Chapter 1

Grid Implementations for *Gorilla*

The developed *Gorilla*-code (Geometric ORbit Integration with Local Linearisation Approach) is a geometric guiding center orbit integrator written in modern `Fortran` based on local linearization of electro-magnetic field quantities. Due to the linearization of these fields, a grid must be implemented consisting of tetrahedra where within each tetrahedral grid element the linearization is performed. In this chapter the two grid implementions used by *Gorilla* are explained. For compatibility reasons, the codes for the grids are also written in `Fortran`.

1.1 Requirements and structure

There exist several requirements that must hold for any given grid in order for the executing code to function correctly. These requirements are:

1. The three-dimensional spatial domain, which is relevant for calculations, must be fully covered by non-overlapping tetrahedra. In this application tetrahedra are necessary since field quantities are used in a piece-wise linearized form, meaning they are saved as a scalar value at a reference point and a corresponding gradient of the quantity. Such a linear representation has four independent parameters, therefore, the use of tetrahedra is ideal since the parameters of the linearized field quantity can be exactly defined via the field quantities at the four vertices of the tetrahedron. One might think that apart from tetrahedra other spatial objects with more vertices can still be used by fitting a linear function of the field quantity, however, such an application would destroy the important property that the field quantities through adjacent faces of tetrahedra are continuous. Furthermore, for vector quantities each vector component is independently linearized.

2. All edges of tetrahedra must coincide with edges of neighboring tetrahedra. Edges that lie on faces of neighboring tetrahedra (these are called *hanging nodes*) or crossings between edges are not permitted. This requirement is given

by the continuity condition of linearized field quantities through the faces of each tetrahedron and as well by the requirements for Maxwell solvers, which will be used to calculate electromagnetic field contributions.

3. Each tetrahedron must be defined via four corner vertices in a given coordinate system. The coordinate values of each vertex are stored in an array and each vertex is identified by its array index. The index of the four vertices belonging to a specific tetrahedron has to be stored in a 4×1 array and can be accessed by indices 1 to 4 within each tetrahedron.

4. The tetrahedra are stored in an array of tetrahedron objects and identified by their index. Each tetrahedron has four defined faces labeled face 1 to 4. Each face i ($i = 1, 2, 3, 4$) is spanned by the vertices of the tetrahedron excluding tetrahedron-vertex i. For instance, face 3 will be spanned by tetrahedron-vertices 1,2,4. This implies that the tetrahedron-vertex i will be the only tetrahedron-vertex not lying on face i.

5. For a given tetrahedron, the neighboring tetrahedron which is separated by the i-th face of the current tetrahedron will be considered the neighbor i to the current tetrahedron with its global labeling index being saved in the i-th position of a 4×1 array.

6. In addition to saving the four faces and neighbors of each tetrahedron, the index of the intersecting face between the original tetrahedron and its neighbor in the index system of the neighbor will be saved with the original tetrahedron. This means that the face through which a particle enters a neighbouring tetrahedron can be determined by knowing through which face it is leaving the current one.

7. Tetrahedra at the outer boundary of the grid will not have neighboring tetrahedra at the boundary face, the neighboring tetrahedron index as well as the index of the face in the index system of the neighboring tetrahedron will be set to -1.

8. The normal vectors corresponding to each face of all tetrahedra must be explicitly calculated and saved together with a reference point. This enables the calculation of normal distances of any arbitrary point to all faces of each tetrahedron.

9. Grids that are made in a coordinate system with a periodic coordinate need to have an additional property set for tetrahedra faces lying at the periodic boundary, depending on which side the face lies. This determines in which direction the coordinate needs to be shifted when the particle passes through the boundary.

1.2 Cylindrical contour Grid

The first implementation of a grid suitable for the *Gorilla* code is the so-called 'cylindrical contour grid' which is generated by the subroutine `make_grid_rect`. This grid is generated in cylindrical coordinates and has uniformly distributed vertices along the coordinate contours R, Z and φ.
In this section the individual procedures and approaches to generating the required grid quantities will be discussed. To clarify, from a programmatic aspect, tetrahedra can here be thought of as instances of a tetrahedron class with a set of properties, such as vertex indices, neigbour indices, neighbor entry face indices, etc. Therefore, any relevant information can be directly saved together with the tetrahedra. Furthermore, tetrahedra properties are saved with the attributes `public, protected` and are thus only permitted to be altered by the grid generating function, assuming the role of a constructor in this context.

1.2.1 Generating the vertices

The first step in implementing this grid is to generate the vertices that will define the corner points for the tetrahedra filling a given space. In order to do this, the domain which will be covered by the grid needs to be specified in the coordinate system where the grid will be generated. In this case, the grid will have vertices equidistantly spaced in R, φ and Z direction with intervals for R and Z being $[R_{min}, R_{max}], [Z_{min}, Z_{max}]$ and $[0, 2\pi]$ for φ, respectively. Now, each coordinate x_i will be discretized into N_i equidistant values for each given interval. Using nested loops, these discretized values will be connected to $N_R \times N_\varphi \times N_Z$ unique triples representing the coordinates of the individual vertices of the grid. Upon generation of the vertices, an incrementer will label each individual vertex with an integer, starting with 1 for the first triple with coordinates$(R, \varphi, Z) = (R_{min}, 0, Z_{min})$. The order of labeling the generated vertices will be defined by the order of the nested loops of the coordinates, in this case being Z, R, then φ.
An important aspect to note is the treatment of periodic boundary conditions. Depending on the coordinate system there might be periodic coordinates, which in this case arise at the φ-coordinate. As mentioned above, the interval given for φ is the closed interval $[0, 2\pi]$. In principle this is not exact as the coordinate value of $\varphi = 2\pi$ is already represented by $\varphi = 0$, therefore the interval should be semi-open $[0, 2\pi)$. However, later for the calculation of normal vectors of each tetrahedron, all coordinates must be within the same period, therefore points lying on the $\varphi = 0$-plane must both act as points lying on $\varphi = 0$ as well $\varphi = 2\pi$, depending on which side

of the boundary the tetrahedron containing the point lies. To avoid confusion, for this grid there are separate vertices with distinct indices for $\varphi = 0$ and $\varphi = 2\pi$ even though they should be topologically identical.

1.2.2 Get correct vertices of tetrahedra

So far, a regular grid of vertices has been implemented without any connections between vertices. In the next step of the grid generation, the very convenient property of the vertex grid that vertices can be easily assigned to tetrahedra via indexing will be used. This procedure looks as follows:
Since the grid is regular (equidistant spacing between all points) 3D-integer coordinates (i, j, k) can be introduced for all points, with the basis vectors being the discretization step sizes times the unit vectors of the cylindrical coordinates (R, φ, Z). Using this, neighboring vertices can easily be connected to form hexahedra. The eight corner points for such hexahedra will have the coordinates $[(i, j, k), (i + 1, j, k), (i + 1, j + 1, k), (i, j + 1, k), (i, j, k + 1), (i + 1, j, k + 1), (i + 1, j + 1, k + 1), (i, j + 1, k + 1)]$. These coordinates are simply used for practical purposes and can easily be converted to the vertex integer label $ind(i, j, k)$ using the following formula:

$$ind(i, j, k) = k + (i - 1) \cdot N_Z + (j - 1) \cdot N_Z \cdot N_R \tag{1.1}$$

After the eight points are connected to form a hexahedron, this hexahedron is then split up into two prisms which are subsequently and independently split up into three tetrahedra. The precise way, how the tetrahedra fit into the hexahedron can be seen in figure 1.1.
Here, the plane containing points $(2, 4, 6, 8)$ cuts the hexahedron in half and therefore splits it up into two symmetric prims. The first prism is then split up into three tetrahedra with the corner points $(1, 2, 4, 5), (2, 4, 5, 6)$ and $(4, 5, 6, 8)$. The point indices of the tetrahedra of the second prism are $(2, 3, 4, 6), (3, 4, 6, 8)$ and $(3, 6, 7, 8)$, respectively.
For each direction (R, φ, Z) the two faces with respect to that direction have the same orientation of the diagonal intersection of the faces. This means, that these hexahedra can be stacked next to each other in any direction without crossing edges of tetrahedra, as long as the orientations of all hexahedra are the same.
The indices of all vertices for each tetrahedron can be retrieved by iterating over all hexahedra within the domain using loops over i, j and k, transforming the corner points into integer labels using formula 1.1 and then saving the correct corner point indices for each of the six types of tetrahedra. Upon generation of each tetrahedron a separate

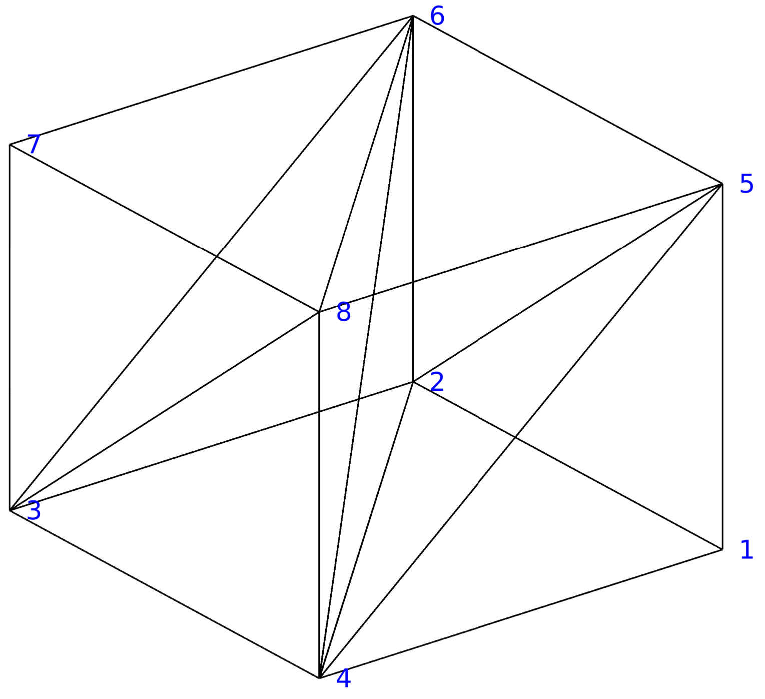


Figure 1.1: Hexahedron split up into six tetrahedra

counter that labels the tetrahedra with an integer label needs to be incremented.
By now, all vertices of the grid have been generated, also the vertices have been properly 'connected' to form tetrahedra that completely fill the domain of the grid.

1.2.3 Get neighbors of tetrahedra

In this section the procedure of finding the indices of neighboring tetrahedra with respect to a given tetrahedron will be explained.
Each tetrahedron can be seen as a part of a hexahedron at position (i, j, k). The indices belonging to the corner points of such a hexahedron are given in section 1.2.2 using 1.1 while the index of the corner point (i, j, k) can be interpreted as the index of the hexahedron at hand. Since the tetrahedra were labeled in the same order as the hexahedra, there is a simple formula to index all six tetrahedra belonging to a given hexahedron at position (i, j, k):

$$ind_{\text{tetra}}(i,j,k,l) = 6(ind_{\text{hexa}}(i,j,k) - 1) + l = \tag{1.2}$$
$$= 6(k - 1 + (i-1) \cdot N_Z + (j-1) \cdot N_Z \cdot N_R) + l$$
$$l = 1, 2, .., 6 \tag{1.3}$$

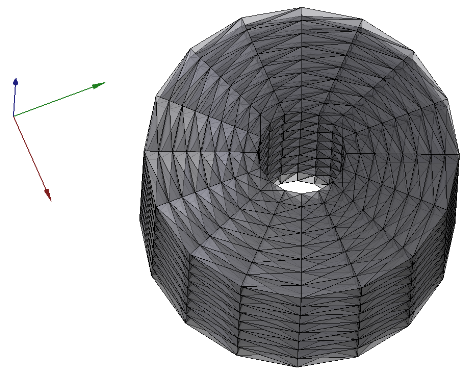
All neighboring tetrahedra must either lie within the same hexahedron as the reference tetrahedron or within a neighboring hexahedron. For this reason it is convenient to use formula 1.2 to obtain the indices of all tetrahedra that are within the same or the neighboring hexahedra. One can then loop over all faces of all tetrahedra and compare the indices of the vertices at the given face of a tetrahedron with the vertices of the faces of all potential neighbors. If three vertices coincide, a matching face and therefore the neighbor to the given face is found. Using this procedure, all tetrahedra can be efficiently connected, by setting the default value for the neighbor indices to -1, all border tetrahedra that have no matching neighbor to a face will automatically have set the correct value, as defined in the requirements for the grid.

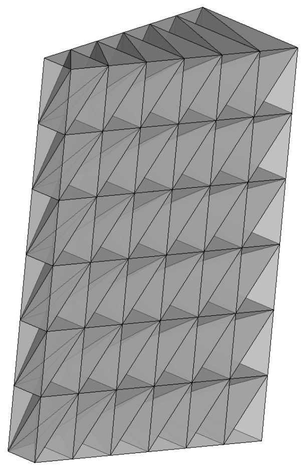
1.2.4 Periodic boundary conditions

Furthermore, since this grid is constructed using cylindrical coordinates, periodicity in the φ coordinate occurs which needs to be treated independently. The issue is that vertices lying on the $\varphi = 0$ plane have both values 0 and 2π in the φ component and particles that move through the boundary experience a jump in coordinate. Furthermore, since normal vectors must be computed from corner vertices and field quantities are later linearized within the grid, coordinates of corner vertices of all tetrahedra must be smooth with respect to each other and not experience such discontinuities. A possible solution to this problem is to introduce an additional set of vertices, where each element corresponds to a vertex on the $\varphi = 0$ plane but with the φ component being shifted to 2π. Neighbor indices can still be obtained by proper indexing, a drawback of this approach is however, that vertices at the periodic boundary plane have different indices on both sides while actually being the same vertex in real space. In case that such information is relevant, this needs to be taken into account independently. While this approach is a working solution, for further grid implementations the approach was changed to not have two indices for any existing vertex. However, here the jump in coordinate needs to be detected upon computation of normal vectors.

1.2.5 Grid visualization

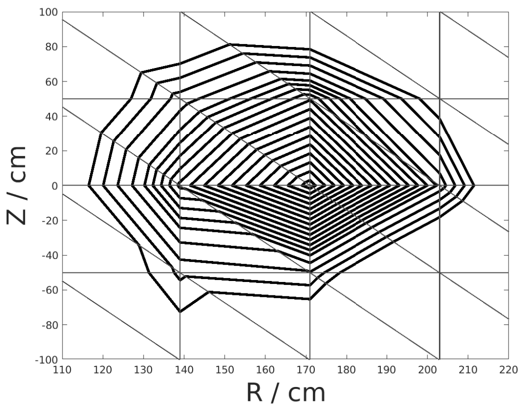
A full 3D representation of a rather coarse cylindrical contour grid with grid size $(N_R, N_\varphi, N_Z) = (8, 16, 8)$ is given in figure 1.2a. Axis orientations are given by the arrows on the top left with the blue arrow indicating the symmetry direction Z. Furthermore, one can see that the center region is not part of the grid, this is simply due to range restrictions in R direction as this region is anyway obstructed by the solenoid in a real tokamak and therefore irrelevant for particle guiding center motion. Figure 1.2b shows additionally a single φ slice of the cylindrical contour grid for better imaginability of intersections at tetrahedra boundaries.


(a) Cylindrical contour grid in real space

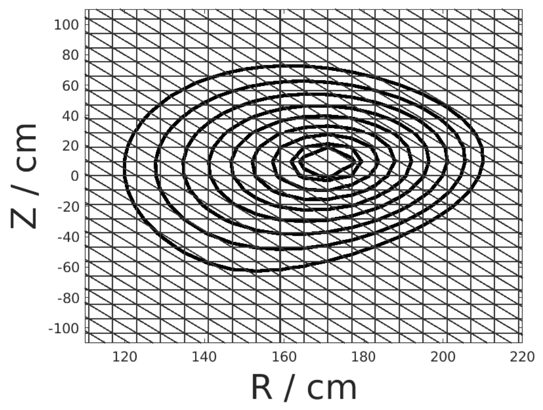

(b) Single φ slice of cylindrical contour grid in real space

1.2.6 Field lines in cylindrical contour grid using cylindrical coordinates

Using the cylindrical contour grid for computation of guiding center motion with *Gorilla* in cylindrical coordinates also allows to easily compute field lines by following electrons at very low energies (it is sufficient to use energies of 10^{-2} eV) and strong magnetic fields (magnetic field components are scaled by $1E5$) leading to miniscule larmor radii and a diminishing curvature drift, hence any drift motion becomes negligible. Such a particle will then accurately follow magnetic field lines. If one follows besaid particle for a long enough time given that the safety factor q_s (which is the ratio of number of toroidal turns and number of poloidal turns in a fusion device) assumes an irrational number, a continuous surface will be covered by a single field line. Such a surface is a so-called flux surface where both the poloidal and toroidal magnetic flux remain constant within the central hole of the torus and the toroidal cross section, respectively. One can furthermore intersect a given flux surface with the $\phi = 0$ plane to make a Poincaré plot. Such a plot, calculated using cylindrical coordinates, is given for the cyldrical contour grid in figure 1.3a and 1.3b:



(a) Poincaré cut of flux surfaces calculated with *Gorilla* using a coarse cylindrical contour grid of $6 \times 16 \times 6$



(b) Poincaré cut of flux surfaces calculated with *Gorilla* using a finer cylindrical contour grid of $32 \times 16 \times 32$

An important thing to note regarding these figures is that due to a linearization of field quantities performed by *Gorilla*, all flux surfaces assume polygonal shapes. This has to do with the fact, that the field lines in a toroidal configuration, which is used for fusion devices, are curved lines in real space and a linearization in cylindrical coordinates will always introduce an interpolation error leading to polygonally shaped flux surfaces and, thus, also polygonal guiding center orbits to 0^{th} order in larmor radius. One can reduce these effects by using a finer mesh, as shown in figure 1.3b, however this comes at the expense of a larger computational cost. Furthermore, a large drawback of polygonal field lines is that for 3D (non-axisymmetric) field configurations chaos will be introduced when trying to calculate guiding center orbits. If one wants to keep linearization of field quantities for simplicity of equations and, thus, performance reasons, a possible solution to this problem is given via appropriate coordinate transformations where field lines assume straight lines. The use of symmetry flux coordinates (SFC) satisfies such a condition [?], however, the cylindrical contour grid will no longer be guaranteed to not have overlaps between tetrahedra, when vertices are directly connected in SFC. Since not having overlaps was one of the initial requirements, a new grid with a more appropriate toroidal shape needs to be implemented for the use of SFC.

1.3 Field Aligned Grid

One is now interested in computing guiding center orbits in symmetry flux coordinates (SFC) where field lines are represented by straight lines. The straightness of field lines in these coordinates allows to compute guiding center motion also for generally non-axisymmetric field configurations without introducing artificial chaos. Furthermore, field lines calculated in SFC will coincide more accurately with physical field lines, compared to when calculated in cylindrical coordinates where they assume polygonal shapes. A problem that arises by using SFC is, however, that in SFC the cylindrical contour grid is no longer guaranteed to satisfy the requirement that tetrahedra must not overlap. Therefore, a new grid with a field aligned geometry is needed. Such a grid can be obtained by positioning the grid vertices equidistantly along the coordinate contours of $(s, \vartheta_f, \varphi)$, where s denotes the normalized flux label (normalized poloidal or toroidal flux), ϑ_f the symmetry flux poloidal angle and φ the toroidal angle, respectively. With such an approach, only a routine is needed that transforms any given point in SFC back to cylindrical coordinates, where physical field quantities are available. It should be furthermore pointed out, that any grid generated by setting equidistant points in three dimensions will be topologically identical to the cylindrical contour grid in cylindrical coordinates, thus an analogous indexing scheme for tetrahedra can be applied. In this section, an approach to obtaining a routine that converts given SFC coordinates to cylindrical coordinates for an axisymmetric field configuration is explained. A different code package for stellarator configurations has been made available by Sergei Kasilov and has been implemented in the *Gorilla* code. In this thesis only the axisymmetric approach will be discussed.

1.3.1 Field lines in toroidal fusion devices and safety factor

When trying to construct a field aligned grid, one must first look at the geometry of magnetic field lines themselves. By definition, magnetic field lines are curves of which the tangent is always parallel to the magnetic field vector [?]. Mathematically this translates to the set of differential equations

$$\frac{\mathrm{d}R}{\mathrm{d}\varphi} = \frac{B^R}{B^\varphi}, \; \frac{\mathrm{d}Z}{\mathrm{d}\varphi} = \frac{B^Z}{B^\varphi}, \tag{1.4}$$

where (R, φ, Z) denote cylindrical coordinates and (B^R, B^φ, B^Z) the contravariant components of the magnetic field. Important properties of field lines are that they always remain closed and cannot cross other field lines. Furthermore, the absolute strength of the magnetic field at a given point is proportional to the number of field

lines going through an infinitesimal area located at that point and perpendicular to the magnetic field vector, thus it is proportional to the areal field line density.
By numerical integration of set 1.4 over φ for some arbitrary starting position using a standard ordinary differential equation solver (e.g. $RK45$), a field line can be traced. In toroidal fusion devices when tracing such a field line associated with a given starting position for one toroidal turn, in general one does not reach the same point in space but rather a different location in the poloidal plane at the starting toroidal angle. The different rates of change in coordinates (ϑ, φ) per toroidal turn are hereby linked by the safety factor

$$q_s = \frac{B^\varphi}{B^\vartheta} \quad \text{to} \quad \mathrm{d}\varphi = q_s \mathrm{d}\vartheta. \tag{1.5}$$

For irrational values of q_s, this field line will completely fill a 2-dimensional surface which is then called flux surface, as both toroidal and poloidal magnetic flux remain constant within a given field line. By gradually changing the starting point of integration for field lines towards the center of a corresponding flux surface in the poloidal plane, one can asymptotically reach a degenerate flux surface which is represented by a single line, this field line is called the magnetic axis and will be used for the point of origin in s-direction for symmetry flux coordinates. For the use of SFC, one must assume that only one magnetic axis exists with all flux surfaces being nested flux surfaces, thus no magnetic islands are allowed. The outermost closed flux surface is called the separatrix, marking the transition between core plasma region and the scrape-off layer. In this thesis, only the core plasma region is considered, additions to the grid must therefore be programmed if one wants to include the scrape-off layer into calculations.
Figure 1.4 shows some schematic field lines in a tokamak for different values of q_s, each for one poloidal turn. As can be seen, only field lines with integer valued q_s are closed after one poloidal turn.

1.3.2 Field line integration and splining of axisymmetric fields

Next, one wants to construct a routine to map symmetry flux coordinate triplets to cylindric coordinates, in which all field quantities are subsequently read out. The approach presented here is only applicable for axisymmetric field configurations, as present in ideal *Tokamaks*. Symmetry flux coordinates topologically represent toroidal coordinates, therefore the first SFC coordinate is a minor radius-like quantity s, which in our case is chosen to be the normalized toroidal flux. However, in theory any flux label can be used, the second coordinate is ϑ which is related but not identical

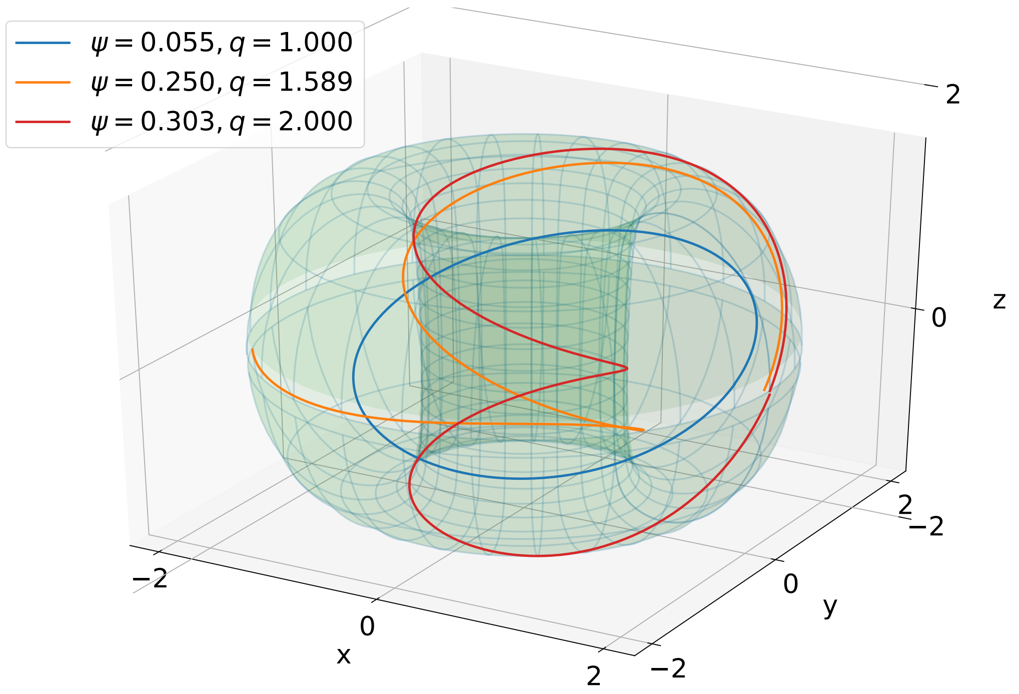

$\psi = 0.055, q = 1.000$
$\psi = 0.250, q = 1.589$
$\psi = 0.303, q = 2.000$
x
y
z
−2
0
2
−2
0
2
−2
0
2

Figure 1.4: Field lines with different safety factors in a tokamak

to the geometrical poloidal angle θ. The last coordinate is the geometric toroidal angle φ, which is the same as in standard cylindrical coordinates. Since field lines are chosen to be straight in these coordinates, the simple relation that the change in ϑ along the field line is proportional to the change in φ must hold, this proportionality factor represents the safety factor shown above. Hence, if one performs a step-wise integration of equation set 1.4 equidistantly along φ from 0 to $q_s 2\pi$ and calculates these points in (R, φ, Z) they must automatically correspond to point sets (s, ϑ, φ) with s being constant along a field line (0 at the magnetic axis and 1 at the separatrix), ϑ being equidistant from 0 to 2π and φ being equidistant from 0 to $q_s 2\pi$. For an axisymmetric configuration, one can now convert these coordinates to cylindrical coordinates via interpolation. A more detailed explantion of the procedure is given below.

Find O-point (magnetic axis)

The first step in implementing SFC is to find the magnetic axis which represents the $s = 0$ flux surface. This is done by starting to integrate a field line at position $(R, \varphi, Z) = (\frac{1}{2}(R_{min} + R_{max}), 0, \frac{1}{2}(Z_{min} + Z_{max}))$. The corresponding field values are obtained from calling the *field* routine that returns all necessary field information for the configuration. From this starting position the field line is intergrated for one toroidal turn using a standard ODE solver. Apart from the set of ODE for the magnetic field line, also the R and Z coordinates are summed up independently by the ODE solver. From this information, one can directly calculate the mean values of R and Z when following the field line by dividing the integrated values of R and Z by the integration angle of 2π. These values are then used for the next guess for the magnetic axis. From this new starting point another field line is followed and the mean values of R and Z are again computed. This iteration quickly converges to the position of the magnetic axis for a toroidally symmetric field. This routine is implemented in the file `field_line_integration_for_SYNCH.f90` where by default 20 iterations are performed.

Find X-point

Now that the magnetic axis has been found, the innermost starting point in the $\varphi = 0$ plane for the field line integrations has been determined. The outermost starting point will be given by a point on the separatrix, which is the boundary between closed and open flux surface domains (i.e. core region and scrape-off layer). To find a starting position on the separatrix, one takes the coordinates of the magnetic axis and parametrizes a line segment in cylindrical R-direction up to the largest R-value possible (saved in `rmx`) for the given configuration. The boundaries of R and Z for the current configuration are saved within the module `field_eq_mod` in file `field_divB0.f90` and can be read out via

```
rmn=rad(1)
rmx=rad(nrad)
zmn=zet(1)
zmx=zet(nzet).
```

This line segment is then split up equidistantly, by default 10000 points are chosen. Points p_i are placed equidistantly with

$$p_i = \vec{\mathrm{O}} + \frac{i}{N} \begin{pmatrix} rmx - O_R \\ 0 \end{pmatrix} \tag{1.6}$$

in cylindric coordinates for $i = 0, 1, 2, .., N$. Now, starting from the magnetic axis, for each of these points a magnetic field line is integrated for two poloidal half turns in successive steps of $\Delta\varphi = 2\pi/10$, resulting in one full turn. After each integration step the current position is compared to the (R, Z)-constraints of the domain, in case a maximum/minimum value is exceeded the current field line is no longer closed. Thus, the previous starting point for the integration can be assumed to represent the starting point for the last flux surface. However, this last closed field line is already suboptimal in quality so the preceding starting point is taken as the last closed field line for this configuration, hence representing the separatrix. As mentioned above, the X-point lies on the separatrix, in addition to this condition, the X-point has the property that the poloidal magnetic field vanishes at its position. Consequently, when following the last closed field line, upon reaching the X-point no poloidal movement occurs, thus (R, Z) remain constant. The algorithm for finding the X-point uses this property by integrating the last closed field line in steps of $\Delta\varphi = 2\pi/10$ and comparing the last (R, Z) position with the position of the previous integration step. The distance between the two positions is evaluated and compared with the distance between the positions of the previous steps, if the new distance is so far the lowest, the new position and the distance are saved in variables `min_d` and `x_point`. These steps are performed until one full poloidal turn is completed, the position of the X-point can then be taken from the variable. To make sure that one only integrates over one poloidal turn, one can use the property of the cross product that $|\vec{a} \times \vec{b}| = |a||b|\sin(\alpha)$ with $|\vec{a}|$ being the poloidal starting position of the integration and $|\vec{b}|$ the current poloidal position. Upon completing one full poloidal turn, the sign of the sine will flip from -1 to 1, by detecting this flip one can stop the integration accordingly.

Scanning flux surfaces

After finding the O-point and the X-point, the next step is to connect them by a straight line in cylindrical coordinates in the $\varphi = 0$-plane, this line segment is subsequently chosen to respresent the $\vartheta = 0$ contour in SFC. On this line segment 500 points are placed equidistantly, then for each of these points, an independent field line integration over one poloidal turn with a step size of $\Delta\varphi = 2\pi/10$ is started. The goal of these particular integrations is to determine the safety factors of the individual field lines, which can be easily calculated from the toroidal integration angles corresponding to exactly one poloidal turn. Again, the approach using the flip of the cross product sign will be used to determine whether the last integration step finished the turn. However, due to the finite size of the integration steps the necessary toroidal integration angle is not precisely determined. Thus, iterations of Newton's method are applied in order to obtain the precise integration angle to complete the turn. To implement such a routine, one needs to take a look at some geometric considerations. Figure 1.5 depicts a sketch of the components relevant for Newton's method:
In order to implement a Newton's scheme, one first takes a look at the normal distance from the position after the last integration step $ymet_{\text{axis}}$ to the $\theta = 0$ axis, for this it is convenient to rotate the system such that the θ axis points in the x-direction. The

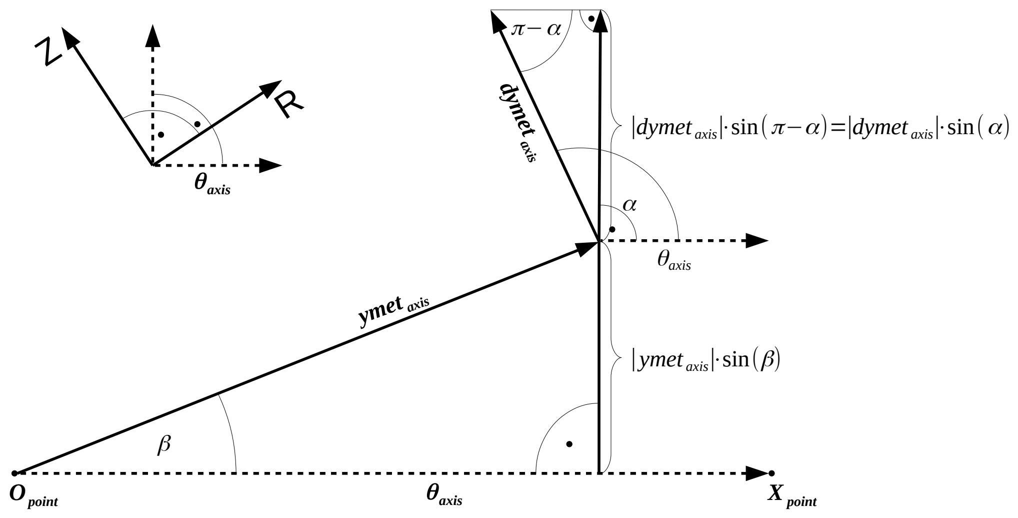


Figure 1.5: Visualization of quantities for Newton's method

normal distance from the end position $\boldsymbol{ymet}_{\text{axis}}$ to the axis $\boldsymbol{\theta}_{\text{axis}}$ is then given by

$$d_\perp = |\boldsymbol{ymet}_{\text{axis}}|\sin(\beta).$$

Next, one is interested in the normal derivative of the field line to the axis with respect to φ. In the routine, the derivatives *dr_dphi* and *dz_dphi* are provided by the numerical integration scheme, here they are combined in vector $\boldsymbol{dymet}_{\text{axis}}$. Using the identity $sin(\pi - x) = sin(x)$ one can evaluate the normal derivative as

$$\frac{\partial d_\perp}{\partial\varphi} = |\boldsymbol{dymet}_{\text{axis}}|\sin(\alpha),$$

with Newton's guess for the correction of the integration angle

$$\Delta\varphi = \sigma\frac{d_\perp}{\left(\frac{\partial d_\perp}{\partial\varphi}\right)} = \sigma\frac{|\boldsymbol{ymet}_{\text{axis}}|\sin(\beta)}{|\boldsymbol{dymet}_{\text{axis}}|\sin(\alpha)}$$

and σ being the sign of $(\boldsymbol{ymet}_{\text{axis}} \times \boldsymbol{\theta}_{\text{axis}})$ to ensure the correct direction of integration. This correction scheme is applied iteratively for each field line, by default 50 iterations are performed to obtain accurate values for φ_{total}.

Upon determining the correct toroidal integration limits to complete one poloidal turn for each point, the safety factors for the field lines are then directly given by

$$q_{\text{saf}} = \frac{\varphi_{\text{total}}}{2\pi}.$$

Apart from the safety factor, also the approximate average minor radius of the field line `rsmall`, the poloidal flux ψ_{surf} (variable `psisurf`) and the toroidal flux Φ_{tor} (variable `phitor`) are computed for each field line. For the application at hand `rsmall` is irrelevant so it will not be discussed, `rbeg` is also an output of the routine which has no physical meaning as it is never evaluated (it is not removed to ensure that the function call stays the same for compatibility with other codes), the poloidal flux ψ_{surf} is obtained from the module `field_eq_mod` by calling the `field_eq` subroutine at the start position of the integration, from this value the poloidal flux at the magnetic axis then needs to be subtracted (its value is computed also via the `field_eq` subroutine). The toroidal flux `phitor` is calculated by numerically integrating equation

$$\frac{\partial \Phi_{\text{tor}}}{\partial \varphi} = R \cdot Z \cdot B_r$$

when performing the field line integration, here B_r denotes the physical component of the magnetic field in the R-direction. For normalization, the obtained result still needs to be divided by 2π.
So far, the flux functions `rsmall`, `qsaf`, `psisurf` and `phitor` were calculated for the array of 500 field lines, with flux functions being defined as functions that remain uniform along a field line / on a flux surface, thus only depending on the flux surface label. Due to this property any flux function can be used to label a given flux surface, in this application the normalized toroidal magnetic flux, here denoted s, will be used as the flux label. Now, one is interested in the positions R, Z, the modulus of the magnetic field `bmod` $= (|\mathbf{B}|)$ and the metric determinant `sqgnorm` $= (\sqrt{|g|})$ in equidistant steps along the field lines. These values are needed for the interpolation routine to convert components from symmetry flux coordinates to cylindric coordinates. Due to axisymmetry in the configuration the field lines only need to be integrated for exactly one poloidal turn, so all field lines need to be integrated for the previously found φ_{total} values in φ direction over 500 equidistant steps. After each step, R and Z are obtained directly as output argument from the standard ODE integrator, physical components of the magnetic field are obtained from calling the `field_eq` routine, thus `bmod` $= \sqrt{B_r^2 + B_p^2 + B_z^2}$, finally `sqgnorm` for symmetry flux coordinates is calculated via `sqgnorm` $= R/|B_p|$.
On a sidenote, due to the straightness of field lines in symmetry flux coordinates the

points along the field lines are equidistant in both φ and ϑ but generally not in s direction, however s remains constant along a given field line.

Interpolation of data with respect to ϑ and normalization

The algorithms concerning field line integration which are explained in the previous subchapters are all part of the same subroutine `field_line_integration_for_SYNCH`. This subroutine is called from a second subroutine `preload_for_SYNCH` where the calculated quantities are saved into separate files. These files are subsequently read out from a third subroutine `load_magdata_in_symfluxcoord`. In this subroutine, the data for each field line is then interpolated with periodic third order splines in theta with the subroutine `spl_per`, the toroidal angle is no longer of interest due to axisymmetry as it is equivalent to the toroidal angle in cylindric coordinates. Moreover, since for each field line the flux label s remains constant, the change of each quantity along a given field line must be purely a function of the symmetry flux coordinate ϑ. The spline coefficients from the `spl_per` subroutine are then saved into variables `R_st`, `Z_st`, `bmod_st`, `sqgnorm_st` (suffix `_st` stands for splined in theta). The calculated flux functions are then also read out from the file, however, these quantities remain constant along the field line so there is no need for splines in ϑ direction. Since only the normalized fluxes are of interest for this application, both ψ_{surf} and Φ_{tor} are divided by their maximum values. The normalized toroidal flux s will label the individual flux surfaces. The precomputed data from `load_magdata_in_symfluxcoord` are directly saved into the module `magdata_in_symfluxcoor_mod` since they only need to be precomputed once. The subroutine `magdata_in_symfluxcoord_ext` then accesses these data and performs the necessary s interpolation for arbitrary positions, which is very efficient compared to the precomputation.

s interpolation of data

In **??**, the method of Lagrange polynomial interpolation is introduced. Now to interpolate the data for a given point (s, ϑ) one searches the field line array via bisection to find the indices and s values of the closest four field lines to the position s. As mentioned, this is done in subroutine `magdata_in_symfluxcoord_ext(inp_label,s,psi,theta, q,dq_ds,sqrtg,bmod,dbmod_dtheta,R,dR_ds,dR_dtheta,Z,dZ_ds,dZ_dtheta)`, whereby depending on the value of the input label `inp_label`, either the variable `s` or `psi` define the input for the minor radial position while the variable `theta` defines the symmetry flux poloidal angle, the toroidal angle `phi` remains invariant, thus, it is not included in the subroutine call, the remaining arguments are outputs of the subroutine. An overview of the parameters is given in table 1.1.

The splines for these four field lines are then evaluated for the given ϑ position, this yields an array of four s values, which are belonging to the field lines, and the corresponding interpolation quantities. With the four s values of the closest field lines, the Lagrange coefficients $\mathscr{L}_k(s)$ are now fully determined, acting as weights for the quantities on the fields lines that are to be interpolated. The output is then given by

$$P(s) = \sum_{k=0}^{n} \mathscr{L}_k(s) f_k$$

for any interpolated quantity f. $P(s)$ hereby interpolates $f(s)$ and f_k represents $f(s_k)$ with discrete values s_k at flux surface k.

Table 1.1: Parameters for `magdata_in_symfluxcoord_ext`

Name; Data type	Description
`inp_label`; `integer`	input switch, where 1 sets `s` and 2 sets `psi` as input
`s`; `double precision`	normalized toroidal flux, also the value of the first component in SFC (symmetry flux coordinates)
`psi`; `double precision`	poloidal flux at (s, ϑ, φ)
`theta`; `double precision`	value of the second component in SFC
`q`; `double precision`	safety factor at (s, ϑ, φ)
`dq_ds`; `double precision`	partial derivative of the safety factor with respect to s at (s, ϑ, φ)
`sqrtg`; `double precision`	square-root of the metric determinant at (s, ϑ, φ)
`bmod`; `double precision`	modulus of the magnetic field at (s, ϑ, φ)
`dbmod_dtheta`; `double precision`	partial derivative of the modulus of the magnetic field with respect to ϑ
`R`, `dR_ds`, `dR_dtheta`; `double precision`	first component of position in cylindric coordinates (R, φ, Z) and its derivatives with respect to s and ϑ
`Z`, `dZ_ds`, `dZ_dtheta` `double precision`	third component of position in cylindric coordinates (R, φ, Z) and its derivatives with respect to s and ϑ

1.3.3 Field aligned grid generation

So far, the subroutine `magdata_in_symfluxcoord_ext(inp_label,s,psi,theta,q,dq_ds, sqrtg,bmod,dbmod_dtheta,R,dR_ds,dR_dtheta,Z,dZ_ds,dZ_dtheta)` has been constructed to convert arbitrary SFC positions (s, ϑ, φ) back to cylindrical coordinates positions (R, φ, Z). Now, the logical scheme that is used in *Gorilla* for generating the field aligned grid is explained. Here, the subroutines that are called in order to generate the grid are structured according to figure 1.6.

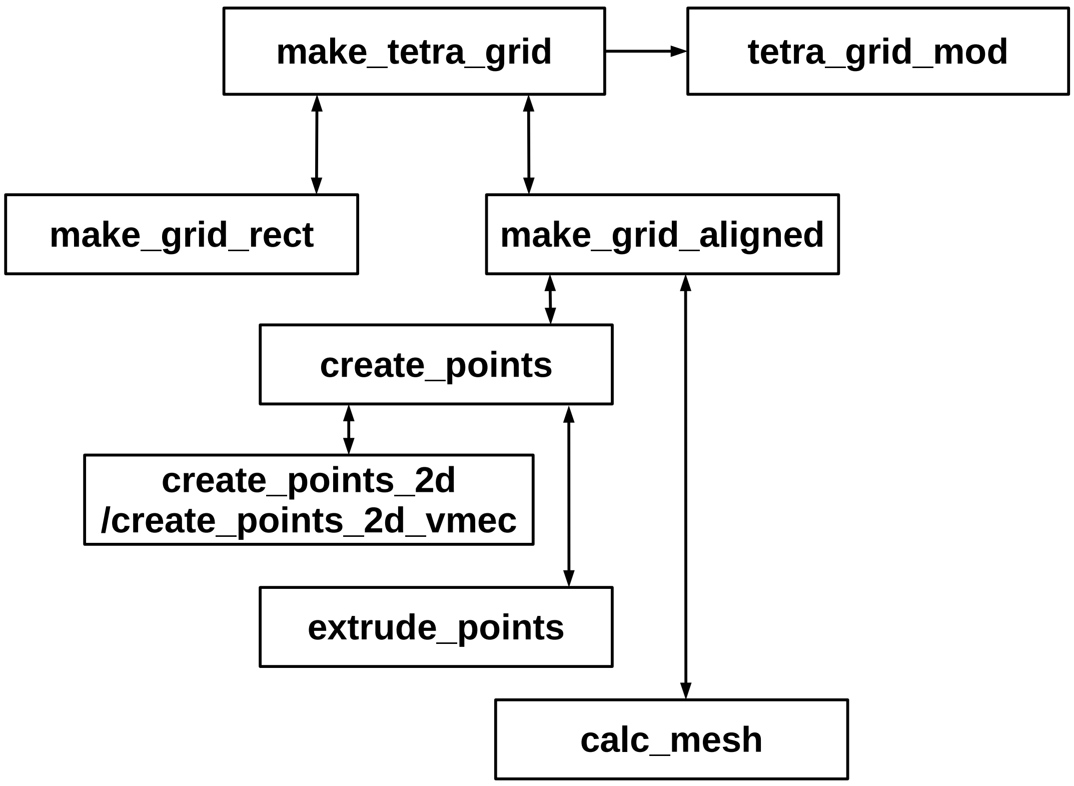


Figure 1.6: Subroutine `make_tetra_grid` code structure

In this hierarchy, the first subroutine `make_tetra_grid` is called with arguments `(grid_kind_in, grid_size_in)`, the first input being an integer label with value 1 for the cylindrical contour grid and 2 for the field aligned grid, the second input is an integer array with 3 elements defining the number of grid elements along each coordinate. From a programmatic aspect, this routine can be thought of as a constructor from object oriented programming as it constructs an instance of a suitable grid according to the Fortran type `tetrahedron_grid` (which can essentially be seen as a class) with additional variables saved together in the module `tetra_grid_mod`. In this module, all geometry related properties of the grid are saved, moreover, they are saved with attributes `public, protected`, thus, while being publically avaiable for read-out they can only be altered from subroutines and functions belonging to the module itself. For this reason the subroutine `make_tetra_grid` must also be defined within the module. By having the entire grid generation being covered with a single subroutine call and not being able to change it otherwise, this adds an additional layer of security regarding unexpected changes of grid quantities due to unintended coding mistakes. Depending on which grid one wants to generate (i.e. either the cylindrical contour or the field aligned grid), subsequently either the subroutine `make_grid_rect(tetra_grid,verts_rphiz,grid_size,Rmin,Rmax,Zmin,Zmax)` or the subroutine `make_grid_aligned(grid_size,efit_vmec)` is called from `make_tetra_grid`. For the `make_grid_rect` subroutine, the first two arguments denote the output, the latter arguments are inputs for the grid generation. The `make_grid_aligned` subroutine only takes input arguments, as it writes the generated grid data directly into the module, thus, the subroutine must be also defined within the module `tetra_grid_mod`. For clarity, the elements of the Fortran module `tetra_grid_mod` and type `tetrahedron_grid` are displayed in tables 1.2 and 1.3. All call parameters of the grid constructing subroutines are hereby briefly explained.

Table 1.2: Variables of Fortran module `tetra_grid_mod`

Name; Data type	Description
`tetra_grid` `type(tetrahedron_grid),` `dimension(ntetr)`	array of instances of type `tetrahedron_grid` with `ntetr` elements
`verts_rphiz` `double precision,` `dimension(nvert,nvert)`	coordinate triples (R, φ, Z) of all `nvert` grid vertices
`verts_xyz` `double precision,` `dimension(nvert,nvert)`	coordinate triples (x, y, z) of all `nvert` grid vertices
`verts_sthetaphi` `double precision,` `dimension(nvert,nvert)`	coordinate triples (s, ϑ, φ) of all `nvert` grid vertices
`ntetr` `integer`	total number of tetrahedra in the grid
`nvert` `integer`	total number of vertices in the grid
`grid_kind` `integer`	switch for which grid version is generated
`grid_size` `integer, dimension(3)`	dimensions of the grid in (R, φ, Z) or (s, φ, ϑ)
`Rmin,Rmax,Zmin,Zmax` `double precision`	dimensions of the fusion device in cylindrical coordinates
`efit_vmec` `integer`	switch that specifies for the field aligned grid whether the tokamak equilibrium (efit) or the stellarator equilibrium (vmec) is taken

Table 1.3: Fortran type `tetrahedron_grid`

name; data type	description
`ind_knot` `integer, dimension(4)`	pointer from tetrahedron vertex index (1 to 4) to total vertex index (1 to `nvert`)
`neighbour_tetr` `integer, dimension(4)`	pointer from the face index to the index of the next tetrahedron
`neighbour_face` `integer, dimension(4)`	index of the neighboring tetrahedron's entry face from the exit face
`neighbour_perbou_phi` `integer, dimension(4)`	1 if the face is on periodic boundary $\varphi = 2\pi$, -1 if on $\varphi = 0$ and 0 otherwise
`neighbour_perbou_theta` `integer, dimension(4)`	1 if the face is on periodic boundary $\vartheta = 2\pi$, -1 if on $\vartheta = 0$ and 0 otherwise

make_grid_aligned

As discussed, the subroutine `make_grid_aligned` generates the data for the field aligned grid and is called from subroutine `make_tetra_grid`. In this section, the substructure and working principle of this subroutine, which again consists of several subroutines, will be discussed. The code structure was previously introduced in figure 1.6 and shows, that elements of `make_grid_aligned` can be further organized into `create_points` and `calc_mesh`, whereas `create_points` can be subdivided into `create_points_2d` and `extrude_points`. In addition to `create_points_2d`, there exists an analogous subroutine named `create_points_2d_vmec` which essentially works the same way but is used for the generation of the field aligned grid in stellarators, therefore, it does not use the conversion routine `magdata_in_symfluxcoord_ext` but an alternative subroutine called `splint_vmec_data` that converts *VMEC*-coordinates back to cylindrical coordinates, this was provided by S. Kasilov. Since there are merely very minor differences in the approach of the grid generation, compared to the axisymmetric field aligned grid, the treatment of subroutine `create_points_2d_vmec` is not within the scope of this thesis.

create_points

Upon calling the subroutine `make_grid_aligned`, first `create_points` is executed with arguments (`verts_per_ring, nphi, verts_rphiz, verts_sthetaphi, efit_vmec, field_periodicity, nvert, r_scaling_func,theta_scaling_func, repeat_center_point`), where `verts_per_ring` denotes an array where each element represents the number of poloidal discretizations for the corresponding ϑ-ring (this is named ring due to the periodicity in ϑ-direction), `nphi` the number of discretizations in toroidal direction (=`grid_size(2)`), `repeat_center_point` the boolean that sets that the central point on the magnetic axis is in fact again a ring in ϑ-direction (instead of a single point at the center which connected to all vertices of the first ring, however, this only makes sense in cylindrical coordinates), finally, the scaling functions `r_scaling_func` and `theta_scaling_func` define the distribution of grid points in s and ϑ-direction, the remaining arguments were already discussed previously.

create_points_2d

The first step in generating the vertices for the grid within `create_points` is to generate the vertices lying on the $\varphi = 0$ plane. This is done with subroutine `create_points_2d`. Here, first the subroutines `preload_for_SYNCH()` and `load_magdata_in_symfluxcoord()` are called in order to obtain access to the conversion routine `magdata_in_symfluxcoord_ext`, which allows to convert components given in axisymmetric symmetry flux coordinates (s, ϑ, φ) back to cylindrical coordinates (R, φ, Z). This conversion is necessary, as magnetic field data will later be accessed by the `field` subroutine which only takes arguments in cylindrical coordinates.
The next step in the subroutine is to define an equidistant vector `r_frac` that stores the unscaled s-values for the individual ϑ-rings. Here, it is important to note that when using symmetry flux coordinates for orbit computation, the first ring cannot lie precisely on the magnetic axis, but rather on an arbitrarily small distance to the magnetic axis. This is an inherent property of the grid, since when putting the innermost ring on the magnetic axis, every second tetrahedron of the central tetrahedra would in fact have zero volume together with a numerically undefined normal vector in the s-direction, furthermore, particle trajectories are computed from one tetrahedral face to the next with a defined convergence normal-distance to the exit face, this approach will no longer work if the tetrahedral volume approaches zero. In reality, however, the biggest problem here is in fact that the conversion routine `magdata_in_symfluxcoord_ext` is no longer sufficiently accurate for diminishingly small values of s, so the vertices are no longer assured to be well-aligned (on infinitesimal scales around the magnetic axis), but rather seemingly chaotic. One might suggest to completely omit the generation of these *pathological* tetrahedra, however, this would lead to holes in symmetry flux coordinate space, as coordinate components would be discontinuous in ϑ-direction when moving poloidally from one tetrahedron to the next. Instead, one defines a minimum s-value `s_min` for the innermost flux surface, leading

to a continuous coordinate space with a lower boundary. This, however, has the implication, that now a small annulus exists in the grid in real space, therefore one has to implement a special way to handle particle orbits, that intersect with this boundary. So far, such a treatment has not been introduced, instead particles that intersected with this boundary were assumed to leave the torus. This has been a valid approach, as this rarely occurs, thus, it does not significantly influence the results obtained by statistics using high particle counts (e.g. 30000 particles). If needed, a possible solution would be to logically connect the tetrahedral faces which lie on opposite sites of the annulus and subsequently add π to the ϑ-component, when a particle intersects with such a plane. To better understand the problem that occurs, a schematic picture of poloidal projection of the field-aligned grid with different minimum values for s is presented in figure 1.7.

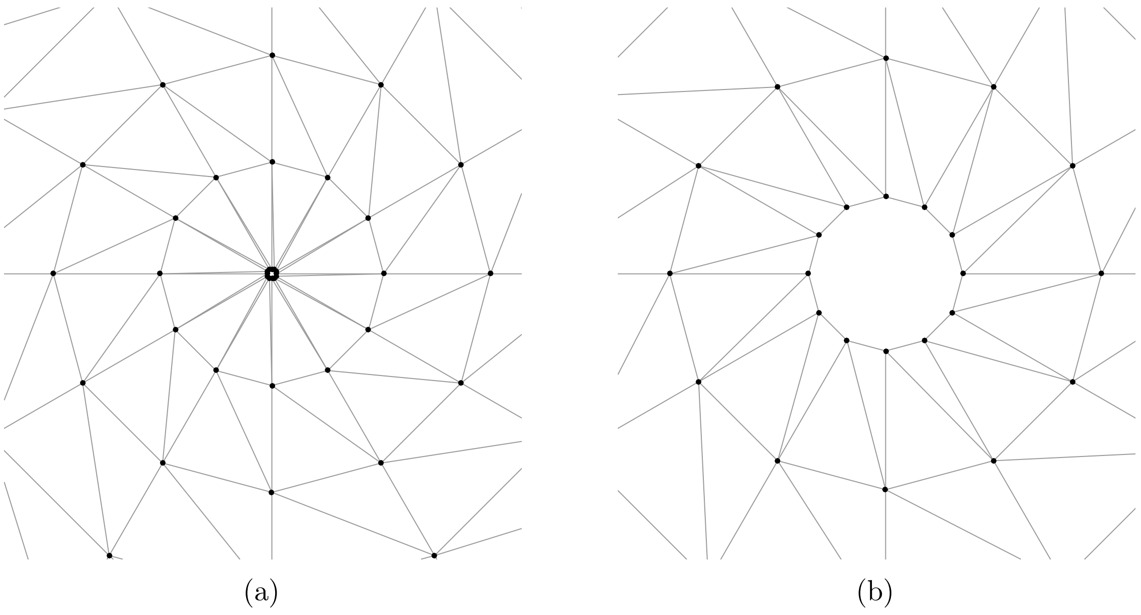


Figure 1.7: This plot presents a magnified schematic picture of the poloidal projection of the field-aligned grid center in cylindrical coordinates for two different values of `s_min`. In plot 1.7a a very small value for `s_min` has been chosen, this is compared to a larger value for plot 1.7b. It is clearly visible that the tetrahedral faces which have two corner points on the inner-most ϑ-ring are only visible in 1.7b, while they appear to be merely lines in 1.7a.

Back to the grid generating procedure, the vector `r_frac` is defined by

```
r_frac  =  s_min + [(dble(i)*(1.d0-s_min), i=1, size(r_frac), 1)]&
           &/(dble(size(verts_per_ring)) ,
```

where "`&`" denotes the line concatenation operator in Fortran. Next in `create_points_2d`, two nested loops are implemented, the outer one iterates over the individual ϑ-rings with loop index `isurf` ranging from `isurf=0` (for the center point/ring) up to `isurf=size(verts_per_ring)`, this loop represents the iteration over the individual ϑ-rings, on which the points will lie. The inner loop iterates with index `j` ranging from `j=1` up to `verts_per_ring(isurf)`, this second loop represents the iteration over the individual vertices of the current ϑ-ring. For each iteration of the outer loop, an additional vector `theta_frac` is generated according to

```
theta_frac = [(i, i=0, n_theta_current-1,1)] / dfloat(n_theta_current) .
```

Additionally, a counter `point_idx` is introduced, that is incremented by 1 each time a vertex is generated, this counter determines the indices of the vertices, by which their coordinates will later be accessed for the read-out of field data. The actual grid points are then generated within the loop by the code fragment

```
s = r_scaling_func(r_frac(isurf))
theta = 2.d0*pi*theta_scaling_func(theta_frac(j))
verts_sthetaphi(:,point_idx) = [s,theta,0.d0]
call magdata_in_symfluxcoord_ext(1,s,psi,theta,q,dq_ds,sqrtg,&
     &bmod,dbmod_dtheta,R,dR_ds,dR_dtheta,Z,dZ_ds,dZ_dtheta)
verts_rphiz(:,point_idx) = [R,0.d0,Z]
point_idx = point_idx +1 .
```

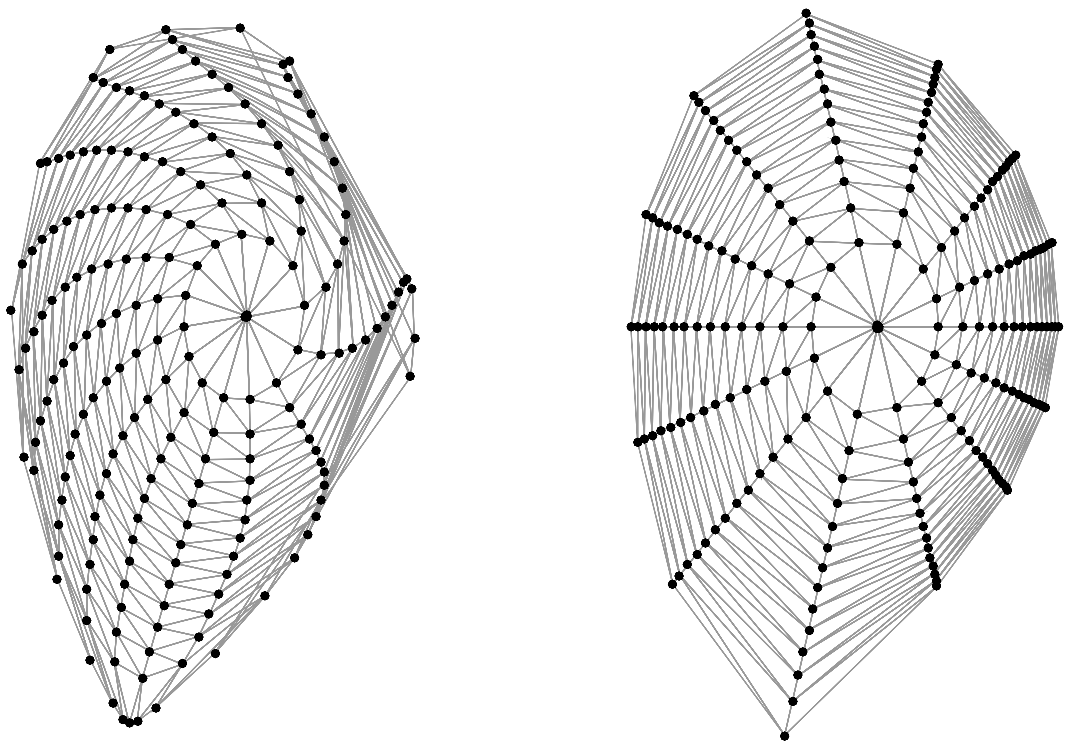
For scaling functions `r_scaling_func(x) = theta_scaling_func(x) = x`, equidistant grid vertices lying on the $\varphi = 0$ plane are hereby generated in symmetry flux coordinates. There is one optional but relevant modification that should further be discussed. Instead of generating vertices according to a distribution defined for the poloidal symmetry flux angle ϑ, one can also distribute vertices according to the geometric poloidal angle θ in cylindrical coordinates. This is particularly interesting if the field aligned grid is intended to be used with the cylindrical coordinate system, as vertices distributed in ϑ experience a strong poloidal shift towards the X-point the closer they lie to the separatrix, in fact if the outermost points were actually to lie precisely on the separatrix, they would all lie exactly at the X-point in real space. Consequently for the grid, if the geometric θ-distribution of vertices changes too drastically from one flux surface to the next, it is possible that scalar products for at least one pair of normal vectors of the individual tetrahedral planes for a given tetrahedron no longer yields a negative value (if one wants to imagine what happens here, assume a tetrahedron and drag one of the vertices through the opposing plane spanned up by the remaining three vertices, the resulting tetrahedron is literally turned inside out, this occurs as the outermost point is increasingly shifted poloidally towards the X-point). A negative scalar product of such planes would imply that in fact the implemented logical association of vertices is invalid and the generated tetrahedron is therefore incompatible with the *Gorilla* integration scheme. Thus, an interpolation routine was implemented, which transforms defined geometric angles θ back to the corresponding symmetry flux angles ϑ for a given value of s. The conversion subroutine is hereby called by

```
call theta_geom2theta_flux(s,2.d0*pi*theta_scaling_func(theta_frac(j))&
     &,theta) !optional ,
```

instead of directly defining `theta` in the second line of the code where the vertices are calculated. The variable that acts as a switch is denoted `geom_flux`, where an integer value of 1 distributes the points directly in symmetry flux angle ϑ and an integer value of 2 distributes the points in geometric angle θ using the subroutine `theta_geom2theta_flux`. The vertices generated by the subroutine `create_points_2d` are presented in figure 1.8, once with vertices poloidally equidistant in symmetry flux coordinate angle ϑ (1.8a) and once with vertices poloidally equidistant in geometric angle θ (1.8b). The grey lines indicate how these points will be connected in cylindrical coordinates to form the tetrahedral mesh, mind that tetrahedra are here represented by triangles in the poloidal projection. On the left side, it can be seen that in figure 1.8a some lines close to the separatrix do indeed intersect, this leads to a corrupted mesh logics in cylindrical coordinates that causes errors with the *Gorilla* integrator, on the right side, figure 1.8b does not show intersections, thus, this grid is compatible with *Gorilla* for computations done in cylindrical coordinates. It should, however, be emphasized, that this problem only occurs for the field-aligned grid when changing to cylindrical coordinates, as long as calculations are performed in symmetry flux coordinates, geometrically aligned grids become in that case obsolete.

extrude_points

The next step in constructing the vertices for the 3D-grid in subroutine `create_points` is now to extrude these points symmetrically into the toroidal (φ) direction. This extrusion is realized in subroutine `extrude_points(verts_per_slice,nphi,phi_position,`



(a) vertices are here poloidally equidistant in symmetry flux angle ϑ

(b) vertices are here poloidally equidistant in geometrical angle θ

Figure 1.8: Poloidal projection of field aligned grids in cylindrical coordinates of grid size $(N_s, N_\vartheta) = (14, 14)$, vertices are indicated by black dots, the grey lines in the back indicate how these vertices will later be connected to form tetrahedra, mind that for 1.8a intersections occur close to the separatrix, while for 1.8b this problem was circumvented by aligning the vertices equidistantly in geometrical angle θ

`verts_sthetaphi/verts_rphiz`), with

```
verts_per_slice = sum(verts_per_ring) + verts_per_ring(1)
```

if the boolean `repeat_center_point` is `true` and

```
verts_per_slice = sum(verts_per_ring) + 1
```

otherwise. The remaining parameters `nphi`, `phi_position` and `verts_sthetaphi/verts_rphiz` represent the number of grid points in toroidal direction, the index of the toroidal component φ in the current coordinate system (the value is hereby 2 for cylindrical coordinates (R, φ, Z) and 3 for symmetry flux coordinates (s, ϑ, φ)) and the coordinates of the vertices that are to be extruded, respectively. The last parameter acts here as an `inout`-variable where the first `verts_per_slice` number of points are the input and the remaining components are returned by the subroutine. It should furthermore be mentioned, that some liberty has been assumed in citing the fortran syntax as `verts_sthetaphi/verts_rphiz` means here that either one can be the input, but directly evaluating this expression would return a syntax error. Using the defined parameters, the point extrusion is performed by copying the previously computed two-dimensional vertex coordinates and saving them together with the appropriate φ-positions. This is a purely index-based operation, given by

```
do i = 2, nphi
     vert_idx = (i-1)*verts_per_slice+1
     phi = (2.d0*pi/field_periodicity*(i-1))/nphi
     points(:,vert_idx:vert_idx+verts_per_slice-1) = &
          & points(:, 1:verts_per_slice)
     points(phi_position,vert_idx:vert_idx+verts_per_slice-1)=phi
end do
```

where `points` is a placeholder for vertex coordinates `verts_sthetaphi/verts_rphiz` and `field_periodicity` denotes the previously introduced periodicity factor of the magnetic field, this is 1 for the axisymmetric field and can have a different integer value for a stellarator configuration (e.g. field_periodicity = 5 for a stellarator field that is invariant under a coordinate shift of magnitude $\Delta\varphi = 2\pi/5$ in toroidal direction). The coordinates for all grid vertices have now been computed, the subroutine `extrude_points` ends here.

calc_mesh

The final step in constructing the field aligned grid is to take the generated vertices, and logically connect them to form tetrahedra in a way, that no unassigned spaces exist within the given coordinate space (the only exception here is the annulus in real space due to the degeneracy of the poloidal symmetry flux component at the magnetic axis, as discussed previously). This contruct will be referred to as mesh, which is calculated by the subroutine `calc_mesh(verts_per_ring, nphi, verts_rphiz(:, :nvert / nphi), ntetr, verts, neighbours, neighbour_faces, perbou_phi, perbou_theta, repeat_center_point = .true.)`. This is called from the subroutine `make_grid_aligned`, as shown in figure 1.6. Subroutine parameters, that have not yet been introduced, are `verts`, `neighbours`, `neighbour_faces`, `perbou_phi` and `perbou_theta`. These variables denote placeholders for tetrahedron related data, where `verts(1:4,nte`

`tr)` stores the indices of the four corner vertices for each tetrahedron, `neighbours(1:4, ntetr)` the indices of the neighboring tetrahedra (1 : `ntetr`) which are lying adjacent to the four faces of the given tetrahedron, `neighbour_faces(1:4,ntetr)` the face indices (1..4) that denote which face of the neighbor is lying adjacent to the current tetrahedron, `perbou_phi(1:4,ntetr)` which tetrahedral faces lie on the periodic boundary in φ direction where 0 is the default value for all tetrahedra, 1 is the value at the $\varphi = 2\pi$ boundary and -1 at the $\varphi = 0$ boundary, this is analogously implemented for `perbou_theta` which treats the periodic boundary conditions in ϑ direction. Upon generation of the tetrahedra, these data are computed and subsequently saved in the previously introduced fortran type `tetrahedron_grid` given in table 1.3, there the tetrahedra are made available for further computations.
Next, the meshing algorithm will be explained in more detail. In order to be able to understand the ideas behind this algorithm, one needs to take a closer look at how the vertices are indexed. For this, a set of poloidal vertices (i.e. $\varphi = 0$) is given in figure 1.9. Here, one can see that vertices are first indexed in ϑ direction (along the ϑ rings which are horizontal sets of vertices in this represenation) where due to periodic boundary conditions in ϑ the first vertex appears twice, once for $\vartheta = 0$ and once for $\vartheta = 2\pi$. Subsequently, the vertices of the adjacing ϑ ring are indexed in the same fashion up to index `verts_per_slice`, which is the number of vertices on the $\varphi = 0$ plane. Due to axisymmetry of the grid, the indices of the vertices that lie on the next slice (in positive φ direction) have the exact same coordinates in the poloidal projection as the vertices lying on the first slice ($\varphi = 0$), only the toroidal component differs by $\Delta\varphi = 2\pi/\texttt{nphi}$ from one slice to the next. The word *slice* was chosen in this context, as it figuratively refers to a cake (the grid) being sliced into `nphi` pieces of equal size, here the vertices lie on the slice faces. In this analogy, two adjacent slice faces are created by the same cut, in the grid the cuts themselves represent the slices on which the vertices lie, whereas the cake pieces can be thought of as the space inbetween these slices which will be covered by the tetrahedra. This analogy was given to justify the selected terminology and will be no longer dwelled upon. The important information, that one can take from this picture is, however, that in the poloidal projection the vertices of different slices are equivalent and thus, the index for each vertex is incremented by `verts_per_slice` to obtain the index of the vertex in the next slice. To account for periodicity in φ, the resulting index is taken `modulo(index,nvert)`. The indices for the second slice are given in parentheses next to the vertex indices of the first slice in figure 1.9.
For the cylindrical contour grid, the next step is to connect the vertices to form hexahedra, which are each comprised of six tetrahedra. However, this approach is not

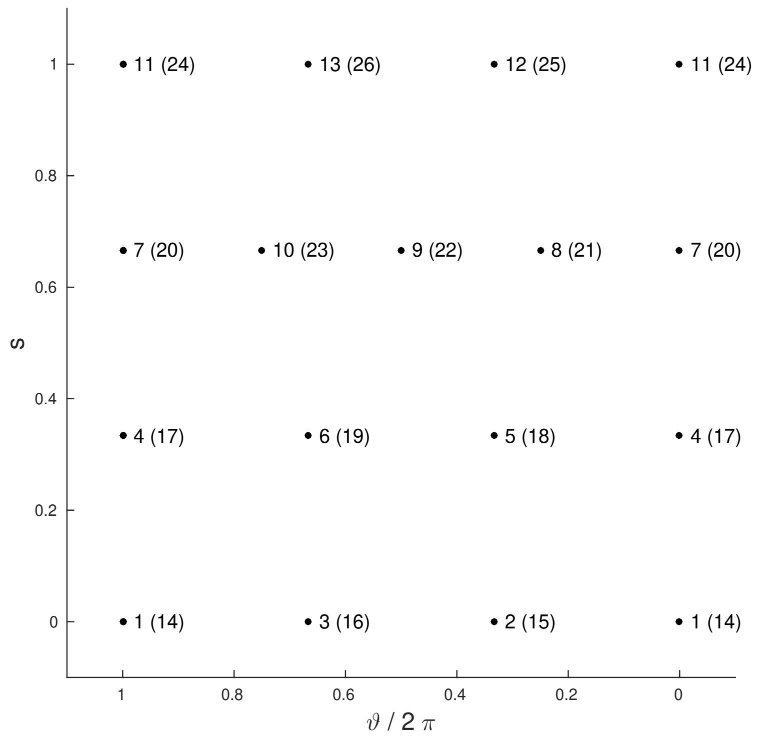


Figure 1.9: Grid vertex indices for the first φ-slice and in parentheses the indices of the next slice, the indices between slices differ by a constant value `verts_per_slice` which is equal to 13 for this configuration

possible as it would require that each ϑ ring had the exact same number of vertices, which is generally no longer the case. Instead, prisms are used to connect the vertices, these prisms have the property that their vertices can directly be indexed to form three tetrahedra for each prism. The two types of prisms that are used to construct the grid are shown in figure 1.10. These prisms can be associated with certain orientations which are here denoted either *top facing* or *bottom facing*. This can be understood by taking a look at figure 1.9. Here, in the lower right corner of the graph, the vertices with indices 1, 2, 4 and 5 can be connected in the poloidal plane by two triangles with the corner points for the first triangle {1, 2, 5} corresponding to prism vertices {0, 2, 3} for the top facing prism and the corner points for the second triangle {1, 4, 5} to prism vertices {0, 1, 2} for the bottom facing prism, respectively. The first triangle has one edge on the upper ϑ-ring, while the second triangle has one edge on the lower ϑ ring, hence the naming top facing and bottom facing. These triangles are in fact the cross sectional representation of the prisms, which are essentially axisymmetrically extruded triangles. The use of prisms is hereby only possible due to the axisymmetric arrangement of vertices on different slices.

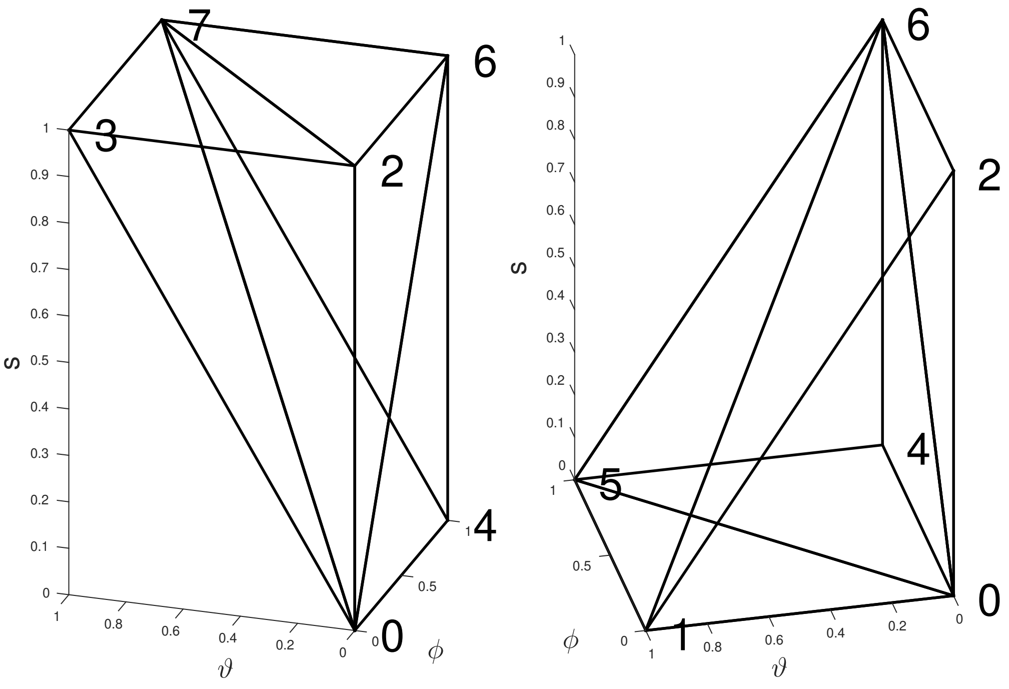


Figure 1.10: Prisms used to index tetrahedra, top facing prism on the left and bottom facing prism on the right

The approach is now to use these two types of prisms to link all vertices of the individual ϑ-rings to the face vertices of the two presented prism types. By doing so, a full slice of the grid will be constructed, where each prism is subsequently split up into three tetrahedra by indexing. Additionally, the tetrahedron properties `verts`, `neighbours`, `neighbour_faces`, `perbou_phi` and `perbou_theta` are computed upon generation. It was already discussed, that the vertices in the poloidal plane can be connected by triangles representing the prisms. The question is merely, how the triangles are to be arranged, such that the poloidal projection of the grid is fully and unambiguously covered by triangles. For this, the Delaunay condition is used to determine if a proposed triangle has desirable properties, i.e. a maximal smallest interior angle. In the following, the triangles and their associated prisms will be collectively referred to as segments. A concrete example shall allow the reader to get a clearer image of how this process is implemented. One is first interested to mesh the vertices of the first two ϑ-rings, therefore one starts by computing how many segments can be put into this set of points. Taking poloidal periodicity into account, this is simply obtained by taking the sum of the number of vertices of the two rings. Next, one takes the first vertex of the ring (for the first ring, the first vertex has index 1) and computes the indices of the neighboring vertices {2, 4, 5} by adding to the current vertex index the values 1, `n_verts_lower` and `n_verts_lower`+1, respectively. For these four indices of the current vertex itself and its neighbors, the poloidal coordinate components are evaluated and saved in counter clockwise order into variables `u`, `v`, `p` and `q`. This is necessary for further evaluation of the Delaunay condition. The current configuration is depicted in figure 1.11.

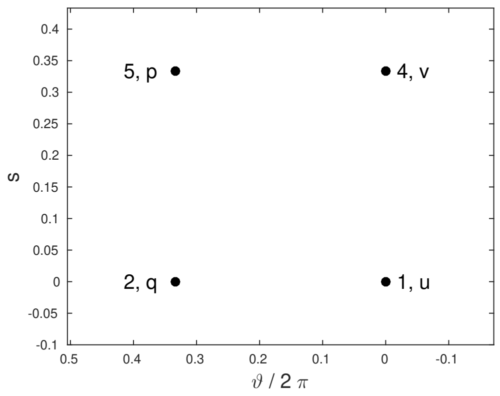


Figure 1.11: Extracted lower right corner domain of figure 1.9, for computing the Delaunay condition for the first prism, the poloidal coordinate components of the neighboring vertices to index 1 are saved into variables `u`, `v`, `p` and `q`, whereas for this case the coordinate tuples $[\vartheta, s]$ are `u = [0,0]`, `v = [0, 0.33]`, `p = [0.33,0.33]` and `q = [0.33,0]`

Next, one must propose a segment orientation (top facing or bottom facing) and compute the Delaunay condition accordingly. By default, for each ϑ-ring, the top facing orientation is initially proposed. Here, this means that the points `u`, `v` and `p` are linked to form the prism face. The Delaunay condition now states, that if one computes the circumcircle to the triangle spanned by these three points, the remaining point `q` may only lie outside this circumcircle or at most exactly on the circumcircle, in which case the Delaunay condition is ambiguous and both prism orientations are allowed. The Delaunay condition for the top facing triangle is hereby given by [?]

```
a = u(1) - q(1)
b = u(2) - q(2)
c = (u(1) - q(1)) ** 2.d0 + (u(2) - q(2)) ** 2.d0
d = v(1) - q(1)
e = v(2) - q(2)
f = (v(1) - q(1)) ** 2.d0 + (v(2) - q(2)) ** 2.d0
g = p(1) - q(1)
h = p(2) - q(2)
i = (p(1) - q(1)) ** 2.d0 + (p(2) - q(2)) ** 2.d0
delta = a*e*i + b*f*g + c*d*h - c*e*g - b*d*i - a*f*h
if (delta<=0.d0) then
     delaunay_condition = .true.
else
     delaunay_condition = .false.
endif
```

The result from this condition is, that if `delaunay_condition =.true.` is returned, the top facing configuration is accepted, if not, the bottom facing configuration with corner points `u`, `v` and `q` is selected instead. This is valid, as this procedure corresponds to the so-called Delaunay flip, which states that if four vertices are linked to form two adjacent triangles that do not satisfy the Delaunay condition (e.g. triangles {u, v, p} and {u, p, q}), one can always obtain two triangles that do satisfy the condition by flipping the orientation of the line segment that divides the two triangles (i.e. leading to new triangles {u, v, q} and {v, p, q}) [?].
Upon determining for `u`, `v`, `p` and `q` which proposed orientation is accepted, the next step is to associate the three vertices of the accepted configuration (i.e. {u, v, p} for top facing or {u, v, q} for bottom facing) with the poloidal face vertices of the prisms shown in figure 1.10. Here the three vertices either correspond to prism vertex indices {0, 2, 3} for the top facing or {0, 1, 2} for the bottom facing configuration. The grid vertex indices of the remaining prism vertices are found as follows.
In order to explain this process, one must first take a look at the binary representation of the prism vertex indices and the corresponding normalized offsets of the vertex

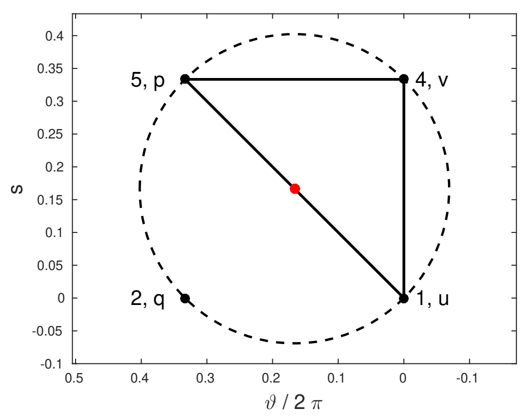


Figure 1.12: Visualization of the Delaunay condition for the top facing segment (solid line), if `q` lies outside or at most exactly on the circumcircle (dashed line, center marked by red dot) around {`u`, `v`, `p`} the Delaunay condition is satisfied. Since for the shown configuration all four vertices lie on the circumcircle the Delaunay condition is satisfied for both the top facing and bottom facing orientation, whenever this occurs the top facing orientation is assumed in this approach.

positions in symmetry flux space with respect to prism vertex 0 (e.g. for prism vertex 3 of the bottom facing configuration, the vertex has an offset in the s and ϑ direction, but not in the φ direction, thus the normalized offset in (φ, s, ϑ) is given by $(0, 1, 1)$). The binary representation of the prism vertex indices and the corresponding normalized positional offsets are given in tab. 1.4.

Table 1.4: Binary representation of prism vertices the corresponding normalized positional offsets

Index	Binary	φ-offset	s-offset	ϑ-offset
0	0 - 0 - 0	0	0	0
1	0 - 0 - 1	0	0	1
2	0 - 1 - 0	0	1	0
3	0 - 1 - 1	0	1	1
4	1 - 0 - 0	1	0	0
5	1 - 0 - 1	1	0	1
6	1 - 1 - 0	1	1	0
7	1 - 1 - 1	1	1	1

One can clearly see that the bitwise representation of the prism vertices coincides with the individual normalized positional offsets in directions φ, s and ϑ, respectively. Thus, due to the specific way in which the vertices are arranged, the indices of the prism vertices can be directly obtained from the normalized positional offsets via

```
vertex_index =base_idx + theta_offset + s_offset*n_verts_lower&   (1.7)
              &+ phi_offset*verts_per_slice ,
```

where `base_idx` denotes the index of the current grid vertex corresponding to prism index 0 and `n_verts_lower` is the number of poloidal grid points for the *lower* (i.e. smaller s-component) ϑ-ring. In this concrete example for the first grid segment, `base_idx` is equal to 1. Using 1.7, the remaining grid vertex indices for the corresponding prism vertices are obtained to fully define the prism in the symmetry flux coordinate space. Next, tetrahedra are constructed using grid vertex indices for the corresponding prism vertices. The way that the prism vertices are combined to form tetrahedra can be deduced from figure 1.10, for simplicity the tetrahedron vertex indices are also given in table 1.5.

Table 1.5: List of tetrahedron types with corresponding prism vertex indices for both top and bottom facing prism orientations

Tetrahedron type	Prism vertex indices	Prism orientation
1	0, 2, 3, 7	top facing
2	0, 2, 6, 7	top facing
3	0, 4, 6, 7	top facing
4	0, 1, 2, 6	bottom facing
5	0, 1, 5, 6	bottom facing
6	0, 4, 5, 6	bottom facing

Since now the indices of the tetrahedron vertices are known for the first three tetrahedra, their values are saved into the first three rows of `verts`. Quantities that remain unknown are `neighbours`, `neighbour_faces`, `perbou_phi` and `perbou_theta`. Next, one is interested in computing the `neighbours` with the associated `neighbour_faces` for the tetrahedra that belong to a given segment. For this, it must first be understood what types of neighbors exist for such a segment, for this it is helpful to take another look at figure 1.10. Here, one can see that there exist different tetrahedron boundaries on the inside of the segments as well as boundaries on the outside of the segment. More precisely, for three tetrahedra with four faces each, there exist 12 tetrahedral boundaries for any given segment. For instance for the top facing prism, there are four interior boundaries in the segment (internal boundaries are double-counted as they belong to two tetrahedra each), furthermore the segment has two boundaries pointing in the φ direction (with one pointing in $(+\varphi)$-direction and one pointing in $(-\varphi)$-direction), two more boundaries point in the $(+s)$-direction, the remaining four boundaries point to the neighboring tetrahedra on the same ϑ-ring, with two boundaries on the $(+\vartheta)$ side of the prism and the last two on the $(-\vartheta)$ side. For the bottom facing segment, this is analogous with the difference, that two boundaries now point in the $(-s)$-direction instead of the $(+s)$-direction. Moreover, to recapitulate, the definition of a neighbor with index i (1:4) to a given tetrahedron is the index of the tetrahedron that shares the three vertices that lie on the plane, spanned by the current tetrahedron vertices excluding the i^{th} vertex. For example, the third face of a tetrahedron lies on the plane spanned by tetrahedron vertices {1, 2, 4}, if this tetrahedron is for instance of type 1 the prism indices for this face would be {0, 2, 7} and the grid vertex indices would be elements {1, 2, 4} of the corresponding row in `verts`. The quantity `neighbour_faces(i)` with value j (1..4) is defined as the index of the face through which one enters the neighboring tetrahedron at face i (1..4). For example, given two tetrahedra with indices `ind_1` and `ind_2` that share a common plane on vertices {2, 3, 4} in the system of the first tetrahedron and vertices {1, 2, 3} for the second tetrahedron, the second tetrahedron is the first neighbor with respect to the first one, thus `neighbours(ind_1,1)=ind_2`. Furthermore one enters the second tetrahedron through its fourth plane, thus `neighbour_faces(ind_1,1)=4` for the first tetrahedron. Conversely, when starting from the second tetrahedron, the first tetrahedron is now the fourth neighbor and one enters the tetrahedron through the first face, therefore `neighbours(ind_2,4)=ind_1` and `neighbour_faces(ind_1,4)=1`. Generally, for any tetrahedron `neighbours(neighbours(ind_1,i),neighbour_faces(ind_1,i)) = ind_1` must hold for all `i` if there exists an adjacent tetrahedron at face `i`. A summary of all tetrahedral boundaries is given in table 1.6. Using this information, one can find the internal neighbors within the segment, furthermore, one can directly compute the indices of the neighbors in φ direction due to the axisymmetric configuration of the system. Here, one should only keep in mind, that when shifting the index of the current tetrahedron by the relative neighbor index of table 1.6, one must account for index periodicity in φ by subsequently shifting the resulting index into the valid regime of tetrahedron indices [1, `ntetr`], where `ntetr` denotes the total number of tetrahedra. Indices can hereby be shifted into this regime using the modulo function in

```
index_shifted = modulo(index-1,ntetr)+1 .
```

Table 1.6: This table shows a summary of all possible types of tetrahedron boundaries and if possible, the relative indices for the neighboring tetrahedron at this boundary with respect to the current tetrahedron index (however, not accounting for toroidal periodicity here, thus, indices must still be shifted into the valid domain [1,`ntetr`] using the modulo function). If a neighboring tetrahedron can be directly specified, also the index of the adjacent face of the neighbor is given. Question marks denote quantities that depend on how the grid is meshed and which are thus not yet specified at the point of tetrahedron creation.

Tetrahedron	Face	Vertices	Boundary	Relative neighb. index	n. Face
1	1	{2, 3, 7}	$+s$	?	?
Type: top facing	2	{0, 3, 7}	$+\vartheta$	?	?
Vertices:	3	{0, 2, 7}	internal	+1	3
{0, 2, 3, 7}	4	{0, 2, 3}	$-\varphi$	-`tetras_per_slice`+2	1
2	1	{2, 6, 7}	$+s$	?	?
Type: top facing	2	{0, 6, 7}	internal	+1	2
Vertices:	3	{0, 2, 7}	internal	-1	3
{0, 2, 6, 7}	4	{0, 2, 6}	$-\vartheta$	?	?
3	1	{4, 6, 7}	$+\varphi$	+`tetras_per_slice`−2	4
Type: top facing	2	{0, 6, 7}	internal	-1	2
Vertices:	3	{0, 4, 7}	$+\vartheta$	?	?
{0, 4, 6, 7}	4	{0, 4, 6}	$-\vartheta$	?	?
4	1	{1, 2, 6}	$+\vartheta$	?	?
Type: bot. facing	2	{0, 2, 6}	$-\vartheta$	?	?
Vertices:	3	{0, 1, 6}	internal	+1	3
{0, 1, 2, 6}	4	{0, 1, 2}	$-\varphi$	-`tetras_per_slice`+2	?
5	1	{1, 5, 6}	$+\vartheta$	?	?
Type: bot. facing	2	{0, 5, 6}	internal	+1	2
Vertices:	3	{0, 1, 6}	internal	-1	3
{0, 1, 5, 6}	4	{0, 1, 5}	$-s$	?	?
6	1	{4, 5, 6}	$+\varphi$	+`tetras_per_slice`−2	?
Type: bot. facing	2	{0, 5, 6}	internal	-1	2
Vertices:	3	{0, 4, 6}	$-\vartheta$	?	?
{0, 4, 5, 6}	4	{0, 4, 5}	$-s$	?	?

So far, a single segment has been created and split into tetrahedra. The corresponding tetrahedron vertices have been saved into the first three rows of `verts`. Furthermore, the neighbor indices for the internal tetrahedron boundaries of the segment as well as the neighbors in φ-direction have been identified together with the associated adjacent faces of the neighbors. These values are now saved into the corresponding fields of arrays `neighbours` and `neighbour_faces`, here it should be further noted that this array is first initialized with values -1 for all elements, this value indicates that no neighbor exists at the specified boundary and the domain of the grid ends. By overwriting the corresponding elements with their actual values, the tetrahedra are linked. Once this process is completed for all tetrahedra, the remaining elements which contain the value -1 correspond to the actual domain boundaries at $s =$ `s_min` and $s = 1$.

At this point of the code, however, there are still some remaining neighbors and neighboring faces for the current segment that cannot yet be computed as their values depend on how the other segments are oriented (which is again determined by the Delaunay condition). Therefore, two nested loops are implemented to construct all segments of the first φ-slice where the inner loop iterates over all segments of the current ϑ-ring and the outer loop iterates over all ϑ-rings. Within these loops, the approach of constructing the tetrahedra corresponds exactly to the example of the first three tetrahedra, as discussed above. After a segment is completed in the ϑ-ring, depending on its orientation, either the indices for {v, p} (if top facing) or for {u, q} (if bottom facing) need to be shifted by one (i.e. in $+\vartheta$-direction) in order to obtain the new point configuration to evaluate the Delaunay condition for the next segment, this is realized via integer offsets `upper_off` and `lower_off`. Due to poloidal periodicity, the indices for {u, v, p, q} must be taken as `modulo(index-1, verts_per_ring_upper)` and `modulo(index-1, verts_per_ring_lower)` for *upper* vertices {v, p} and *lower* vertices {u, q}, respectively. This ensures that the last vertices of the current ring are correctly linked with the first vertices. Due to toroidal symmetry the remaining tetrahedra of the full grid can actually be easily obtained upon finishing the first slice by subsequently shifting the tetrahedra vertices by the number of points per slice as well as shifting the neighbor indices by the number of tetrahedra per slice. For a better overview, the whole procedure of the tetrahedron generation is given as pseudo-code.

```
do ring=1,n_rings
    set index offsets for u, v, p, q to zero
    set reference point (index for first u) index to 1
    set prism index to 1
```

```
do segment=1,prisms_per_ring(ring)
     compute u, v, p, q from reference point and offsets
     evaluate Delaunay_condition(u, v, p, q) for top facing prism
     if (Delaunay_condition==.true.)  then
          make top facing prism from normalized positional &
               & offsets
          save current prism index in top_facing_prisms for &
               & neighbor indexing with next ring
          index tetrahedra from top facing prism
          increment index offset for upper vertices v, p
     else
          make bottom facing prism from normalized positional &
               & offsets
          index tetrahedra from bottom facing prism
          if (.not.  ring == 1) then
               find neighbors with corresponding prism &
                    & in top_facing_prisms
          endif
          increment index offset for lower vertices u, q
     endif
     find internal neighbors
     find neighbors to adjacent phi slices
     if (segment ==1) then
          find periodic boundary faces in -theta direction
     endif
     if (.not.  segment ==1 ) then
          find neighbors with the previous segment
     endif
     if (segment == prisms_per_ring(ring)) then
          find periodic boundary faces in +theta direction
     endif
enddo
```

```
    find neighbors between first and last segment in ring
    shift reference point to next ring by incrementing the index &
        & by n_verts_lower
    save tetrahedron quantities based on &
        & ind_tetra(i) = (prism index-1)*3+i for i = [1..3] &
        & and increment prism index by 1
enddo
shift tetrahedra indices to obtain tetrahedra of all remaining slices
find periodic boundary faces in phi direction
```

There are two code elements that were not explained yet, the first being how unknown neighbors between two adjacent prisms are found and the second being how the periodic boundaries are determined. The first code element is implemented in subroutine `connect_prisms`. This subroutine takes arguments `(prism_1_idx, prism_2_idx, verts, neighbours, neighbour_faces)`, where the first three arguments are input quantities that define which adjacent prisms are being linked as well as the information which vertices these prisms contain. The latter two arguments are the two-dimensional arrays where the found neigbor properties are saved in the corresponding fields, these are, thus, inout quantities. The approach in this subroutine is in fact very simple, one constructs two nested loops where each loop iterates over the individual tetrahedra of one of the prisms. Within these loops the vertex indices of the two tetrahedra are compared, if three vertices coincide a neighbor has been found. In such a case one has to further find for both tetrahedra which point is not contained in the other tetrahedron. Here, the position of the index in the four element vertex array of the tetrahedron which is not included in the other tetrahedron directly corresponds to the value of `neighbour_faces` of the adjacent tetrahedron. Once a neighbor and its corresponding neighbor face have been found, the values are saved accordingly. At the end of iteration, all neighbors of the two input prisms are successfully linked, if the two prisms are indeed adjacent.
For the second code element, the periodicity boundaries need to be treated separately for ϑ and φ, whereas both are identically initialized with the value 0 for all elements. In the ϑ-direction, depending on the orientation, the prisms at the poloidal boundary can be directly indexed according to the results of 1.6. If for instance a top facing prism is the first segment in a ring, the boundaries in $-\vartheta$-direction are given by the fourth face of the second tetrahedron and the fourth face of the third tetrahedron

inside the prism, respectively. Thus, the value for the periodic boundary is then saved as `theta_perbou(ind_tetr,i)=-1`, where `ind_tetr` denotes the tetrahedron index and `i` the index of the face adjacent to the boundary. This is analogous for the $\vartheta = 2\pi$-boundary with the differences that here the neighbors in positive ϑ-direction are considered and a value of 1 is assumed in `theta_perbou`. For the φ-direction, however, the process is in fact much simpler as it can be directly solved via indexing as both prism types have their first tetrahedron with its fourth face adjacent to the $-\varphi$ boundary and their third tetrahedron with its first face adjacent to the $+\varphi$ boundary. Thus, one can directly write

```
perbou_phi = 0
perbou_phi(4, :tetras_per_slice:3) = -1
perbou_phi(1, n_tetras - tetras_per_slice + 3::3) = 1
```

Now, all necessary tetrahedron quantities have been successfully computed and returned to the overlying subroutine `make_grid_aligned`. The last step is to allocate the tetrahedra according to the type `tetrahedron _grid` and set the necessary quantities by.

```
allocate(tetra_grid(1:ntetr))
do i=1,ntetr
     tetra_grid(i)%ind_knot = verts(:, i)
     tetra_grid(i)%neighbour_tetr = neighbours(:, i)
     tetra_grid(i)%neighbour_face = neighbour_faces(:, i)
     tetra_grid(i)%neighbour_perbou_phi(:)  = perbou_phi(:, i)
     tetra_grid(i)%neighbour_perbou_theta(:)  = perbou_theta(:, i)
enddo
```

The field aligned grid is now finished. In order to visualize the resulting grid, some figures are shown below.

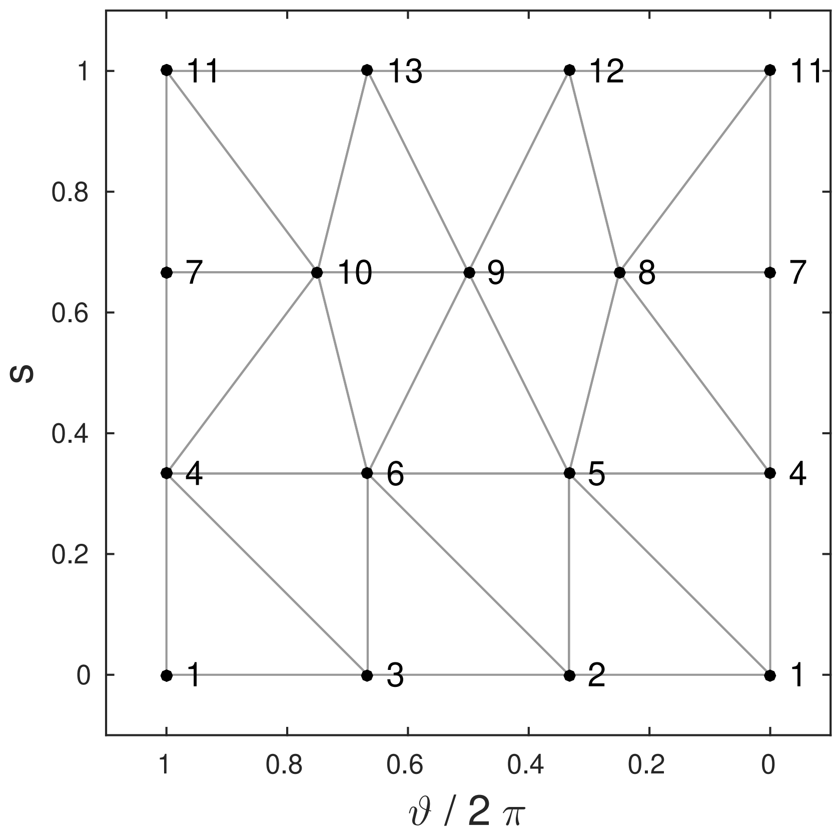


Figure 1.13: Depiction of the field aligned grid in symmetry flux coordinates with an increased number of poloidal vertices at $s = 2/3$, vertex coordinates were extended poloidally to 2π to make the representation cleaner

One can see in figure 1.13, that even though a variable number of vertices was used for the different ϑ-rings, the algorithm produces a very well aligned grid using the Delaunay condition.

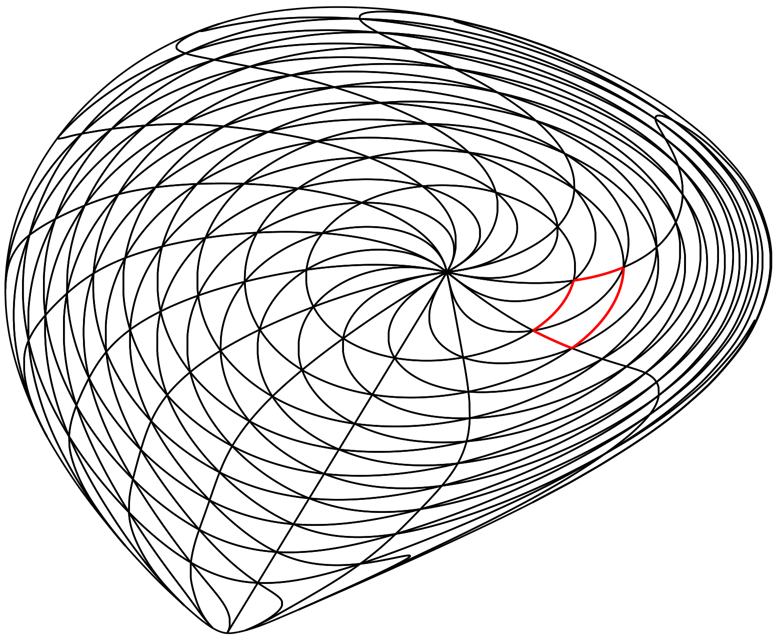


Figure 1.14: Poloidal projection of the field aligned grid in real space, the cross sectional countour of the two combined prisms from figure 1.15 is marked in red

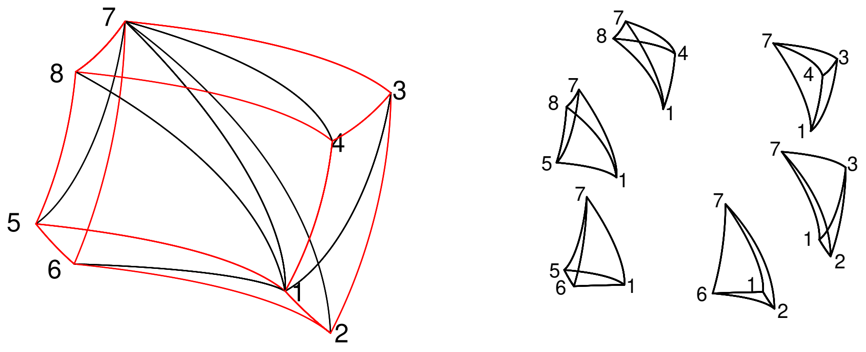


Figure 1.15: On the left, two adjacent prisms of opposing orientation are drawn in real space with the red lines indicating how the two prisms form a hexagonal shape for a constant number of points per ϑ-ring, the individual tetrahedra are plotted on the right, the corners are hereby merely indexed to enable a clearer association of the tetrahedra