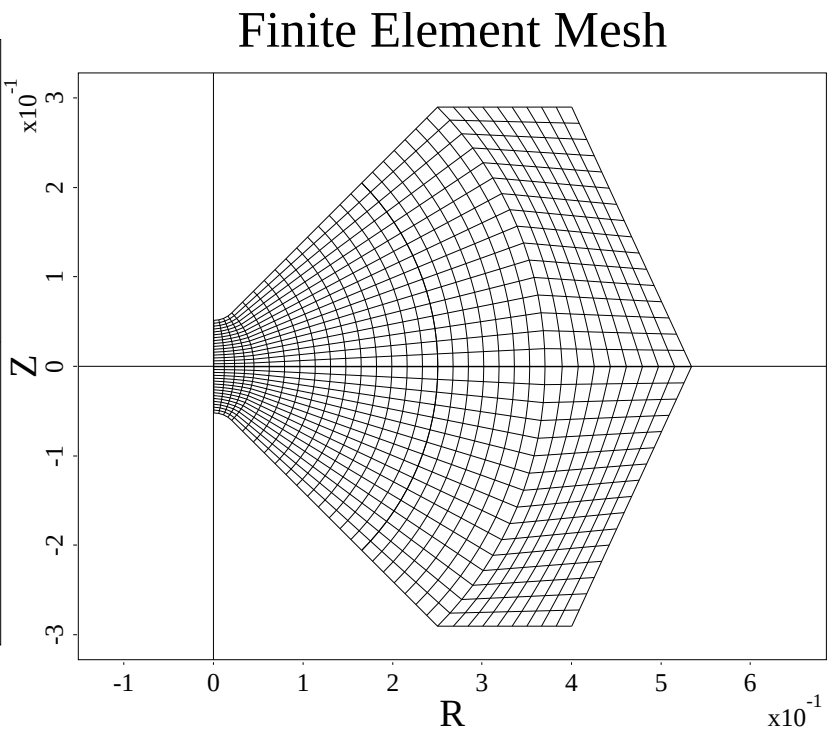
Highlights of the new algorithm

- The new grid initialization is designed to accommodate all types of geometries.
- The geometry is treated as a single contiguous block, very convenient for post-processing.
- There is a hierarchy of components and operations at each component level.
- This hierarchy gives us the flexibility needed to capture features like the curvature of the domain.

The inner cone tips are now rounded. The diagnostic gap is cut out $\Longrightarrow$ Conforming Geometry

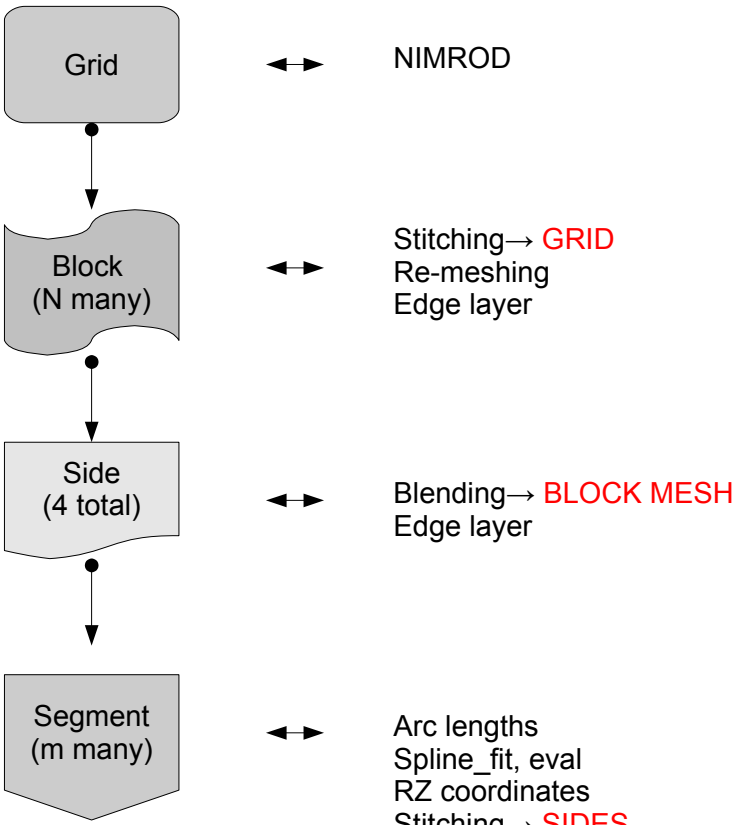


OLD GEOMETRY



NEW GEOMETRY

We have a hierarchy of four OBJECTS and a language associated with this hierarchy

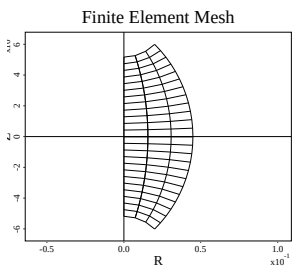


Basic Meshing Procedure

- The domain is decomposed into several blocks specified by the user.
- Create segments.
- Join segments together to form the sides.
- Blend the 4 sides to generate the blocks.
- Stitch blocks to lay out the final grid.

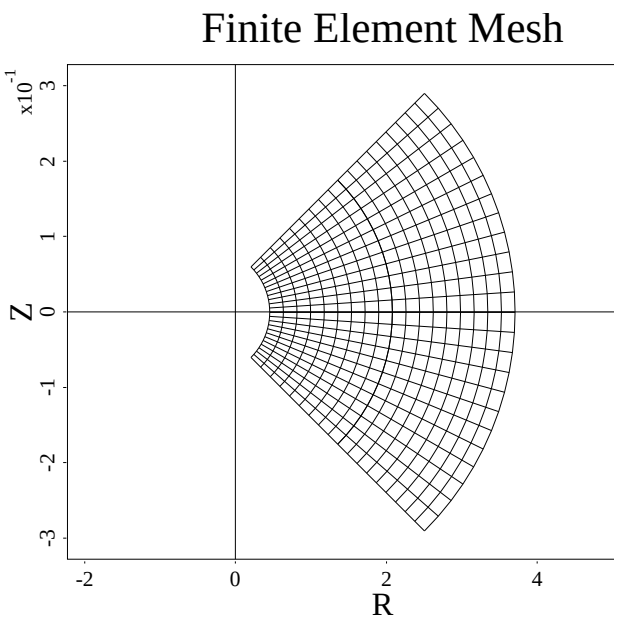
GRID: the global NIMROD lagrange_quad structure

- The grid can be decomposed into blocks specified by the user.
- Multi-block feature increases the versatility of the meshing algorithm as shown below: (note block 1 is blown up for illustrative purposes)



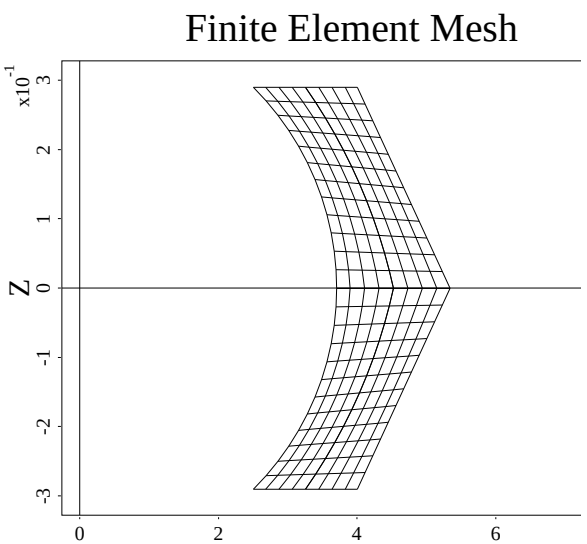
Block 1

+



Block 2

+



Block 3

BLOCKS: building-blocks of the grid

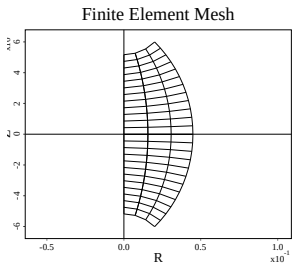
- have their own mesh: `block%grid`
- have user-assigned weights that determine how many cells a block should have along the direction in which the domain is split. This is the logical **x** for our example.

$$block\%mx = \frac{block\%weight * block\%length}{\text{total weighted edge length of the grid}} \quad (1)$$

```
TYPE :: block_type
        TYPE(side_type), DIMENSION(4) ::
        side
        REAL(r8), DIMENSION(:,:,:),
        ALLOCATABLE :: grid
        INTEGER(i4) :: mz, mx, my, id
        INTEGER(i4) :: weight
        REAL(r8) :: length
```

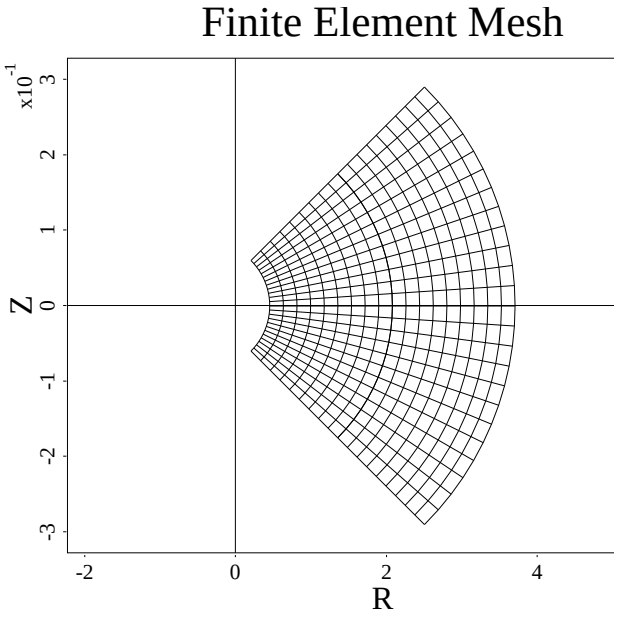
An example of block weighting

- Block 1 is much shorter in the horizontal direction than the other blocks. Thus, it is assigned a weight of 2 to increase the cell number along the short edge.



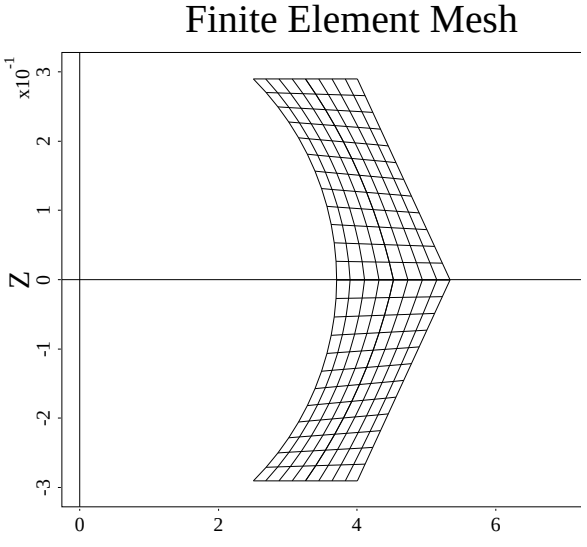
Block 1

+



Block 2

+



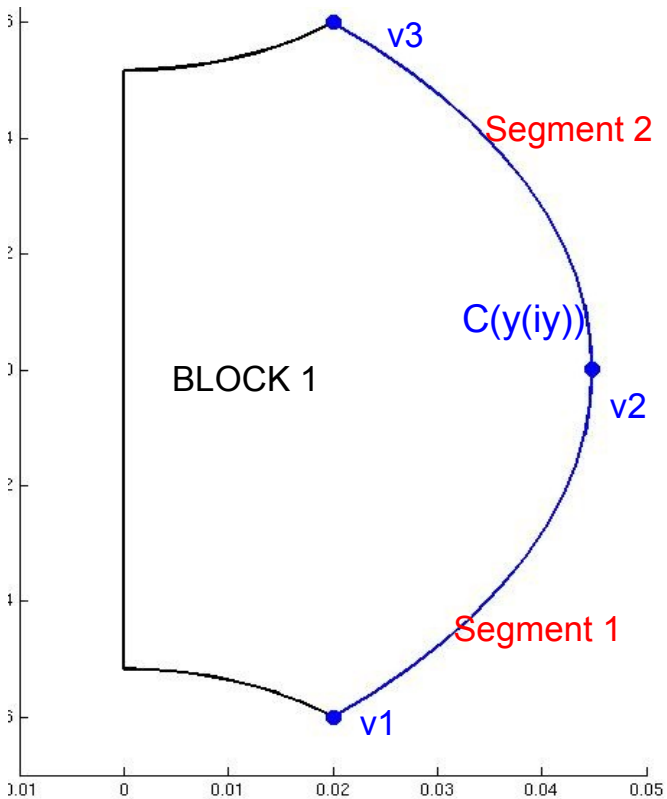
Block 3

SIDES: Components that make up the blocks

- Length and weigthed length of each side is stored.
- RZ variable stores the **computed** physical locations of the element vertices along a side.

```
TYPE ::  side_type
         TYPE(seg_type), DIMENSION(:),
         ALLOCATABLE ::  segment
         REAL(r8), DIMENSION(2,2) ::  vertx !
         end vertices of each side
         REAL(r8), DIMENSION(:,:), ALLOCATABLE
         ::  RZ
         REAL(r8) ::  length
         REAL(r8) ::  wlen
         INTEGER(r8) ::  n_vertex
         INTEGER(i4) ::  mz
```

Each side is a piecewise analytical or numerical assembly of segments.



- Sides are made of segments joined together at the ends called vertices (v1, v2 etc.). Vertex information is input into the code.
- Sub-division into segments is necessary to treat special points like
  1. Fixed locations such as the injector mouths (must pack around them)
  2. Vertices where curves defining the boundary change in slope
  3. Sometimes for convenience.

SEGMENT: the fundamental unit of the domain

- The entire construction is based on segments.
- Segment construction needs the following inputs:
  - RZ coordinates of the vertices (segment ends)
  - segment shape (‘line’, ‘circle’ or ‘numeric’ etc.)
  - other parameters such as the location of the radius of curvature for a circular segment.
  - segment weight, to ensure there is a sufficient number samples along the segment

```
TYPE ::  seg_type
          TYPE(spline_type) ::  hit_spl
          REAL(r8), DIMENSION(2) ::
          vertx1,vertx2,coord
          REAL(r8) ::  seg_length , radius, slope,
          slice
          CHARACTER(16) ::  seg_shape
          INTEGER(i4) ::weight ! weight for each segment
```

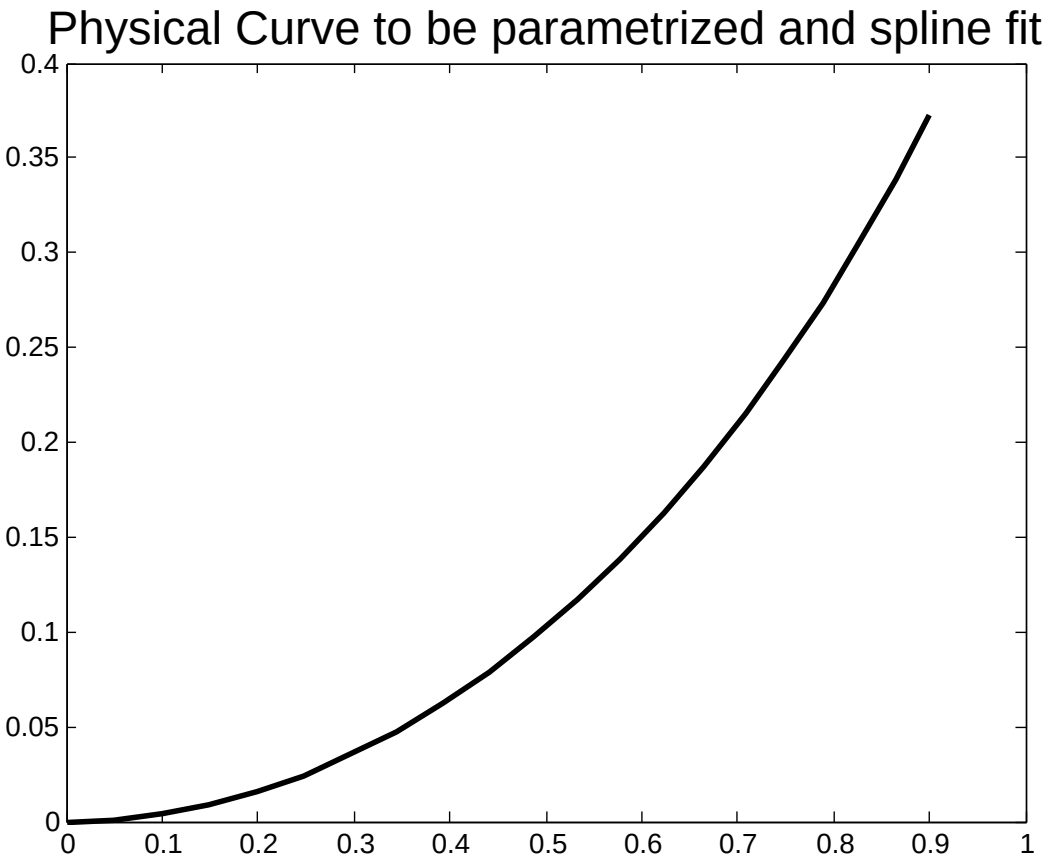
CONSTRUCTION and THE PROCEDURE

Overview of the meshing procedure

1. Input.txt is read in. It contains the following data: Number of blocks, block weights, number of vertices along each side within each block, vertex coordinates, segment shapes, segment weights.
2. Domain decomposition into blocks.
3. Segments are parameterized, then spline fit and evaluated to obtain actual RZ locations along each segment.
4. Segments are joined together to form the sides.
5. 4 sides are blended to generate the blocks (mesh)
6. Blocks are stitched together to lay out the final grid
7. Special operations such as creating the edge layer
8. Finite element nodes

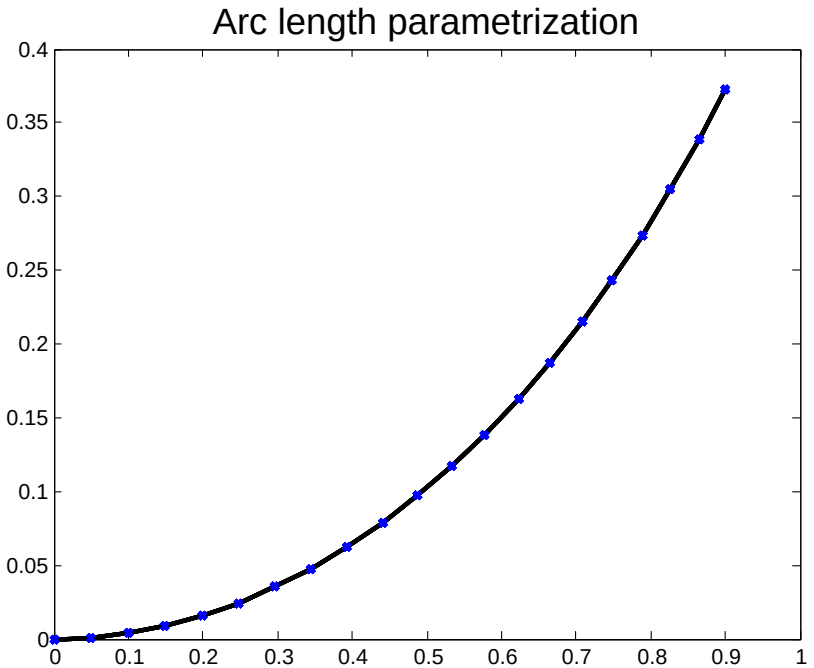
We begin with an analytically defined segment

The arclength, ℓ and weigthed arclength, ℓw of the segment are calculated



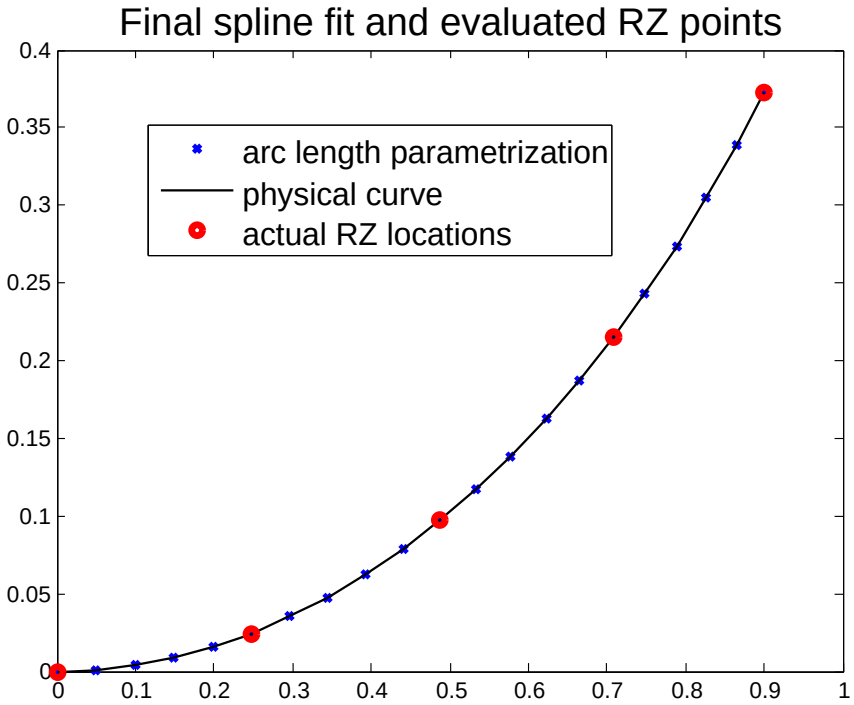
Next the segment is parameterized in terms of its arclength

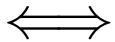
- The arclength is split into 100 equally spaced intervals: s$\in$[0,..,100] where ℓ(s=0)=0 and ℓ(s=100)=arclength
- ℓ(s) returns the local length for the s'th sample

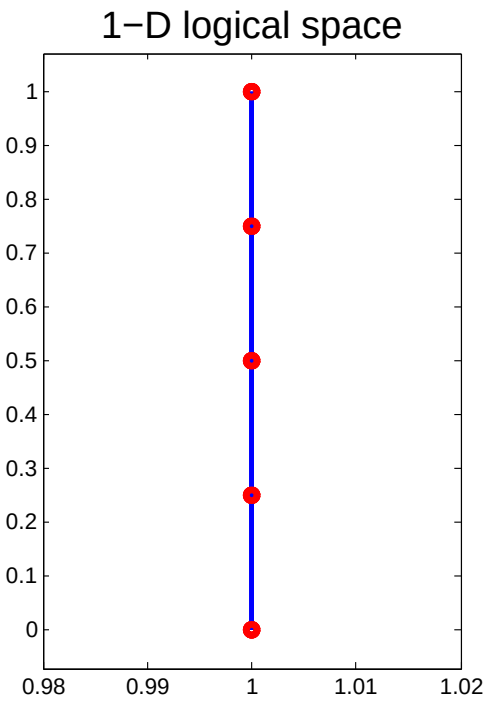


- R $\rightarrow$ R(ℓ)=R(ℓ(s)), returns the physical coordinate corresponding to s.
- SIZE(R)=101 so it is **oversampled**.
- Reduce the sample size $\rightarrow$ interpolate via. spline_fit and evaluate

Spline fit and evaluate the parameterized curve to extract the actual RZ locations along the curve



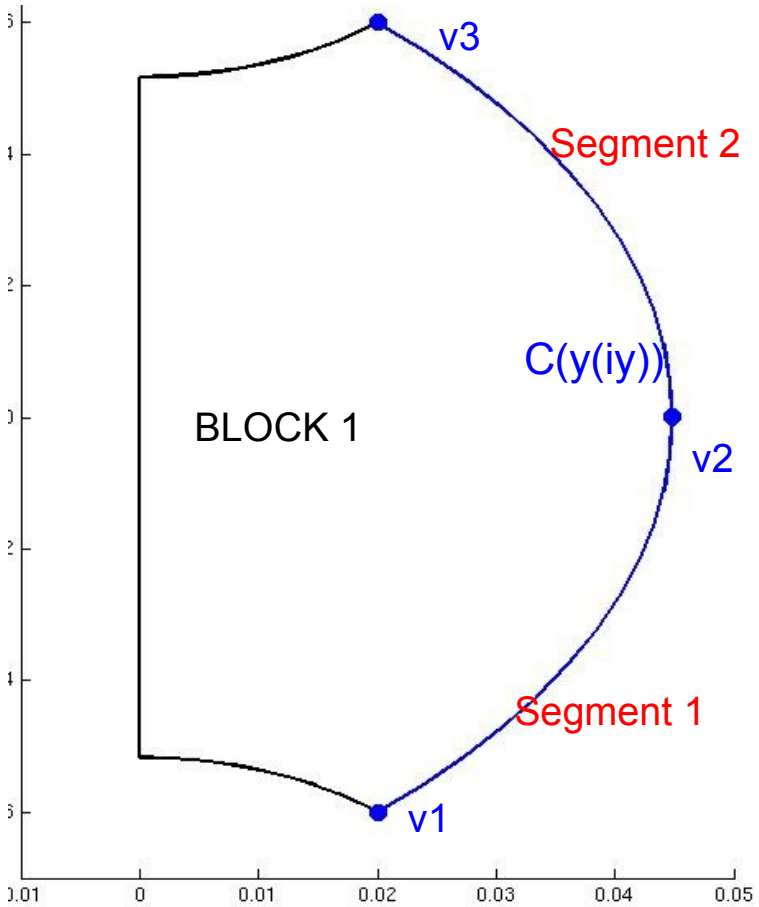




In essence, this is a 1-1 mapping from the arclength space (LEFT) into the logical space (RIGHT). This 1-1 correspondence is critical and always true

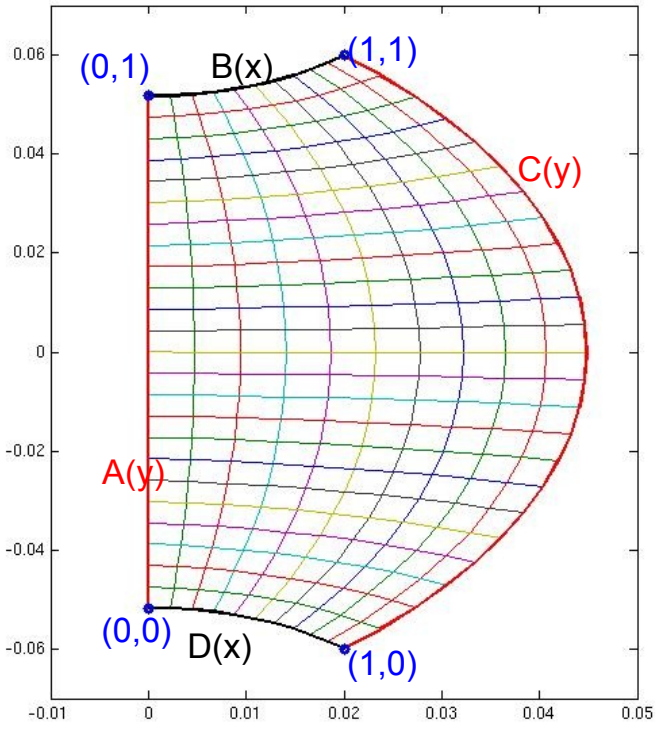
Sides are formed by joining the segments together at the end points

Segments 1 and 2 joined at vertex v2 results in side C(y(iy))



block grid is constructed by ‘blending’ the four bounding sides

- The ‘blending’ is the crucial piece of the whole algorithm.
- The four sides denoted A(y), B(x), C(y) and D(x) have a 1-1 correspondence with a logical rectangle bound by [0,1]×[0,1]
- The ‘blending’ algorithm guarantees block meshes at block boundaries match.



BLOCK 1

The mesh is generated by 'blending' the 4 sides

The Blending algorithm uses a **Boolean Sum Interpolation**

$$\begin{aligned} grid(:,ix,iy) = {} & A(iy) * (1 - x(ix)) + B(ix) * y(iy) + \\ & C(iy) * x(ix) + D(ix) * (1 - y(iy)) \\ & - A(0) * (1 - x(ix)) * (1 - y(iy)) - B(0) * (1 - x(ix)) * y(iy) \\ & - C(end) * x(ix) * y(iy) - D(end) * x(ix) * (1 - y(iy)) \end{aligned} \quad (2)$$

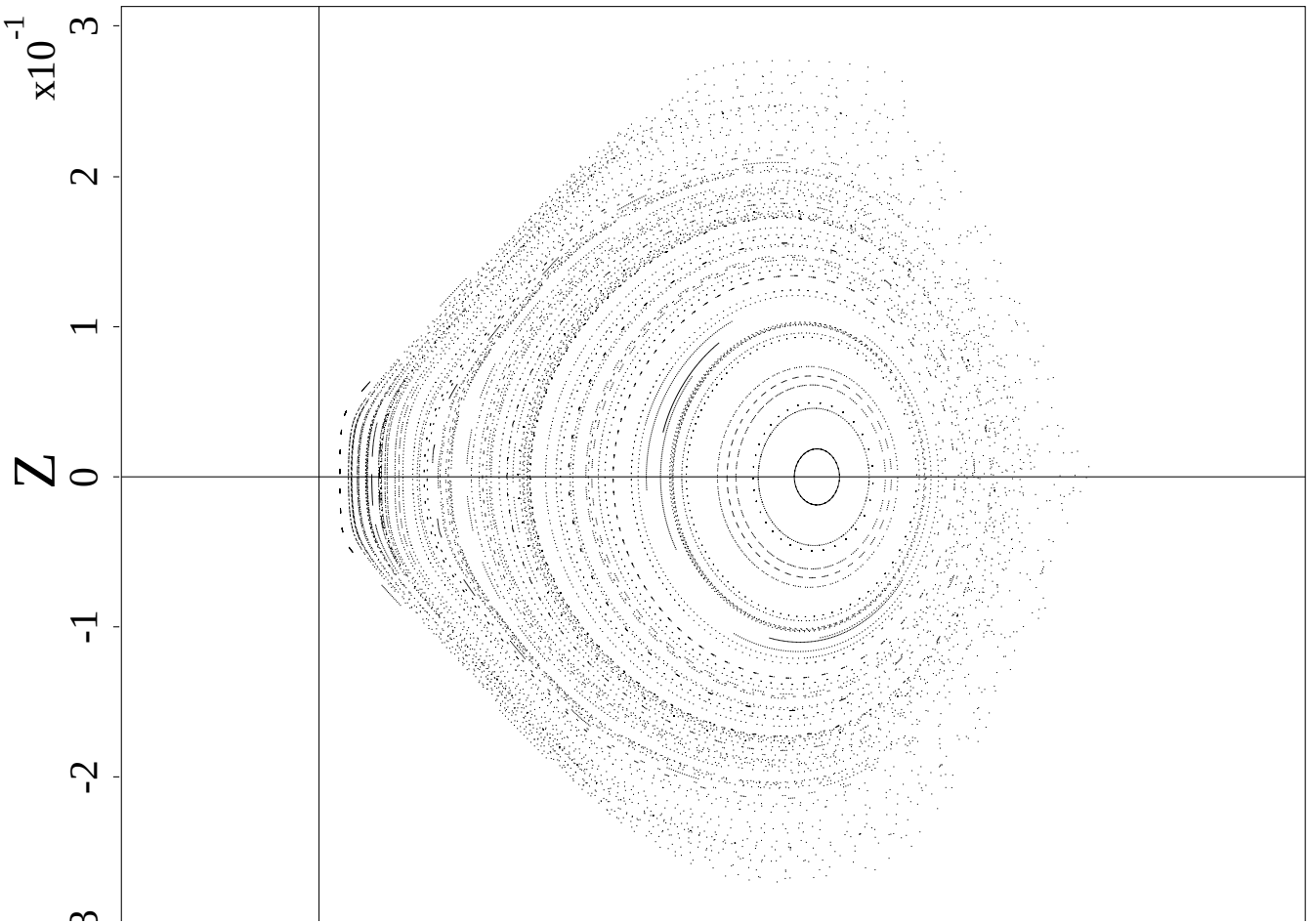
- This maps each one of the sides to a surface.
- The need for 1-1 correspondence becomes clear
- The terms in blue are corrections for the overcount due each corner getting mapped twice.
- Thus, the algorithm solely depends on the correct parameterization of the 4 sides.

And finally we stitch the blocks together to create the final NIMROD grid

- Blocks are stitched side-by-side or tiled vertically.
- Stitching requires that the mesh on either side of a block boundary match
- The ‘blending’ algorithm guarantees this
  - as the mesh always returns the side coordinates for boundary values by construction
  - and adjacent blocks have a common side by definition

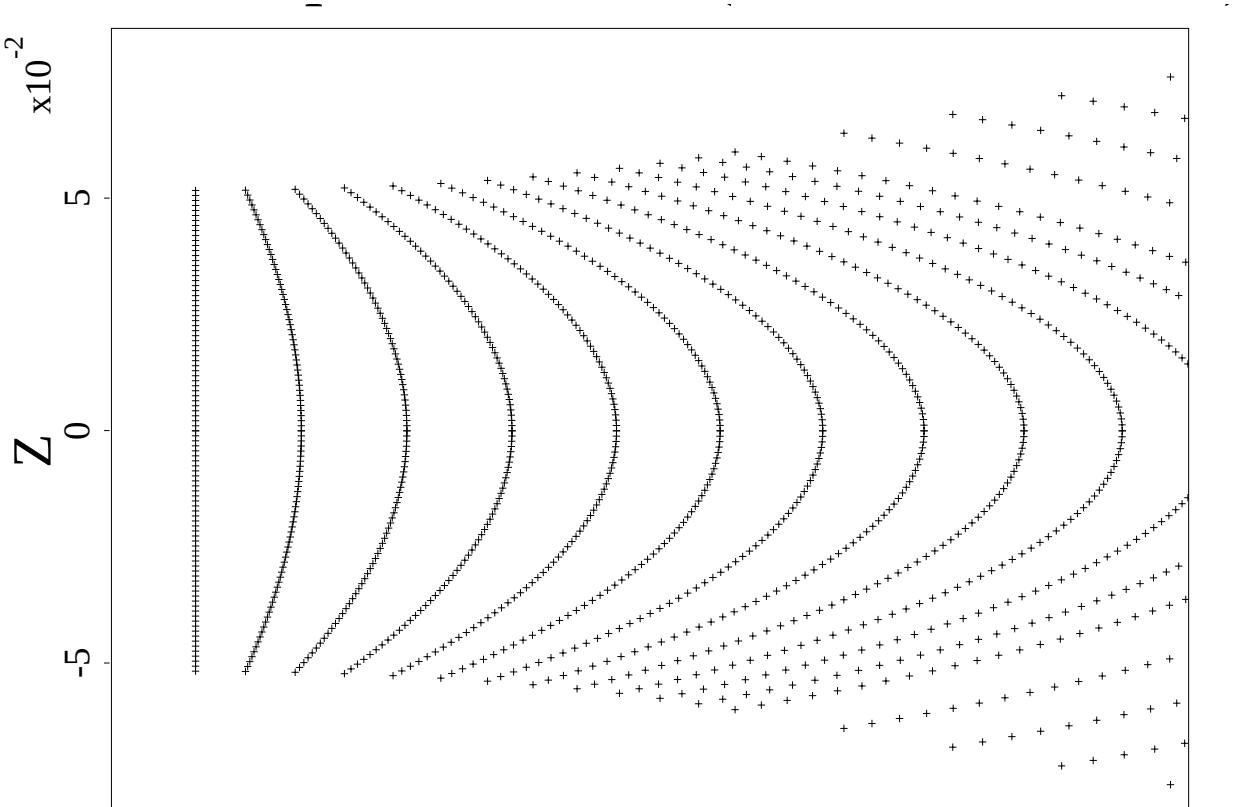
Poincare Plots are now easily generated

Poincare surface plot of a decaying Taylor state axisymmetric spheromak



NODES-example with pd=3

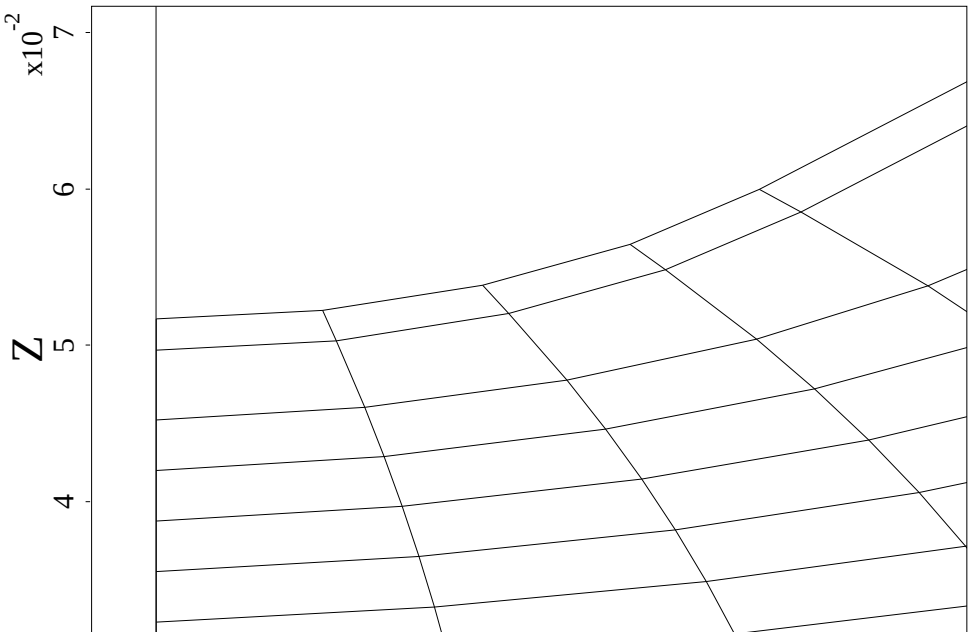
- The nodes are laid out in a fashion identical to the grid except we now have to increase our total sample number in arclength and logical space to accomodate for the increase in total number of points.



Special functions - Edge layer

Edge layer has been successfully incorporated.

- Second to last points to all external sides within each block are shifted to locations that are 'ethick' away from the wall.
- The points are shifted in such way as to retain the cell-to-cell C1 continuity of the original grid.



CONCLUSION & WORK IN PROGRESS

- The new grid initialization successfully generates the new HIT-SI geometry and some other basic geometries.
- The code can now handle curved geometries with no special lines of coding.
- Internal nodes are laid out in exactly the same fashion as the grid.
- We are currently working on two special operations that we will need when the edge layer is present: ‘re-meshing’ and packing algorithms

NODES

Finite element nodes are laid in a fashion identical to the grid itself. In this case, we subsample the intervals along each arclength by (pd-1). So between ℓ(s) and ℓ(s+1), we now have $\ell(s+\frac{\Delta\ell}{pd-1})$, $\ell(s+\frac{2\Delta\ell}{pd-1})$, ... $\ell(s+\frac{j\Delta\ell}{pd-1})$,... where j runs from (1, pd-1) and $\Delta\ell$=$\ell(s+1)-\ell(s)$

- $\ell(s+\frac{j\Delta\ell}{pd-1})$ is again mapped to a coordinate system: R($\ell(s+\frac{j\Delta\ell}{pd-1})$), which is again oversampled significantly. So R($\ell(s+\frac{j\Delta\ell}{pd-1})$) is processed through the same spline functions to reduce the sample size to mx*(pd-1)
- We also perform a corresponding subsampling in the logical space: x(ix), x(ix+$\frac{\Delta x}{pd-1}$), x(ix+2 $*$ $\frac{\Delta x}{pd-1}$),... x(ix+$\frac{j\Delta x}{pd-1}$) where $\Delta x = x(ix+1)-x(ix)$