AFFIDAVIT

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master‘s thesis.

Date

Signature

Abstract

This is a placeholder for the abstract. It summarizes the whole thesis to give a very short overview. Usually, this the abstract is written when the whole thesis text is finished.

Contents

List of Figures

Chapter 1

Introduction

1.1 Overview - GORILLA

..why grid is needed.., what is the goal (efficient and noise-insensitive computation of guiding center orbits and of transport coefficients)

Chapter 2

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`.

2.1 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.

2.2 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.

2.2.1 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.

2.2.2 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:

$$ind(i, j, k) = k + (i - 1) \cdot N_Z + (j - 1) \cdot N_Z \cdot N_R \tag{2.1}$$

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 2.1.
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 2.1 and then saving the correct corner point indices for each of the six types of tetrahedra. Upon generation of each tetrahedron a separate

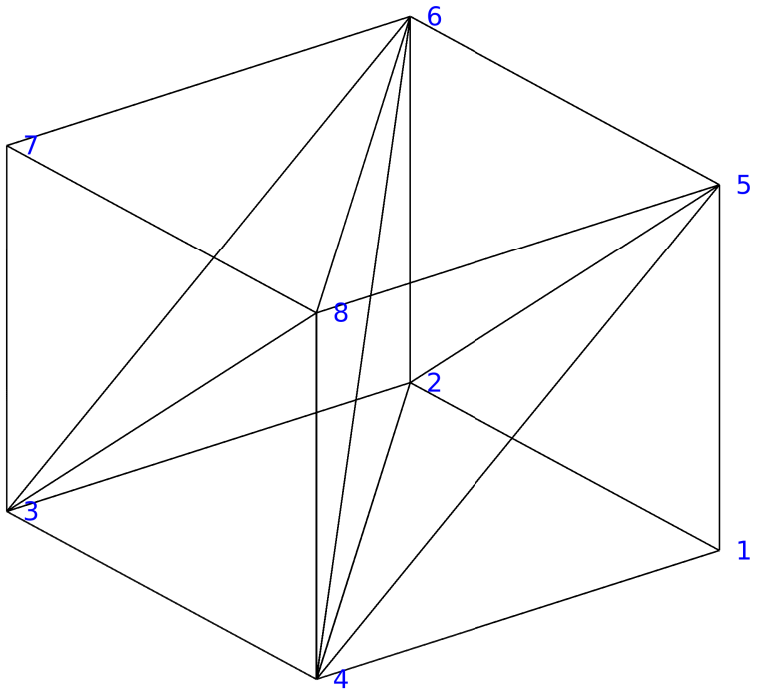


Figure 2.1: Hexahedron split up into six tetrahedra

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.

2.2.3 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 2.2.2 using 2.1 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):

$$ind_{\text{tetra}}(i,j,k,l) = 6(ind_{\text{hexa}}(i,j,k) - 1) + l = \tag{2.2}$$
$$= 6(k - 1 + (i-1)\cdot N_Z + (j-1)\cdot N_Z \cdot N_R) + l$$
$$l = 1, 2, .., 6 \tag{2.3}$$

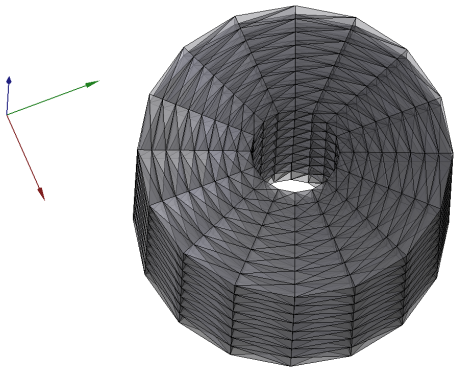
All neighboring tetrahedra must either lie within the same hexahedron as the reference tetrahedron or within a neighboring hexahedron. For this reason it is convenient to use formula 2.2 to obtain the indices of all tetrahedra that are within the same or the neighboring hexahedra. One can then loop over 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.

2.2.4 Periodic boundary conditions

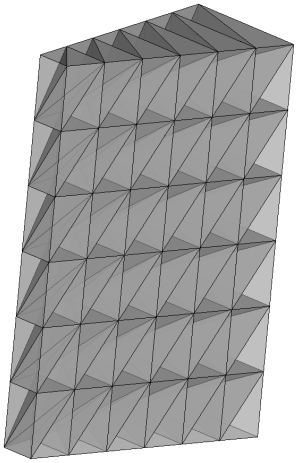
Furthermore, since this grid is constructed using cylindrical coordinates, periodicity in the φ coordinate occurs which needs to be treated independently. The issue is that vertices lying on the $\varphi = 0$ plane have both values 0 and 2π in the φ component and particles that move through the boundary 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.

2.2.5 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 2.2a. 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 2.2b 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

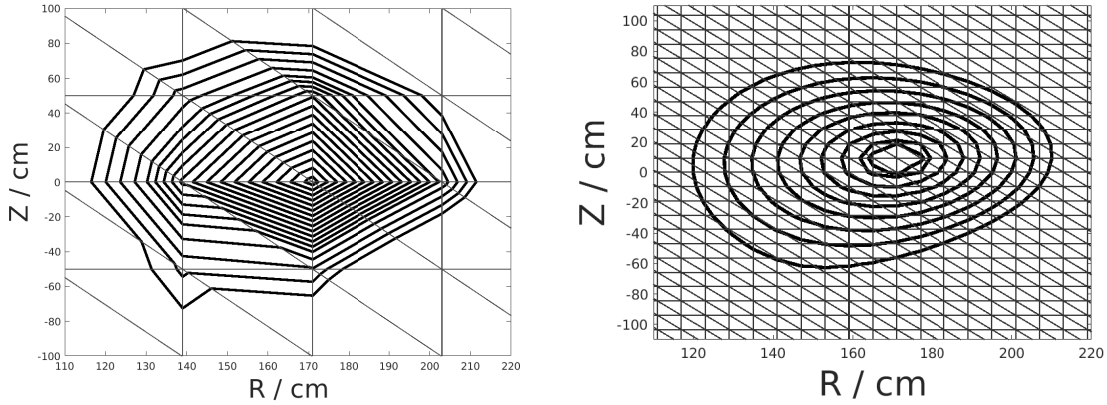


(b) Single φ slice of cylindrical contour grid in real space

2.2.6 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 2.3a and 2.3b:



(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 2.3b, 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 [1], 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.

2.3 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.

2.3.1 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 [1]. Mathematically this translates to the set of differential equations

$$\frac{\mathrm{d}R}{\mathrm{d}\varphi} = \frac{B^R}{B^\varphi}, \ \frac{\mathrm{d}Z}{\mathrm{d}\varphi} = \frac{B^Z}{B^\varphi}, \tag{2.4}$$

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 2.4 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

$$q_s = \frac{B^\varphi}{B^\vartheta} \quad \text{to} \quad \mathrm{d}\varphi = q_s \mathrm{d}\vartheta. \tag{2.5}$$

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 2.4 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.

2.3.2 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

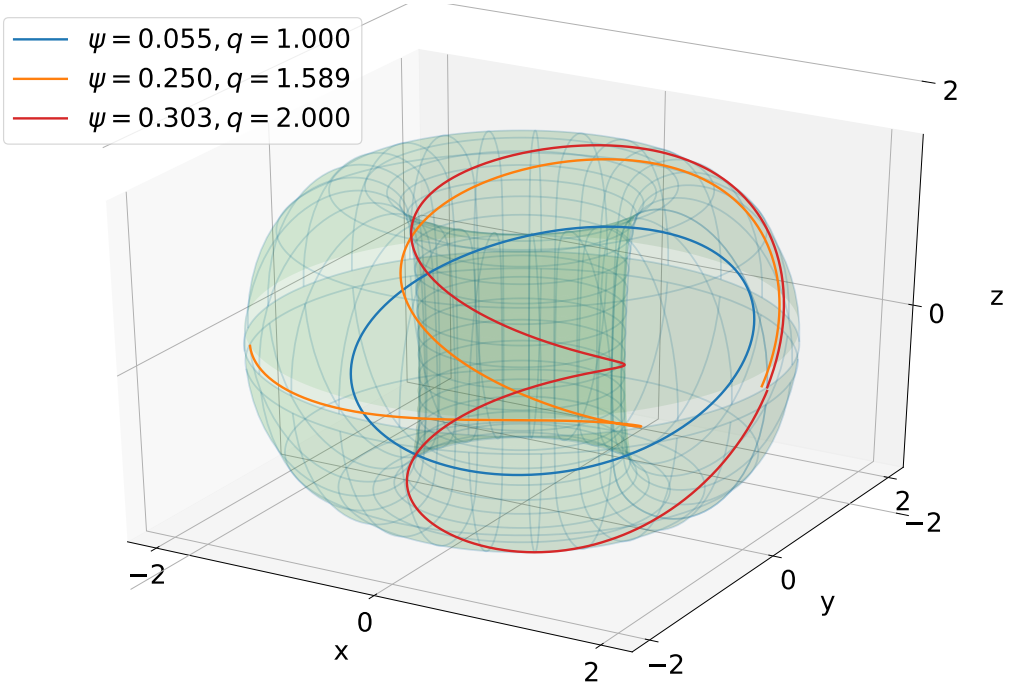


Figure 2.4: Field lines with different safety factors in a tokamak

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 2.4 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

$$p_i = \vec{\mathrm{O}} + \frac{i}{N} \begin{pmatrix} rmx - O_R \\ 0 \end{pmatrix} \tag{2.6}$$

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 2.5 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

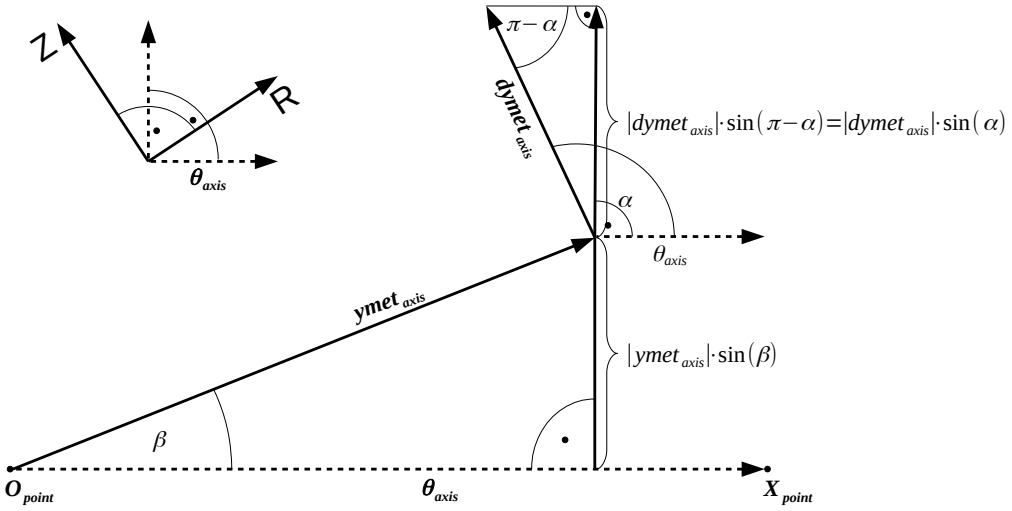


Figure 2.5: Visualization of quantities for Newton's method

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 Appendix A, 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 2.1.

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 2.1: 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

2.3.3 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 2.6.

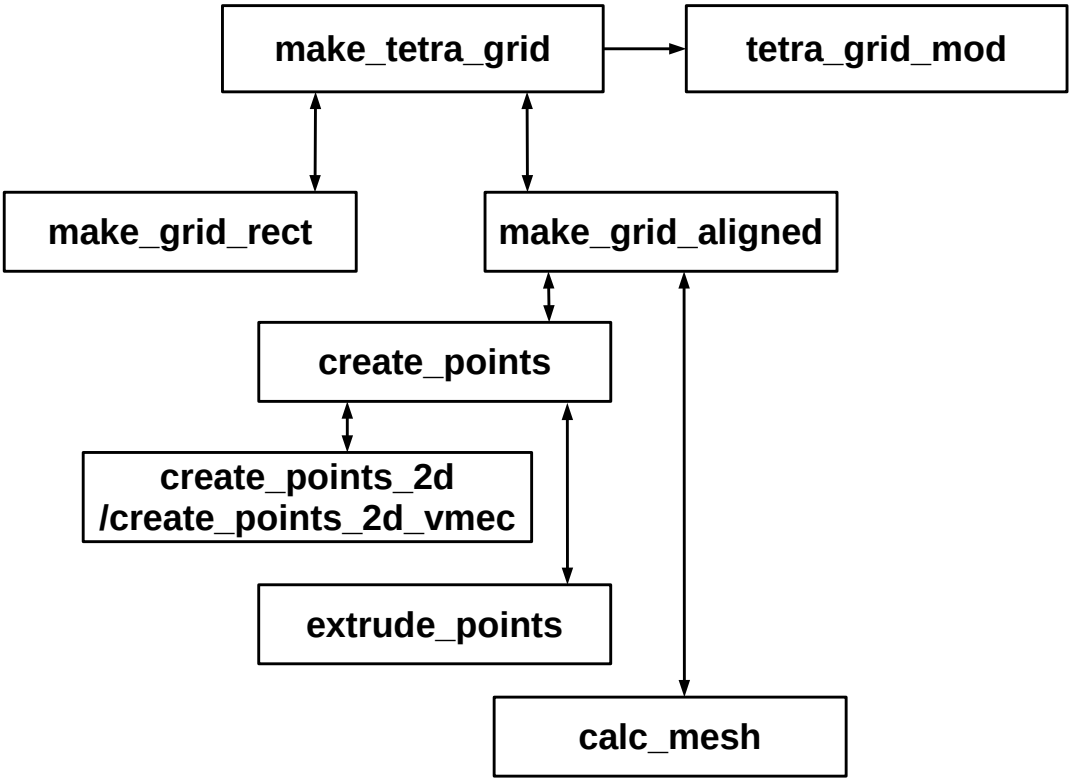


Figure 2.6: Subroutine `make_tetra_grid` code structure

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 un-

intended 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 2.2 and 2.3. All call parameters of the grid constructing subroutines are hereby briefly explained.

Table 2.2: 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 2.3: 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 2.6 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

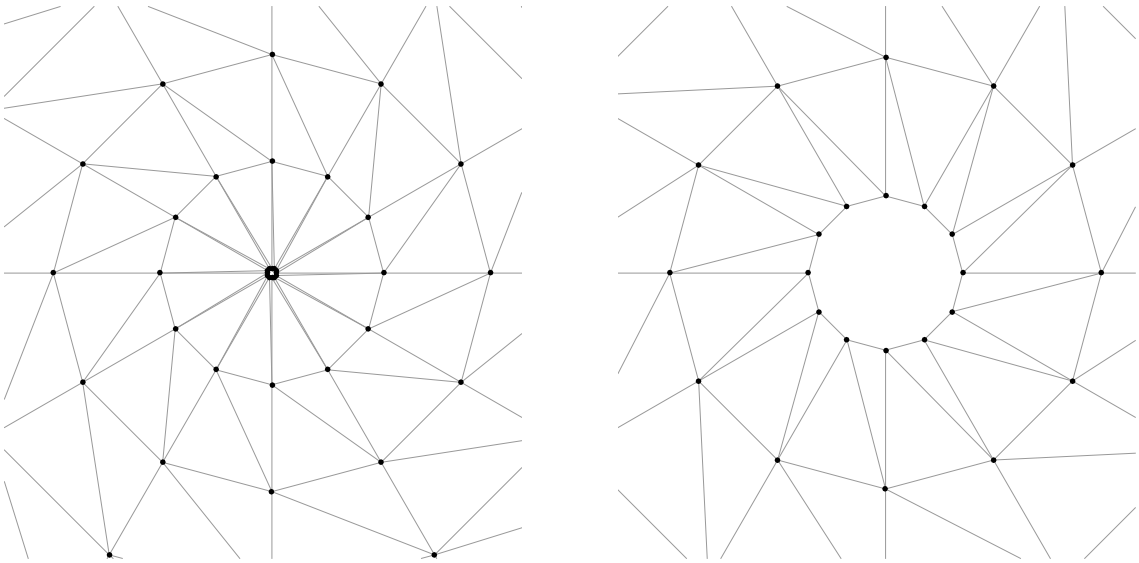
Upon calling the subroutine `make_grid_aligned`, first `create_points` is executed with arguments (`verts_per_ring, nphi, verts_rphiz, verts_sthetaphi, efit_vmec, field_periodicity, nvert, r_scaling_func,theta_scaling_func, repeat_center_point`), where `verts_per_ring` denotes an array where each element 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 2.7.



(a) (b)

Figure 2.7: 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 2.7a a very small value for `s_min` has been chosen, this is compared to a larger value for plot 2.7b. It is clearly visible that the tetrahedral faces which have two corner points on the inner-most ϑ-ring are only visible in 2.7b, while they appear to be merely lines in 2.7a.

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

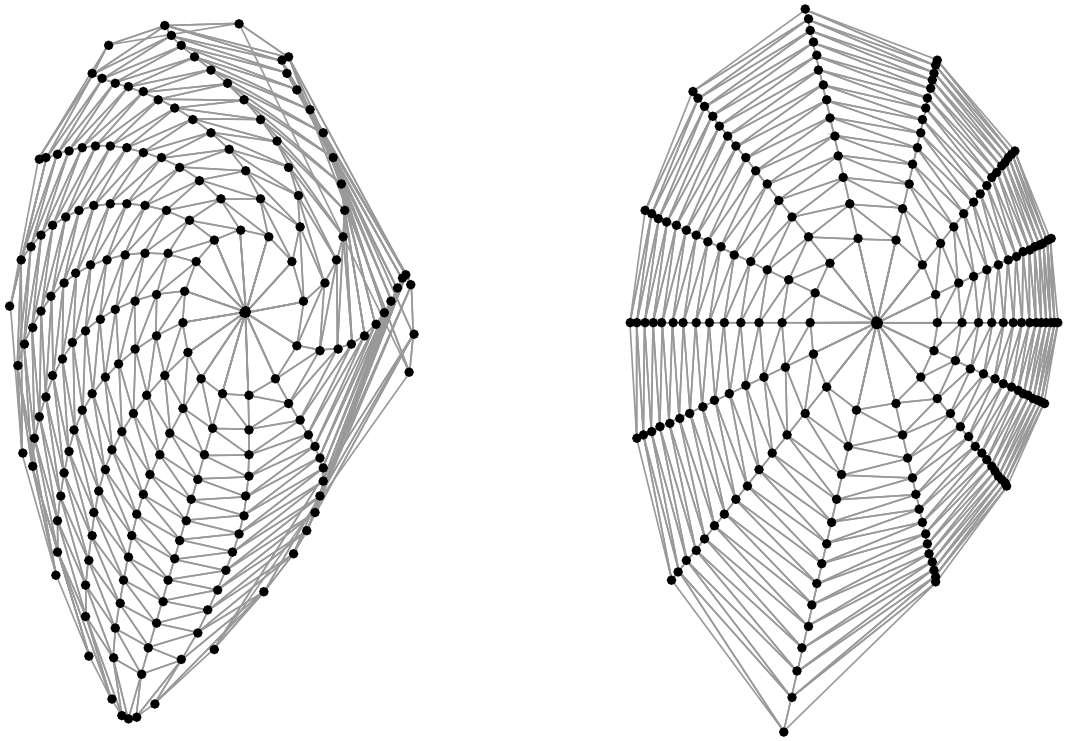
here, assume a tetrahedron and drag one of the vertices through the opposing plane spanned up by the remaining three vertices, the resulting tetrahedron is literally turned inside out, this occurs as the outermost point is increasingly shifted poloidally towards the X-point). A negative scalar product 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 2.8, once with vertices poloidally equidistant in symmetry flux coordinate angle ϑ (2.8a) and once with vertices poloidally equidistant in geometric angle θ (2.8b). 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 2.8a 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 2.8b 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 2.8: 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 2.8a intersections occur close to the separatrix, while for 2.8b 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_t heta, repeat_center_point = .true.)`. This is called from the subroutine `make_gr id_aligned`, as shown in figure 2.6. 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 2.3, 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 2.9. 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 $\Delta\varphi = 2\pi/$`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 2.9.

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

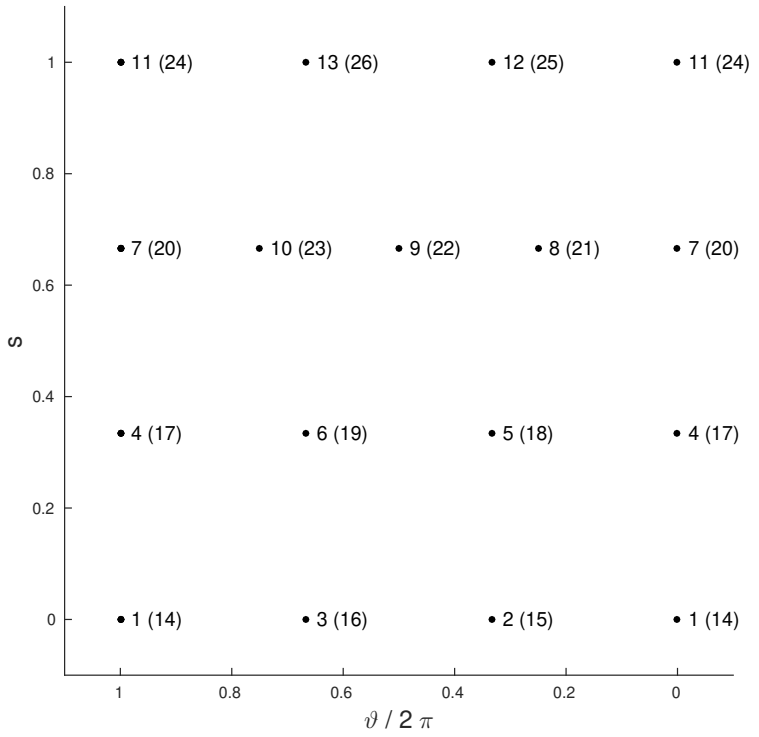


Figure 2.9: Grid vertex indices for the first φ-slice and in parentheses the indices of the next slice, the indices between slices differ by a constant value `verts_per_slice` which is equal to 13 for this configuration

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 2.10. 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 2.9. 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.

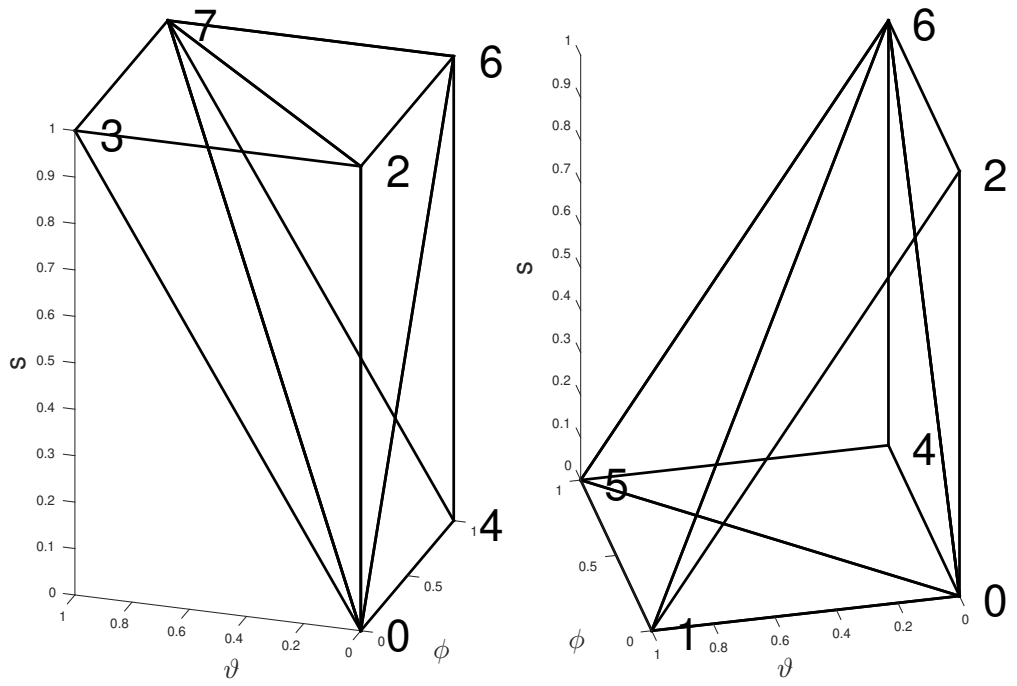


Figure 2.10: Prisms used to index tetrahedra, top facing prism on the left and bottom facing prism on the right

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 2.11.

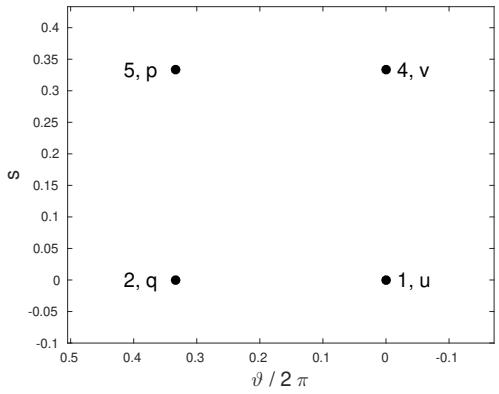


Figure 2.11: Extracted lower right corner domain of figure 2.9, for computing the Delaunay condition for the first prism, the poloidal coordinate components of the neighboring vertices to index 1 are saved into variables `u`, `v`, `p` and `q`, whereas for this case the coordinate tuples $[\vartheta, s]$ 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 [8]

```
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}) [8].
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 2.10. 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

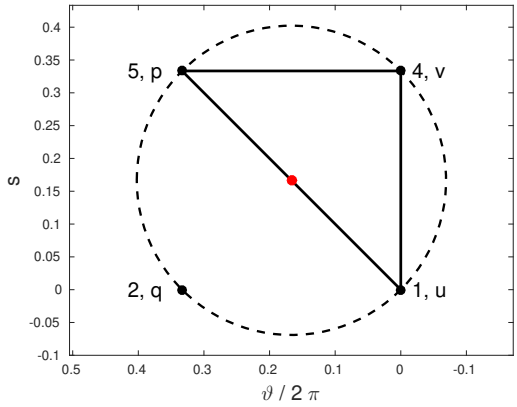


Figure 2.12: Visualization of the Delaunay condition for the top facing segment (solid line), if q lies outside or at most exactly on the circumcircle (dashed line, center marked by red dot) around {u, v, p} the Delaunay condition is satisfied. Since for the shown configuration all four vertices lie 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. 2.4.

Table 2.4: 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

$$\begin{aligned}\texttt{vertex\_index} =& \texttt{base\_idx + theta\_offset + s\_offset*n\_verts\_lower\&} \\ & \texttt{\&+ phi\_offset*verts\_per\_slice}\,,\end{aligned} \tag{2.7}$$

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 2.7, 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 2.10, for simplicity the tetrahedron vertex indices are also given in table 2.5.

Table 2.5: 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 2.10. 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 2.6. 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 2.6, 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 2.6: 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 2.6. 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.

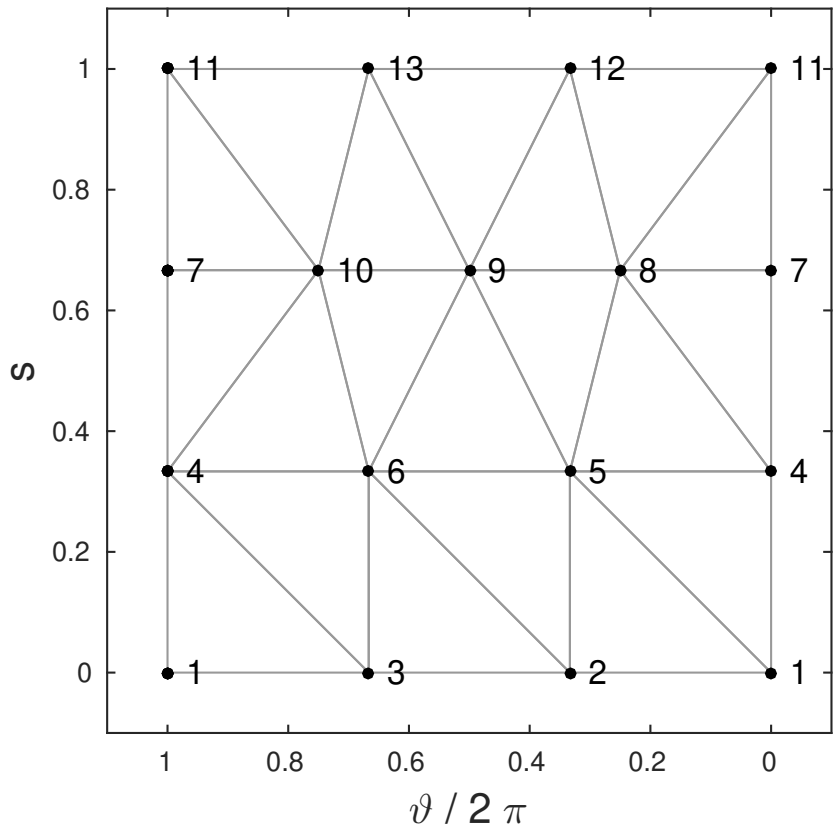


Figure 2.13: Depiction of the field aligned grid in symmetry flux coordinates with an increased number of poloidal vertices at $s = 2/3$, vertex coordinates were extended poloidally to 2π to make the representation cleaner

One can see in figure 2.13, 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.

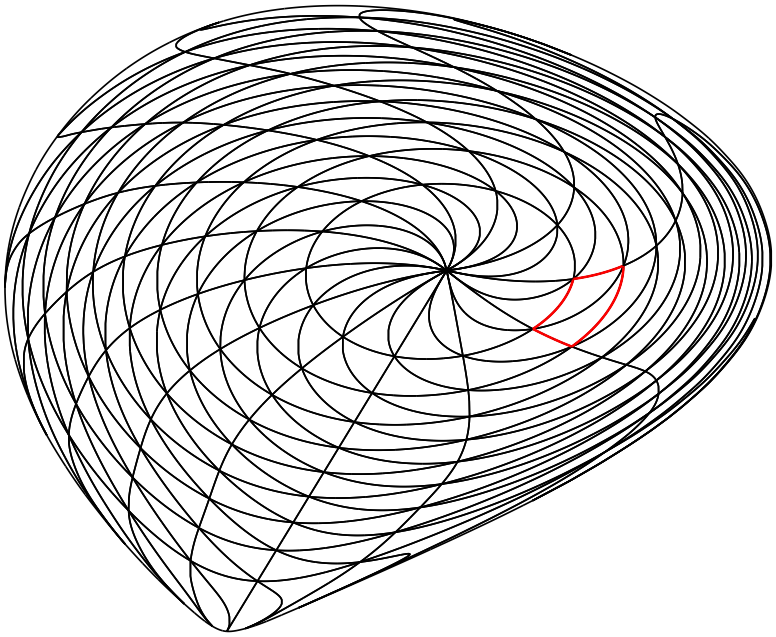


Figure 2.14: Poloidal projection of the field aligned grid in real space, the cross sectional countour of the two combined prisms from figure 2.15 is marked in red

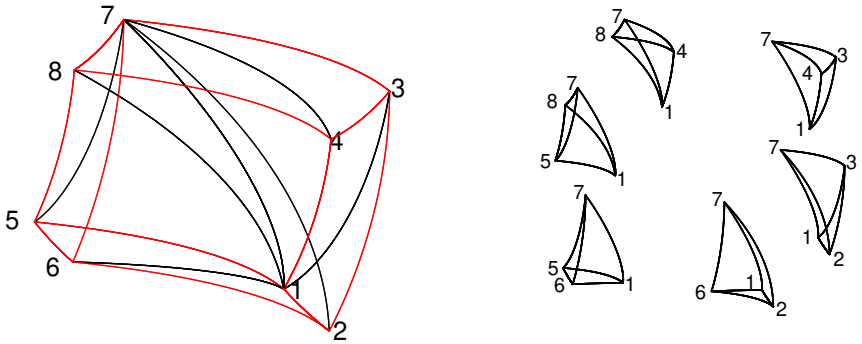


Figure 2.15: On the left, two adjacent prisms of opposing orientation are drawn in real space with the red lines indicating how the two prisms form a hexagonal shape for a constant number of points per ϑ-ring, the individual tetrahedra are plotted on the right, the corners are hereby merely indexed to enable a clearer association of the tetrahedra

Chapter 3

Analytical treatment of equations of motion in *Gorilla*

3.1 Analytical solution of equations of motion

In this section, an analytic solution to the linear equations of motion within a tetrahedron will be derived, following the formulation of guiding center equations by *Solov'ev* and *Morozov* [5] and the local linearization by M. Eder *et al* [7].
By denoting the extended set of variables with z^i, where $z^i = x^i$ for $i = 1, 2, 3$ and $z^4 = v_\parallel$, the linearized equation set assumes a standard form

$$\frac{\mathrm{d}z^i(\tau)}{\mathrm{d}\tau} = a^i_l(\tau) z^l(\tau) + b^i, \tag{3.1}$$

where for $i, l = 1, 2, 3$ the matrix elements are

$$\begin{aligned}
a^i_l &= \varepsilon^{ijk}\left(2\frac{\partial U}{\partial x^l}\frac{\partial}{\partial x^j}\frac{B_k}{\omega_c} + \frac{\partial U}{\partial x^j}\frac{\partial}{\partial x^l}\frac{B_k}{\omega_c}\right) \qquad \text{for} \qquad 1 \le i, l \le 3, \\
a^i_4 &= \varepsilon^{ijk}\frac{\partial A_k}{\partial x^j} \qquad \text{for} \qquad 1 \le i \le 3, \\
a^4_l &= 0 \qquad \text{for} \qquad 1 \le l \le 3, \\
a^4_4 &= \varepsilon^{ijk}\frac{\partial U}{\partial x^i}\frac{\partial}{\partial x^j}\frac{B_k}{\omega_c},
\end{aligned} \tag{3.2}$$

and the components of vector b^i are

$$\begin{aligned}
b^i &= \varepsilon^{ijk}\left(2U_0\frac{\partial}{\partial x^j}\frac{B_k}{\omega_c} + \left(\frac{B_k}{\omega_c}\right)_0\frac{\partial U}{\partial x^j}\right) \qquad \text{for} \qquad 1 \le i \le 3, \\
b^4 &= \varepsilon^{ijk}\frac{\partial U}{\partial x^i}\frac{\partial A_k}{\partial x^j}.
\end{aligned} \tag{3.3}$$

Mind, that for all $i, l = 1, .., 4$ the coefficients a^i_l and b^i remain constant within the scope of pushing through a tetrahedron. More precisely, since $U = v^2_\parallel/2$, only U_0 (the

value of U at the first vertex of the tetrahedron) and the gradient $\partial U/\partial x^i$ depend on initial conditions (x^i, $v_\parallel$ and $v_\perp$) of the particle entering a given tetrahedron, all other quantities (as well as some components of U) are independent of initial conditions and, thus, can be precomputed upon generation of the array `tetra_physics` in module `tetra_physics_mod`.

3.1.1 Reduction to a set of three linear ODEs

One is now interested in finding an analytic expression for $z^i(\tau)$.
Here, one can start by looking at the fourth component which is the parallel velocity as a function of time. One can quickly show that this equation is in fact decoupled from x^i, allowing for it to be solved independently, resulting to

$$v_\parallel(\tau) = \left(v_\parallel(0) + \frac{b}{a}\right) \mathrm{e}^{a\tau} - \frac{b}{a}, \tag{3.4}$$

with a and b being

$$a = \varepsilon^{ijk} \frac{\partial U}{\partial x^i} \frac{\partial}{\partial x^j} \frac{B_k}{\omega_c}, \qquad b = \varepsilon^{ijk} \frac{\partial U}{\partial x^i} \frac{\partial A_k}{\partial x^j},$$

which are both constant within the linearized fields. For compactness, this expression will be abbreviated for further calculation using $\eta = (v_\parallel(0) + \frac{b}{a})$ and $\theta = (\frac{b}{a})$, thus,

$$v_\parallel(\tau) = \eta \mathrm{e}^{a\tau} - \theta. \tag{3.5}$$

The set of differential equations can now be formulated in a way that all time dependence is confined to a driving term. In order to achieve this, one starts with equation 3.1 and takes only the first three components into account, which yield for $i = 1, 2, 3$

$$\begin{aligned} \frac{\mathrm{d}x^i(\tau)}{\mathrm{d}\tau} &= a_l^i x^l(\tau) + a_4^i z^4(\tau) + b^i \\ &= \underbrace{\varepsilon^{ijk}\left(2\frac{\partial U}{\partial x^l}\frac{\partial}{\partial x^j}\frac{B_k}{\omega_c} + \frac{\partial U}{\partial x^j}\frac{\partial}{\partial x^l}\frac{B_k}{\omega_c}\right)}_{=a_l^i} x^l(\tau) + \underbrace{\varepsilon^{ijk}\left(\frac{\partial A_k}{\partial x^j}\right)}_{=a_4^i} \underbrace{z^4(\tau)}_{=v_\parallel(\tau)} \\ &+ \underbrace{\varepsilon^{ijk}\left(2U_0\frac{\partial}{\partial x^j}\frac{B_k}{\omega_c} + \left(\frac{B_k}{\omega_c}\right)_0 \frac{\partial U}{\partial x^j}\right)}_{=b^i}. \end{aligned} \tag{3.6}$$

Furthermore, using $U(x^i) = U_0 + x^i \frac{\partial U}{\partial x^i}$ and $U = \frac{v_\parallel^2}{2}$ simplifies this equation to

$$\frac{\mathrm{d}x^i(\tau)}{\mathrm{d}\tau} = \underbrace{\varepsilon^{ijk}\frac{\partial U}{\partial x^j}\frac{\partial}{\partial x^l}\frac{B_k}{\omega_c}}_{=\tilde{a}_l^i} x^l(\tau) + \underbrace{v_\parallel(\tau)\varepsilon^{ijk}\frac{\partial A_k}{\partial x^j} + v_\parallel^2(\tau)\varepsilon^{ijk}\frac{\partial}{\partial x^j}\frac{B_k}{\omega_c} + \varepsilon^{ijk}\left(\frac{B_k}{\omega_c}\right)_0\frac{\partial U}{\partial x^j}}_{q^i(\tau)},$$

which can be compactly written as

$$\frac{\mathrm{d}x^i(\tau)}{\mathrm{d}\tau} = \tilde{a}_l^i x^l(\tau) + q^i(\tau), \tag{3.7}$$

where

$$\tilde{a}_l^i = \varepsilon^{ijk}\frac{\partial U}{\partial x^j}\frac{\partial}{\partial x^l}\frac{B_k}{\omega_c} \tag{3.8}$$

is a constant matrix and the driving term $q^i(\tau)$ is explicitly given by

$$q^i(\tau) = \varepsilon^{ijk}\left(v_\parallel(\tau)\frac{\partial A_k}{\partial x^j} + v_\parallel^2(\tau)\frac{\partial}{\partial x^j}\frac{B_k}{\omega_c} + \left(\frac{B_k}{\omega_c}\right)_0\frac{\partial U}{\partial x^j}\right). \tag{3.9}$$

For compactness, this is abbreviated as

$$\begin{aligned} q^i(\tau) &= v_\parallel(\tau)\underbrace{\varepsilon^{ijk}\left(\frac{\partial A_k}{\partial x^j}\right)}_{\alpha^i} + v_\parallel^2(\tau)\underbrace{\varepsilon^{ijk}\frac{\partial}{\partial x^j}\frac{B_k}{\omega_c}}_{\beta^i} + \underbrace{\varepsilon^{ijk}\left(\frac{B_k}{\omega_c}\right)_0\frac{\partial U}{\partial x^j}}_{\gamma^i} \\ &= v_\parallel(\tau)\alpha^i + v_\parallel^2(\tau)\beta^i + \gamma^i. \end{aligned} \tag{3.10}$$

The terms α^i, β^i and γ^i are constant within a tetrahedral cell, where the electromagnetic fields are linear. Thus, all time dependence arises therefore from $v_\parallel(\tau)$.

3.1.2 Homogeneous solution to equation of motion

The next step of calculating the analytical solution to the homogeneous part of equation 3.7 is to start with the following Ansatz

$$\vec{x} = e^{\lambda\tau}\vec{\psi}\,. \tag{3.11}$$

Using this, equation 3.1 yields the eigenvalue equation

$$\lambda e^{\lambda\tau}\vec{\psi} = \hat{a}e^{\lambda\tau}\vec{\psi}\,, \tag{3.12}$$

where $\hat{a}$ denotes matrix a_l^i, λ the associated eigenvalues and ψ the eigenvectors, respectively. Therefore, the general solution to the homogeneous differential equation

can be written as

$$\vec{x}_{(h)}(\tau) = C_1 e^{\lambda_1 \tau}\vec{\psi_1} + C_2 e^{\lambda_2 \tau}\vec{\psi_2} + C_3 e^{\lambda_3 \tau}\vec{\psi_3}. \tag{3.13}$$

Here, the C_l denote a vector of arbitrary constants given by initial conditions of the problem. Equation 3.13 can be rewritten using index notation

$$x^i_{(h)}(\tau) = \psi^i_l C^l e^{\lambda^l \tau}, \tag{3.14}$$

where ψ^i_l is a matrix with eigenvectors $\vec{\psi_l}$ as columns, each corresponding to the respective eigenvalues λ^l.

3.1.3 Particular solution: variation of constants

Next, the particular solution to the inhomogeneous ordinary differential equation set 3.1 will be derived. In order to do this, the method of variation of constants is applied. With this approach, the coefficients $\widetilde{C}_l$ are treated as functions of τ

$$\vec{x}_{(p)}(\tau) = \widetilde{C}_1(\tau) e^{\lambda_1 \tau}\vec{c_1} + \widetilde{C}_2(\tau) e^{\lambda_2 \tau}\vec{c_2} + \widetilde{C}_3(\tau) e^{\lambda_3 \tau}\vec{c_3}, \tag{3.15}$$

which again can be denoted in index notation

$$x^i_{(p)}(\tau) = \psi^i_l \widetilde{C}^l(\tau) e^{\lambda^l \tau}. \tag{3.16}$$

Calculating the derivative of this equation yields

$$\begin{aligned} \frac{\mathrm{d}\vec{x}}{\mathrm{d}\tau} &= \widetilde{C}'_1(\tau) e^{\lambda_1 \tau}\vec{\psi_1} + \widetilde{C}_1(\tau)\lambda_1 e^{\lambda_1 \tau}\vec{\psi_1} \\ &\quad + \widetilde{C}'_2(\tau) e^{\lambda_2 \tau}\vec{\psi_2} + \widetilde{C}_2(\tau)\lambda_2 e^{\lambda_2 \tau}\vec{\psi_2} \\ &\quad + \widetilde{C}'_3(\tau) e^{\lambda_3 \tau}\vec{\psi_3} + \widetilde{C}_3(\tau)\lambda_3 e^{\lambda_3 \tau}\vec{\psi_3}\ . \end{aligned} \tag{3.17}$$

The expression is subsequently inserted into equation 3.7, this leads to

$$\vec{q}(\tau) = \widetilde{C}'_1(\tau) e^{\lambda_1 \tau}\vec{\psi_1} + \widetilde{C}'_2(\tau) e^{\lambda_2 \tau}\vec{\psi_2} + \widetilde{C}'_3(\tau) e^{\lambda_3 \tau}\vec{\psi_3}, \tag{3.18}$$

which can be compactly rewritten as

$$q^i(\tau) = \psi^i_l \widetilde{C}'^l(\tau) e^{\lambda^l \tau} \tag{3.19}$$

using index notation. The next step in calculating the coefficients $\widetilde{C}^l(\tau)$ is to multiply the eigenvectors ψ_l^i with the inverse matrix $\bar{\psi}_i^j$, where $\bar{\psi}_i^j \psi_l^i = \psi_i^j \bar{\psi}_l^i = \delta_l^j$

$$\bar{\psi}_i^j q^i(\tau) = \underbrace{\bar{\psi}_i^j \psi_l^i}_{\delta_l^j} \widetilde{C}'^l(\tau) e^{\lambda^l \tau}. \tag{3.20}$$

$$\widetilde{C}'^l(\tau) = \bar{\psi}_i^l q^i(\tau) e^{-\lambda^l \tau} \tag{3.21}$$

One can now integrate this expression from 0 to τ since the initial conditions will be given for $\tau = 0$:

$$\widetilde{C}^l(\tau) = \int_0^\tau \bar{\psi}_i^l q^i(\tau') e^{-\lambda^l \tau'} \mathrm{d}\tau' \tag{3.22}$$

By inserting this into equation 3.15, the particular solution to the inhomogeneous differential equation 3.1 is obtained,

$$x_{(p)}^i(\tau) = \psi_l^i e^{\lambda^l \tau} \int_0^\tau \bar{\psi}_k^l q^k(\tau') e^{-\lambda^l \tau'} \mathrm{d}\tau' \ . \tag{3.23}$$

The general solution to this equation is the superposition of the homogeneous solution with a particular solution, which in this case can be written as

$$x_g^i(\tau) = \psi_l^i e^{\lambda^l \tau} \left(C^l + \int_0^\tau \bar{\psi}_k^l q^k(\tau') e^{-\lambda^l \tau'} \mathrm{d}\tau' \right) \tag{3.24}$$

Next, one can derive an integratable expression for q^k. This can be achieved by inserting equation 3.5 into equation 3.10

$$q^k(\tau) = \left(e^{a\tau}\eta - \theta\right) \alpha^k + \left(e^{a\tau}\eta - \theta\right)^2 \beta^k + \gamma^k. \tag{3.25}$$

It should be noted that a is in fact equal to the negative of one of the eigenvalues λ^l. The expression above can be re-written collecting powers of $e^{a\tau}$. It is also convenient to abbreviate this further as

$$q^k(\tau) = e^{a\tau}(\underbrace{\eta\alpha^k - 2\eta\theta\beta^k}_{D^k}) + e^{2a\tau}(\underbrace{\eta^2\beta^k}_{F^k}) + (\underbrace{\theta^2\beta^k - \theta\alpha^k + \gamma^k}_{E^k}), \tag{3.26}$$

which leads to

$$q^k(\tau) = e^{a\tau} D^k + e^{2a\tau} F^k + E^k. \tag{3.27}$$

Now, one can put this expression into equation 3.24 and thereby obtains

$$x^i(\tau) = \psi_l^i e^{\lambda^l \tau} \left(C^l + \int_0^\tau \left(e^{(a-\lambda^l)\tau'} \bar{\psi}_k^l D^k + e^{(2a-\lambda^l)\tau'} \bar{\psi}_k^l F^k + e^{-\lambda^l \tau'} \bar{\psi}_k^l E^k \right) d\tau' \right). \tag{3.28}$$

The individual integrals over τ' yield

$$\int_0^\tau e^{(a-\lambda^l)\tau'} d\tau' = \frac{1}{a-\lambda^l}(e^{(a-\lambda^l)\tau} - 1) \tag{3.29}$$

$$\int_0^\tau e^{(2a-\lambda^l)\tau'} d\tau' = \frac{1}{2a-\lambda^l}(e^{(2a-\lambda^l)\tau} - 1) \tag{3.30}$$

$$\int_0^\tau e^{-\lambda^l \tau'} d\tau' = -\frac{1}{\lambda^l}(e^{-\lambda^l \tau} - 1) \tag{3.31}$$

Using these results, equation 3.28 can be re-written as

$$x^i(\tau) = \psi_l^i \left(C^l e^{\lambda^l \tau} + \frac{\bar{\psi}_k^l D^k}{a-\lambda^l}(e^{a\tau} - e^{\lambda^l \tau}) + \frac{\bar{\psi}_k^l F^k}{2a-\lambda^l}(e^{2a\tau} - e^{\lambda^l \tau}) - \frac{\bar{\psi}_k^l E^k}{\lambda^l}(1 - e^{\lambda^l \tau}) \right). \tag{3.32}$$

Next, the components of C^l need to be determined for given initial conditions

$$x^i(\tau = 0) = x^i_{(0)}. \tag{3.33}$$

Setting $\tau = 0$ in equation 3.32 yields

$$x^i_{(0)} = \psi_l^i C^l + 0 + 0 - 0 = \psi_l^i C^l. \tag{3.34}$$

By multiplying with the inverse matrix $\bar{\psi}_i^l$, the coefficients are given by

$$C^l = \bar{\psi}_i^l x^i_{(0)}. \tag{3.35}$$

Therefore, formula 3.32 can be rewritten and yields the formal solution of eq. (3.7)

$$\begin{aligned} x^i(\tau) = \psi_l^i \Bigg(& \bar{\psi}_k^l x^k_{(0)} e^{\lambda^l \tau} + \frac{\bar{\psi}_k^l D^k}{a-\lambda^l}(e^{a\tau} - e^{\lambda^l \tau}) + \frac{\bar{\psi}_k^l F^k}{2a-\lambda^l}(e^{2a\tau} - e^{\lambda^l \tau}) \\ & - \frac{\bar{\psi}_k^l E^k}{\lambda^l}(1 - e^{\lambda^l \tau}) \Bigg), \end{aligned} \tag{3.36}$$

with D^k, E^k and F^k being constant within the linearized field.

3.1.4 Axisymmetric case

In the previous section the analytical solution for the particle coordinates as functions of time was derived. In the derivation the eigenvalues λ^l and eigenvectors ψ_l^i of the (3x3) matrix $\tilde{a}_l^i$ played an essential role in calculating the coordinates $x^i(\tau)$. From the form of the elements of $\tilde{a}_l^i$ one can deduce that for a general non-axisymmetric system one eigenvalue will always be equal to zero. This is however not problematic as long as the eigenvalue corresponds to a non-trivial eigenvector, which is the case. If one is interested in calculating the analytical solution for the coordinates in an axisymmetric (symmetric in φ) configuration, the additional symmetry will reduce the problem to a two-dimensional system and furthermore not allow the use of the same derivation shown in the previous section. This occurs since the matrix a_l^i then has two zero-valued eigenvalues which no longer have two linearly independent eigenvectors. In the derivation above the inverse of the matrix containing the eigenvectors ψ_l^i was needed but since this matrix becomes singular when the eigenvectors are no longer linearly independent, a new approach is necessary. In the upcoming sub-sections an analog solution for the axisymmetric case will be derived.

3.1.5 Axisymmetric homogeneous solution

This section presents the derivation of the analytical solution to the homogeneous part of equation 3.1 for the toroidally axisymmetric case. Here, all derivatives with respect to φ are 0 and it is furthermore assumed that no electric field is present. The matrix a_l^i can then be written in the following notation omitting all zero-valued elements and introducing the abbreviations $d_{ij} = \frac{\partial h_i}{\partial x^j}$ and $u_i = \frac{\partial U}{\partial x^i}$, where h_k denotes the direction of the magnetic field with $h_k = B_k/B$ and the factor cm/e arises from substituting for the cyclotron frequency $\omega_c = eB/cm$.

$$a_l^i = \frac{cm}{e} \begin{pmatrix} -d_{21}u_3 & 0 & -d_{23}u_3 \\ d_{31}u_1 + d_{11}u_3 & 0 & -d_{33}u_1 + d_{13}u_3 \\ d_{21}u_1 & 0 & d_{23}u_1 \end{pmatrix}$$

Due to the linearization of the electromagnetic fields, the values for d_{ij} and u_i remain constant within a given tetrahedron. From the form of matrix a_l^i can be deduced that the values for $\frac{\mathrm{d}x_2}{\mathrm{d}\tau}$ do not depend on x_2, the system of differential equations therefore reduces to a two-dimensional system, where the x_2-component can be calculated independently from the solutions for x_1 and x_2. The two-dimensional system of equations for the x_1 and x_3 component can be formulated using a reduced matrix

$$\tilde{a}_l^{*,i} = \frac{cm}{e} \begin{pmatrix} -d_{21}u_3 & -d_{23}u_3 \\ d_{21}u_1 & d_{23}u_1 \end{pmatrix}$$

such that

$$\begin{pmatrix} \dot{x_1}(\tau) \\ \dot{x_3}(\tau) \end{pmatrix} = \frac{cm}{e} \begin{pmatrix} -d_{21}u_3 & -d_{23}u_3 \\ d_{21}u_1 & d_{23}u_1 \end{pmatrix} \cdot \begin{pmatrix} x_1(\tau) \\ x_3(\tau) \end{pmatrix} + \begin{pmatrix} q_1(\tau) \\ q_3(\tau) \end{pmatrix}. \tag{3.37}$$

In this reduced system of equations, the Ansatz

$$\vec{x} = e^{\lambda\tau}\vec{\psi} \tag{3.38}$$

will be used in order to construct the homogeneous solution, where λ denotes the eigenvalues of $\tilde{a}_l^{*,i}$ and $\vec{\psi}$ the corresponding eigenvector.
Using this, the homogeneous part of equation 3.1 yields the eigenvalue equation

$$\lambda e^{\lambda\tau}\vec{\psi} = \hat{a}^* e^{\lambda\tau}\vec{\psi} \tag{3.39}$$

The general solution to the homogeneous differential equation can then be written as

$$x^i_{(h)}(\tau) = C_1 e^{\lambda_1\tau}\psi_1^i + C_2 e^{\lambda_2\tau}\psi_2^i. \tag{3.40}$$

Explicit calculation of the eigenvalues and corresponding eigenvectors of $a_l^{*,i}$ yields

$$\lambda_1 = 0 \tag{3.41}$$
$$\lambda_2 = \lambda = \frac{cm}{e}\left(d_{23}u_1 - d_{21}u_3\right) \tag{3.42}$$
$$\psi_1^i = \begin{pmatrix} \frac{-d_{23}}{d_{21}} \\ 1 \end{pmatrix} \tag{3.43}$$
$$\psi_2^i = \begin{pmatrix} \frac{-u_3}{u_1} \\ 1 \end{pmatrix} \tag{3.44}$$

and the general solution to the homogeneous system therefore becomes

$$x^i_{(h)}(\tau) = C_1 \begin{pmatrix} \frac{-d_{23}}{d_{21}} \\ 1 \end{pmatrix} + C_2 e^{\frac{cm}{e}(d_{23}u_1 - d_{21}u_3)\tau} \begin{pmatrix} \frac{-u_3}{u_1} \\ 1 \end{pmatrix} \tag{3.45}$$

with the C_i denoting arbitrary constants given by initial conditions of the problem. The eigenvectors furthermore need not be normalized since any normalization constant

could be pulled into the C_i.

3.1.6 Axisymmetric particular solution: variation of constants

Now that the solution to the homogeneous part has been found, one is interested in a particular solution to the inhomogeneous differential equation 3.1 to construct the general solution. In order to do this, one can once more apply the method of variation of constants where coefficients C_i of the homogeneous solution are treated as functions of τ:

$$x^i_{(p)}(\tau) = C_1(\tau)\psi^i_1 + C_2(\tau)e^{\lambda\tau}\psi^i_2 \tag{3.46}$$

Calculating the derivative of this equation yields:

$$\frac{\mathrm{d}x^i_{(p)}}{\mathrm{d}\tau} = {C_1}'(\tau)\psi^i_1 + {C_2}'(\tau)e^{\lambda\tau}\psi^i_2 + C_2(\tau)\lambda e^{\lambda\tau}\psi^i_2 \tag{3.47}$$

Inserting this expression into equation 3.37 results in

$$q^i(\tau) = {C_1}'(\tau)\psi^i_1 + {C_2}'(\tau)e^{\lambda\tau}\psi^i_2. \tag{3.48}$$

It is convenient to write this expression as a matrix vector product, explicitly given as

$$\begin{pmatrix} q^1(\tau) \\ q^3(\tau) \end{pmatrix} = \underbrace{\left[\psi^i_1, e^{\lambda\tau}\psi^i_2\right]}_{M} \cdot \begin{pmatrix} {C_1}'(\tau) \\ {C_2}'(\tau) \end{pmatrix} = \begin{pmatrix} \frac{-d_{23}}{d_{21}} & \frac{-u_3}{u_1}e^{\lambda\tau} \\ 1 & e^{\lambda\tau} \end{pmatrix} \cdot \begin{pmatrix} {C_1}'(\tau) \\ {C_2}'(\tau) \end{pmatrix}. \tag{3.49}$$

By inverting M and multiplying by this inverse matrix from the left one obtains the explicit expression

$$\begin{pmatrix} {C_1}'(\tau) \\ {C_2}'(\tau) \end{pmatrix} = M^{-1} \cdot \begin{pmatrix} q^1(\tau) \\ q^3(\tau) \end{pmatrix} = \frac{cm}{e\lambda} \begin{pmatrix} -d_{21}u_1 & -d_{21}u_3 \\ d_{21}u1e^{-\lambda\tau} & d_{23}u_1e^{-\lambda\tau} \end{pmatrix} \cdot \begin{pmatrix} q^1(\tau) \\ q^3(\tau) \end{pmatrix} \tag{3.50}$$

for ${C_i}'(\tau)$.
One can now formally replace τ by τ' and integrate this expression from 0 to τ since the initial conditions will be given for $\tau = 0$:

$$\begin{pmatrix} C_1(\tau) \\ C_2(\tau) \end{pmatrix} = \frac{cm}{e\lambda} \int_0^\tau d\tau' \begin{pmatrix} -d_{21}u_1 & -d_{21}u_3 \\ d_{21}u_1e^{-\lambda\tau'} & d_{23}u_1e^{-\lambda\tau'} \end{pmatrix} \cdot \begin{pmatrix} q^1(\tau') \\ q^3(\tau') \end{pmatrix} \tag{3.51}$$

Next, one can continue by using the explicit expression for q^i, given in equation 3.27.

Using the fact that $a = -\lambda$ simplifies this result to

$$q^k(\tau) = e^{-\lambda\tau} D^k + e^{-2\lambda\tau} F^k + E^k \tag{3.52}$$

where each vectorial quantity consists only of the x_1 and x_3 components.
The results for $C_i(\tau)$ can then be written as

$$C_1(\tau) = \frac{cm}{e} d_{21} u_k \left[\frac{e^{-2\lambda\tau} - 1}{2\lambda^2} F^k + \frac{e^{-\lambda\tau} - 1}{\lambda^2} D^k - \frac{\tau}{\lambda} E^k \right] \tag{3.53}$$

$$C_2(\tau) = \frac{cm}{e} u_1 d_{2k} \left[\frac{1 - e^{-3\lambda\tau}}{3\lambda^2} F^k + \frac{1 - e^{-2\lambda\tau}}{2\lambda^2} D^k + \frac{1 - e^{-\lambda\tau}}{\lambda^2} E^k \right] \tag{3.54}$$

with u_k and d_{2k} being

$$u_k = \begin{pmatrix} u_1 \\ u_3 \end{pmatrix}, \qquad d_{2k} = \begin{pmatrix} d_{21} \\ d_{23} \end{pmatrix}. \tag{3.55}$$

These expressions for $C_i(\tau)$ can now be put into equation 3.46 to calculate the particular solution. Superposition of particular and homogeneous solution constructs the general solution of the ODE set.

$$x^i_{(g)}(\tau) = x^i_{(h)} + x^i_{(p)} = (\tilde{C}_1 + C_1(\tau))\psi^i_1 + (\tilde{C}_2 + C_2(\tau)) e^{\lambda\tau} \psi^i_2 \tag{3.56}$$

Since the $C_i(\tau)$ were integrated from $\tau' = 0$ to $\tau\prime = \tau$ they vanish for $\tau = 0$, whereby initial conditions require that

$$x^i_{(g)}(\tau = 0) = x^i_{(0)} = \tilde{C}_1 \psi^i_1 + \tilde{C}_2 \psi^i_2 \,. \tag{3.57}$$

Explicitly, this gives two equations for $\tilde{C}_i$, namely

$$x^1_{(0)} = -\frac{d_{23}}{d_{21}} \tilde{C}_1 - \frac{u_3}{u_1} \tilde{C}_2, \tag{3.58}$$

$$x^3_{(0)} = \tilde{C}_1 + \tilde{C}_2. \tag{3.59}$$

Using $\frac{cm}{e}\lambda = (d_{23} u_1 - d_{21} u_3)$, the $\tilde{C}_i$ are therefore

$$\tilde{C}_1 = -\frac{cm}{e} d_{21} \frac{u_i x^i_{(0)}}{\lambda} \,, \tag{3.60}$$

$$\tilde{C}_2 = \frac{cm}{e} u_1 \frac{d_{2i} x^i_{(0)}}{\lambda} \,. \tag{3.61}$$

Inserting this into equation 3.56 yields the analytical result for the coordinates x^1, x^3:

$$\begin{aligned} x^1(\tau) &= \frac{cm}{e\lambda}\Bigg[x^k_{(0)}\left(d_{23}u_k - u_3 d_{2k}e^{\lambda\tau}\right) \\ &- d_{23}u_k\left(\frac{e^{-2\lambda\tau}-1}{2\lambda}F^k + \frac{e^{-\lambda\tau}-1}{\lambda}D^k - \tau E^k\right) \\ &- u_3 d_{2k}\left(\frac{e^{\lambda\tau}-e^{-2\lambda\tau}}{3\lambda}F^k + \frac{e^{\lambda\tau}-e^{-\lambda\tau}}{2\lambda}D^k + \frac{e^{\lambda\tau}-1}{\lambda}E^k\right)\Bigg] \end{aligned} \tag{3.62}$$

$$\begin{aligned} x^3(\tau) &= \frac{cm}{e\lambda}\Bigg[x^k_{(0)}\left(-d_{21}u_k + u_1 d_{2k}e^{\lambda\tau}\right) \\ &+ d_{21}u_k\left(\frac{e^{-2\lambda\tau}-1}{2\lambda}F^k + \frac{e^{-\lambda\tau}-1}{\lambda}D^k - \tau E^k\right) \\ &+ u_1 d_{2k}\left(\frac{e^{\lambda\tau}-e^{-2\lambda\tau}}{3\lambda}F^k + \frac{e^{\lambda\tau}-e^{-\lambda\tau}}{2\lambda}D^k + \frac{e^{\lambda\tau}-1}{\lambda}E^k\right)\Bigg] \end{aligned} \tag{3.63}$$

Now that x^1 and x^3 have been found, x^2 can be calculated via the second component of the differential equation set 3.7, yielding

$$\dot{x}^2(\tau) = a_1^{*,2}x^1(\tau) + a_3^{*,2}x^3(\tau) + q^2(\tau)\,, \tag{3.64}$$

which explicitly evaluated reads

$$\begin{aligned} \dot{x}^2(\tau) = &\frac{cm}{e}\left(d_{31}u_1 + d_{11}u_3\right)x^1(\tau) + \frac{cm}{e}\left(-d_{33}u1 + d_{13}u_3\right)x^3(\tau) \\ &+ e^{-\lambda\tau}D^2 + e^{-2\lambda\tau}F^2 + E^2\,. \end{aligned} \tag{3.65}$$

Formally replacing τ by τ' and subsequent integration over τ' from 0 to τ yields the result for $x^2(\tau)$. Since this equation depends only on $\dot{x}^2(\tau)$ and not on $x^2(\tau)$, an arbitrary constant C can be added to $x^2(\tau)$ which will be determined by initial conditions. Since the integral over τ' starts at 0, this constant will be given by $C = x^2_{(0)}$.

For clarity, one can define that

$$\begin{aligned}
\mathrm{X}^1(\tau) &= \int_0^\tau x^1(\tau')d\tau' && (3.66)\\
&= \frac{cm}{e\lambda}\Bigg[x^k_{(0)}\left(d_{23}u_k\tau - u_3 d_{2k}\frac{e^{\lambda\tau}-1}{\lambda}\right)\\
&- d_{23}u_k\left(\frac{e^{-2\lambda\tau}+2\lambda\tau-1}{4\lambda^2}F^k - \frac{e^{-\lambda\tau}+\lambda\tau-1}{\lambda^2}D^k - \frac{\tau^2}{2}E^k\right)\\
&- u_3 d_{2k}\left(\frac{2e^{\lambda\tau}+e^{-2\lambda\tau}-3}{6\lambda^2}F^k + \frac{e^{\lambda\tau}+e^{-\lambda\tau}-2}{2\lambda^2}D^k + \frac{e^{\lambda\tau}-\lambda\tau-1}{\lambda^2}E^k\right)\Bigg],\\
\mathrm{X}^3(\tau) &= \int_0^\tau x^3(\tau')d\tau' && (3.67)\\
&= \frac{cm}{e\lambda}\Bigg[x^k_{(0)}\left(-d_{21}u_k\tau + u_1 d_{2k}\frac{e^{\lambda\tau}-1}{\lambda}\right)\\
&+ d_{21}u_k\left(\frac{e^{-2\lambda\tau}+2\lambda\tau-1}{4\lambda^2}F^k - \frac{e^{-\lambda\tau}+\lambda\tau-1}{\lambda^2}D^k - \frac{\tau^2}{2}E^k\right)\\
&+ u_1 d_{2k}\left(\frac{2e^{\lambda\tau}+e^{-2\lambda\tau}-3}{6\lambda^2}F^k + \frac{e^{\lambda\tau}+e^{-\lambda\tau}-2}{2\lambda^2}D^k + \frac{e^{\lambda\tau}-\lambda\tau-1}{\lambda^2}E^k\right)\Bigg],\\
\mathrm{Q}^2(\tau) &= \int_0^\tau q^2(\tau')d\tau' && (3.68)\\
&= \frac{1-e^{-\lambda\tau}}{\lambda}D^2 + \frac{1-e^{-\lambda\tau}}{2\lambda}F^2 + \tau E^2\,.
\end{aligned}$$

The solution for $x^2(\tau)$ can then be compactly written as

$$x^2(\tau) = \frac{cm}{e}\left(d_{31}u_1 + d_{11}u_3\right)\mathrm{X}^1(\tau) + \frac{cm}{e}\left(-d_{33}u_1 + d_{13}u_3\right)\mathrm{X}^3(\tau) + \mathrm{Q}^2(\tau) + \underbrace{\mathrm{C}}_{=x^2_{(0)}}\,. \tag{3.69}$$

3.2 Integration of equations of motion with RK4

3.2.1 Derivation of the RK4-Error for *Gorilla*

In the previous section, the analytical derivation of $z^i(\tau)$ has been found for both the general and the axisymmetric case. While such an explicit expression yields valuable information about the properties of the solution (compare **??**), for numerics it is not necessarily optimal to evaluate this analytic expression directly. For one, the form of the solution can introduce strong numerical cancellations which drastically limits the accuracy of the evaluated solution (especially terms of the form $(e^{a\tau}-1)$ can introduce massive errors for small steps in τ), for another, it can be very inefficient to

separately evaluate all quantities occuring in the analytical expression. The alternative approach is to ignore the analytical information about components of a_l^i and b^i as they remain invariant for a given tetrahedron and initial conditions anyway. Finding an expression in the form of a power series as function of the full components holds the advantage that the solution up to a specific order can be computed more efficiently, as well as that the accuracy of the solution can be tuned by choosing up to which order one computes the series. However, other parameters like the coarseness of the grid might be more optimal to change in this regard. Furthermore, since the exact solution inserted into the Hesse normal form of a face of a given tetrahedron represents a transcendental equation for τ, a low order polynomial representation brings the advantage that approximate solutions can in fact be algebraically computed by finding the smallest positive root of this polynomial. This briefly described problem is the fundamental problem which will be solved in the later described `pusher_tetra_orbit` subroutine, therefore it is of high interest to find alternative, hopefully more efficient, ways to solve for τ.

As previously discussed, within each tetrahedron the set of four linear ODE for z^i is given by

$$\frac{\mathrm{d}z^i}{\mathrm{d}\tau} = a_l^i z^l + b^i, \tag{3.70}$$

where $i = 1, .., 4$ and the components of a_l^i and b^i are constant. It is convenient to use the vector notation of writing non-scalar quantities in bold and labeling matrices with a hat, i.e. $\mathbf{z}$, $\hat{\mathbf{a}}$ and $\mathbf{b}$, also, " $\cdot$ " is used to denote matrix multiplications.

$$\frac{\mathrm{d}\mathbf{z}}{\mathrm{d}\tau} = \hat{\mathbf{a}} \cdot \mathbf{z} + \mathbf{b}. \tag{3.71}$$

Firstly, the initial conditions for $\mathbf{z}(\tau)$ are denoted $\mathbf{z}(0) = \mathbf{z}_0$. Next, one is interested in the solution of $\mathbf{z}(\tau)$ in the form of a power series

$$\mathbf{z} = \mathbf{z}_0 + \tau \mathbf{z}_1 + \tau^2 \mathbf{z}_2 + ... = \sum_{k=0}^{\infty} \tau^k \mathbf{z}_k, \tag{3.72}$$

where the $\mathbf{z}_k$ are held constant. Substituting this expression into (3.71) one obtains

$$\sum_{k=0}^{\infty} (k+1)\tau^k \mathbf{z}_{k+1} = \mathbf{b} + \sum_{k=0}^{\infty} \tau^k \hat{\mathbf{a}} \cdot \mathbf{z}_k. \tag{3.73}$$

Independently equating the different powers of τ yields an infinite series with elements

$$\begin{aligned} \mathbf{z}_1 &= \mathbf{b} + \hat{\mathbf{a}} \cdot \mathbf{z}_0, \\ 2\mathbf{z}_2 &= \hat{\mathbf{a}} \cdot \mathbf{z}_1, \\ 3\mathbf{z}_3 &= \hat{\mathbf{a}} \cdot \mathbf{z}_2, \\ &\dots, \\ k\mathbf{z}_k &= \hat{\mathbf{a}} \cdot \mathbf{z}_{k-1}, \\ &\dots \end{aligned} \tag{3.74}$$

which is solved by

$$\mathbf{z}_k = \frac{1}{k!}\hat{\mathbf{a}}^{k-1} \cdot \mathbf{z}_1.$$

Thus, series (3.72) is explicitly given by

$$\mathbf{z} = \mathbf{z}_0 + \sum_{k=1}^{\infty} \frac{\tau^k}{k!} \left(\hat{\mathbf{a}^{k-1}} \cdot \mathbf{b} + \hat{\mathbf{a}}^k \cdot \mathbf{z}_0 \right). \tag{3.75}$$

This series converges for all values of $\hat{\mathbf{a}}$ and τ, due to the factorial in the denominator. It is here convenient to introduce a more general notation for sets of equations with solely implicit time dependence

$$\frac{\mathrm{d}\mathbf{z}}{\mathrm{d}\tau} = \mathbf{f}(\mathbf{z}), \tag{3.76}$$

where $\mathbf{f}(\mathbf{z})$ is generally a nonlinear vector function only of $\mathbf{z}$. However, in the present application using the linear set (3.71) this function is

$$\mathbf{f}(\mathbf{z}) = \mathbf{b} + \hat{\mathbf{a}} \cdot \mathbf{z}. \tag{3.77}$$

Denoting now $\mathbf{z}_{RK4}$ the result of a single 4-th order Runge Kutta step from 0 to τ with initial values $\mathbf{z}_0$. As shown in appendix B, this result is explicitly evaluated to

$$\begin{aligned} \mathbf{z}_{RK4} &= \mathbf{z}_0 + \frac{\tau}{6}\mathbf{f}(\mathbf{z}_0) + \frac{\tau}{3}\mathbf{f}\left(\mathbf{z}_0 + \frac{\tau}{2}\mathbf{f}(\mathbf{z}_0)\right) + \frac{\tau}{3}\mathbf{f}\left(\mathbf{z}_0 + \frac{\tau}{2}\mathbf{f}\left(\mathbf{z}_0 + \frac{\tau}{2}\mathbf{f}(\mathbf{z}_0)\right)\right) \\ &+ \frac{\tau}{6}\mathbf{f}\left(\mathbf{z}_0 + \tau\mathbf{f}\left(\mathbf{z}_0 + \frac{\tau}{2}\mathbf{f}\left(\mathbf{z}_0 + \frac{\tau}{2}\mathbf{f}(\mathbf{z}_0)\right)\right)\right) . \end{aligned} \tag{3.78}$$

By now denoting $\mathbf{f}_0 = \mathbf{f}(\mathbf{z}_0)$, for an arbitrary displacement $\Delta\mathbf{z}$, one further obtains in accordance with (3.77)

$$\mathbf{f}(\mathbf{z}_0 + \Delta\mathbf{z}) = \mathbf{f}_0 + \hat{\mathbf{a}} \cdot \Delta\mathbf{z}. \tag{3.79}$$

Applying this formula to (3.78) yields the explicit expression for the approximative

RK4-solution

$$\mathbf{z}_{RK4} = \mathbf{z}_0 + \tau\mathbf{f}_0 + \frac{\tau^2}{2}\hat{\mathbf{a}}\cdot\mathbf{f}_0 + \frac{\tau^3}{6}\hat{\mathbf{a}}\cdot\hat{\mathbf{a}}\cdot\mathbf{f}_0 + \frac{\tau^4}{24}\hat{\mathbf{a}}\cdot\hat{\mathbf{a}}\cdot\hat{\mathbf{a}}\cdot\mathbf{f}_0 = \mathbf{z}_0 + \sum_{k=1}^{4}\frac{\tau^k}{k!}\hat{\mathbf{a}}^{k-1}\cdot\mathbf{f}_0\ . \quad (3.80)$$

By using the same notation $\mathbf{f}_0 = \mathbf{b} + \hat{\mathbf{a}}\cdot\mathbf{z}_0$ in (3.75), the exact solution for $\mathbf{z}$ reads

$$\mathbf{z} = \mathbf{z}_0 + \sum_{k=1}^{\infty}\frac{\tau^k}{k!}\hat{\mathbf{a}}^{k-1}\cdot\mathbf{f}_0, \quad (3.81)$$

with the *RK4*-error being explicitly given by

$$\Delta\mathbf{z}_{RK4} = \sum_{k=5}^{\infty}\frac{\tau^k}{k!}\hat{\mathbf{a}}^{k-1}\cdot\mathbf{f}_0. \quad (3.82)$$

Thus, the 4-th order Runge Kutta solution (3.80) differs from the exact solution (3.81) by terms of the order $\tau^5\hat{\mathbf{a}}^4$ and higher. Moreover, due to the parabolic time dependence of the motion along the field line in the linearized system, the truncated solution $\mathbf{z}_{RK4}$ resolves this motion exactly, thus, discrepancies arise only due to FLR (finite larmor radius) effects and, hence, depend on particle species and initial conditions. For instance, electrons have a larmor radius which is roughly 40 times smaller than the larmor radius of ions, therefore, the Runge-Kutta error for ions is therefore expected to be much larger for ions than for electrons.

3.2.2 Taylor expansion of the analytical solution

The analytical solution for the non-axisymmetric case can be written as

$$x^i(\tau) = \psi^i_l\bar{\psi}^l_k\left(x^k_{(0)}e^{\lambda^l\tau} + \frac{D^k}{a-\lambda^l}(e^{a\tau} - e^{\lambda^l\tau}) + \frac{F^k}{2a-\lambda^l}(e^{2a\tau} - e^{\lambda^l\tau}) + \frac{E^k}{\lambda^l}(e^{\lambda^l\tau} - 1)\right), \quad (3.83)$$

whereas the fourth order Runge-Kutta method (*RK4*) computes the solution exactly up to the fourth order of the corresponding Taylor expansion [6]:

$$
\begin{aligned}
x^i_{\mathrm{RK4}} =& \psi^i_l \bar{\psi}^l_k \Bigg[x^k_{(0)} + \tau \left(\lambda^l x^k_{(0)} + D^k + F^k + E^k \right) \\
&+ \frac{\tau^2}{2} \left((\lambda^l)^2 x^k_{(0)} + (a + \lambda^l) D^k + (2a + \lambda^l) F^k + \lambda^l E^k \right) \\
&+ \frac{\tau^3}{6} \left((\lambda^l)^3 x^k_{(0)} + (a^2 + \lambda^l (a + \lambda^l)) D^k + (4a^2 + \lambda^l (2a + \lambda^l)) F^k + (\lambda^l)^2 E^k \right) \\
&+ \frac{\tau^4}{24} \left((\lambda^l)^4 x^k_{(0)} + (a + \lambda^l)(a^2 + (\lambda^l)^2) D^k + (2a + \lambda^l)(4a^2 + (\lambda^l)^2) F^k + (\lambda^l)^3 E^k \right) \Bigg]
\end{aligned}
\tag{3.84}
$$

By substituting D^k, E^k and F^k with their original values in terms of α^k, β^k and γ^k, respectively, one can reduce the numerical cancelation, which unfortunately is quite strong in this representation. Equation 3.84 can then be rewritten as

$$
\begin{aligned}
x^i_{\mathrm{RK4}} =& \psi^i_l \bar{\psi}^l_k \Bigg[x^k_{(0)} + \tau \left(\lambda^l x^k_{(0)} + q^k_{(0)} \right) \\
&+ \frac{\tau^2}{2} \left((\lambda^l)^2 x^k_{(0)} + \lambda^l q^k_{(0)} + (v_{\parallel,0} + \frac{b}{a})(a\alpha^k + 2v_{\parallel,0}\beta^k) \right) \\
&+ \frac{\tau^3}{6} \Bigg((\lambda^l)^3 x^k_{(0)} + (\lambda^l)^2 q^k_{(0)} + (a v_{\parallel,0} + b)(a + \lambda^l)\alpha^k \\
&+ (-2b(a + \lambda^l)(v_{\parallel,0} + \frac{b}{a}) + 2a(2a + \lambda^l)(v_{\parallel,0} + \frac{b}{a})^2)\beta^k \Bigg) \\
&+ \frac{\tau^4}{24} \Bigg((\lambda^l)^4 x^k_{(0)} + (\lambda^l)^3 q^k_{(0)} + a(a^2 + \lambda^l a + \lambda^2)(v_{\parallel,0} + \frac{b}{a})\alpha^k \\
&+ (-2\frac{b}{a}(v_{\parallel,0} + \frac{b}{a}) + a(8a^2 + 4\lambda^l a + 2(\lambda^l)^2)(v_{\parallel,0} + \frac{b}{a})^2)\beta^k \Bigg) \Bigg]
\end{aligned}
\tag{3.85}
$$

To calculate the absolute error of this method, further terms of the Taylor expansion of x^i_{RK4} can be computed, only the leading order is usually of interest, though.

$$
\begin{aligned}
\Delta x^i_{(5)} = \psi^i_l \sum_{l=1}^{3} \sum_{j=5}^{\infty} \Bigg(& \frac{(\lambda^l \tau)^j}{j!} \left(\bar{\psi}^l_i x^i_{(0)} - \frac{\bar{\psi}^l_k D^k}{a - \lambda^l} - \frac{\bar{\psi}^l_k F^k}{2a - \lambda^l} \right) \\
& + \frac{(a\tau)^j}{j!} \frac{\bar{\psi}^l_k D^k}{a - \lambda^l} + \frac{(2a\tau)^j}{j!} \frac{\bar{\psi}^l_k F^k}{2a - \lambda^l} - \frac{(-\lambda^l)^{j-1} \tau^j}{j!} \bar{\psi}^l_k E^k \Bigg)
\end{aligned}
\tag{3.86}
$$

An expression for the Runge-Kutta error of position as function of initial conditions, eigenvalues and of eigenvectors of $\hat{a}$ has thereby been found.

3.3 Measurement of the RK4 error

In this section numerical experiments are performed to measure the *RK4* errors during orbit integration in *Gorilla*. The idea is here to evaluate orbits for given initial conditions (given by the 4D-vector $z^i(0) = [x_0^i, v_{\parallel,0}]$ and the perpendicular adiabatic invariant denoted `perpinv` $= -\frac{1}{2}v_{\perp,0}^2/B(x_0^i)$, upon entering a tetrahedron), where for each tetrahedral transition of the particle, the associated value of the orbit parameter τ is computed. Here, two independent integration steps of step length τ are then computed for these initial conditions, once with a single *RK4* step and once with the adaptive *RK4/5* scheme for a relative accuracy of 10^{-17} (a short introduction to Runge-Kutta integration can be found in appendix B). Subsequently, the obtained results for the first component of $z^i(\tau)$ are subtracted from one another and the absolute value of the difference is saved, constituting the *measured error*. Next, the leading order term ($\mathcal{O}(\tau^5)$) of equation (3.82) is evaluated independently, allowing for an analytic estimation of the error associated with the *RK4* method, which is therefore denoted the *analytic error*. The idea is here that for sufficiently smooth functions and for negligible numerical inaccuracies, the leading order contribution should in fact give a very good estimation for the measured error, thus, plotting the *analytic error* over the *measured error* is expected to result in a directly proportional behavior with slope 1. As previously discussed, contributions to the *RK4*-error in the linearized electromagnetic field arise strictly from finite larmor radius effects, hence, the accuracy depends both on particle species and on initial conditions z^i and `perpinv`. Since particles with higher mass also have larger larmor radii for a given kinetic energy, the computations of errors are performed for α-particles. For initial conditions, a starting position at $(s, \vartheta, \varphi) = (0.8, 0, 0)$ is defined in symmetry flux coordinates, furthermore, a kinetic energy of 3 keV and a pitch parameter of 0.8 are chosen. Since calculations are performed in both cylindrical coordinates and symmetry flux coordinates, starting positions are transformed if necessary by using the subroutine `magdata_in_symfluxcoord_ext` that was previously introduced in chapter chapter 2. Because the average tetrahedral flight time of the particle scales proportionally to the size of the tetrahedron, the grid size was furthermore chosen to be $(N_R, N_\varphi, N_Z) = (N_s, N_\vartheta, N_\varphi) = (5, 5, 5)$ and for a second calculation to be $(12, 12, 12)$, both configurations resembling a very coarse grid on which the *RK4*-error should be much larger than for a typical grid configuration (e.g. $(100, 100, 100)$) used for orbit computation. The obtained results of the errors for the first 1000 tetrahedral pushings using these starting conditions are presented in figure 3.1 to 3.4.
The main information that one can take from these figures seems to be, that the *analytical error* does in fact coincide exceptionally well with the *measured error*

for larger errors, while for smaller errors the deviation can reach several orders of magnitude which seems to be quite large. To get a qualitative measure of the numerical errors, that occur in the computations, the *analytical error* was computed in two ways. Once, by computing the series expansion up to fourth and fifth order and taking the difference between the two solutions and once by directly evaluating the fifth order contribution. These two results are analytically completely equivalent, however, as one can see in the figures, the two variations behave quite different for small errors. This leads to the conclusion, that the observable discrepancies for small errors might actually have a non-negligible numerical error contribution. The results from these regions are, therefore, not to be misinterpreted to poorly resemble the validity of eq. (3.82), but rather, that the scale of the differences is of the order of the numerical accuracy and, thus, no further information can be gathered from this regime. Another observation, that can be deduced from the figures is that the grid size does in fact have an important influence on the magnitude of the errors, however, given that for alpha particles the errors are so small even for an extremely coarse grid, one can safely assume that with more realistic grid configurations (i.e. (100,100,100)) the Runge-Kutta error does not have a noteworthy influence on the results of guiding center orbits, obtained by the *Gorilla* code.

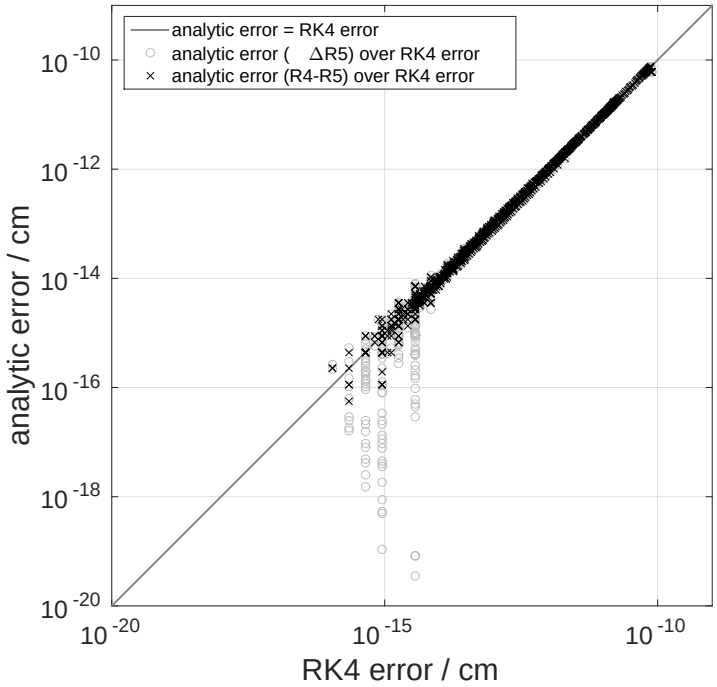


Figure 3.1: Double-logarithmic plot of two versions for the *analytic error* (difference of fourth and fifth order solution (x), direct computation of fifth order contribution (o)) are hereby plotted as function of the *measured error* for a grid size of $(N_R, N_\varphi, N_Z) = (5, 5, 5)$, calculations were performed using cylindrical coordinates

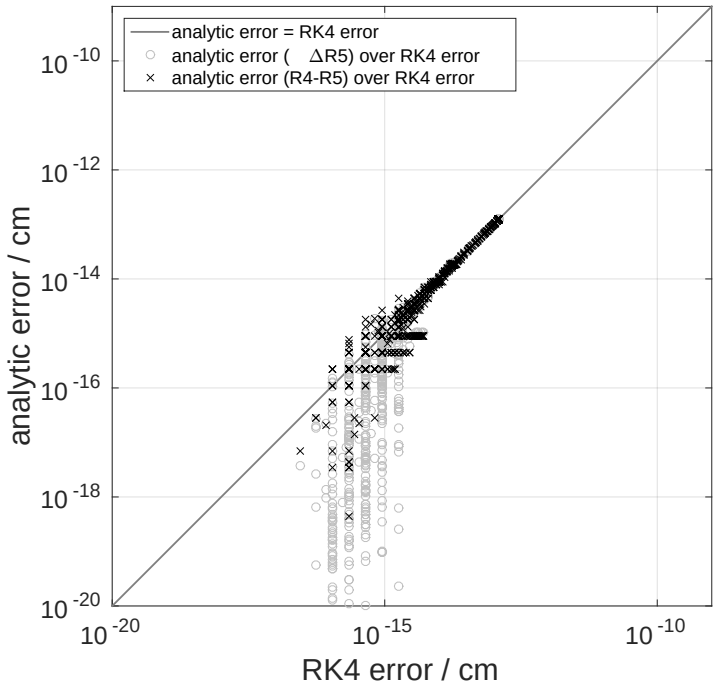


Figure 3.2: Double-logarithmic plot of two versions for the *analytic error* (difference of fourth and fifth order solution (x), direct computation of fifth order contribution (o)) are hereby plotted as function of the *measured error* for a grid size of $(N_R, N_\varphi, N_Z) = (12, 12, 12)$, calculations were performed using cylindrical coordinates

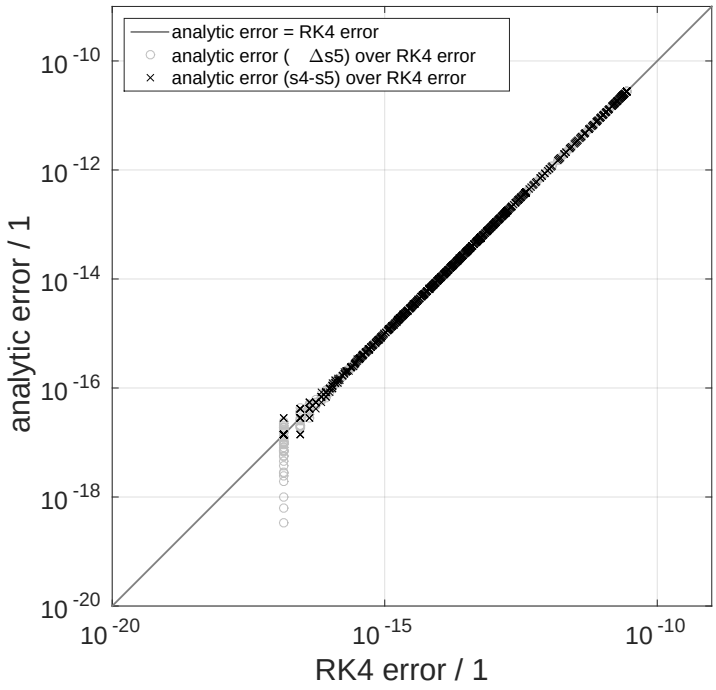


Figure 3.3: Double-logarithmic plot of two versions for the *analytic error* (difference of fourth and fifth order solution (x), direct computation of fifth order contribution (o)) are hereby plotted as function of the *measured error* for a grid size of $(N_s, N_\vartheta, N_\varphi) = (5, 5, 5)$, calculations were performed using symmetry flux coordinates

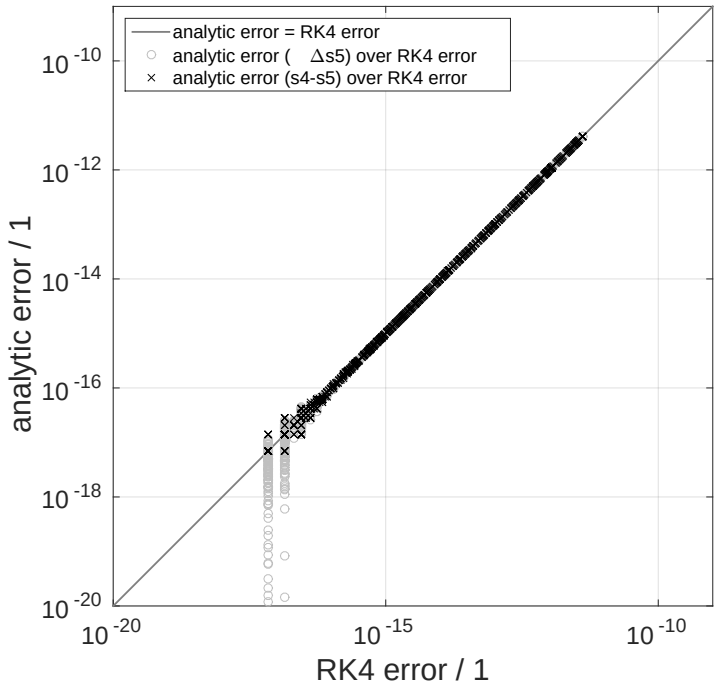


Figure 3.4: Double-logarithmic plot of two versions for the *analytic error* (difference of fourth and fifth order solution (x), direct computation of fifth order contribution (o)) are hereby plotted as function of the *measured error* for a grid size of $(N_s, N_\vartheta, N_\varphi) = (12, 12, 12)$, calculations were performed using symmetry flux coordinates

Chapter 4

Particle orbit pusher algorithms

This chapter is dedicated to the implemented algorithms for finding the first intersection of particle orbits with the tetrahedral cell boundaries in the grids that were previously introduced. Here, a particle can either start at an arbitrary position inside a given tetrahedron or directly at a face of a tetrahedron through which it enters. These routines efficiently compute the next exiting position of the particle through the tetrahedron and the associated flight time of the trajectory. Since this procedure can be thought of as a pushing of the particle orbit through the tetrahedron, the implemented routines are denoted *pusher*-routines. On a sidenote, the fact that both position and time are obtained directly by the approaches used in the pusher routines, a box counting scheme can easily be implemented for future applications, allowing for a very efficient approximation of particle distribution functions, which in turn are a necessary part for possible future computations of kinetic plasma equilibria. The focus of the pusher routines lies, however, not only on the computation of the trajectory and the calculation of the next intersection but rather on finding a numerically inexpensive scheme that allows to save computational cost while reliably yielding accurate results for the exiting position. In the diploma thesis of M. Eder [4], a prior version of the presented pusher routine was discussed in great detail, this routine was named `pusher_tetra_orb`. Due to new insights and structural limiations of the previous code, this code was refactored and extended in cooperation with M. Eder. The resulting code was named `pusher_tetra_orbit`, an overview of the code is given below, however, due to large similarities with the previous approach discussed in [4], the new route will be presented in less detail. Apart from this routine, a second routine named `pusher_tetra_analytic` was implemented based on the previously derived polynomial expansion of the particle orbit. While the results are in theory equivalent for both pushing routines, the approaches are completely independent and thus may vary in both computational efficiency and numerical accuracy, depending on up to which order the analytical expansion of the orbit is computed. Furthermore, for starting a particle at a given position without knowing to which tetrahedron it belongs, an additional routine `find_tetra` was constructed to find the corresponding tetrahedron index to start a calculation.

4.1 Pusher routine `pusher_tetra_orbit`

As discussed, the pusher subroutine `pusher_tetra_orbit` computes the position and time where the particle trajectory first exits a given current tetrahedron. In reality, however, the occurring problem is not only to directly compute the orbit of a single tetrahedron passing, but rather to let a particle start at a position in space and trace its orbit for a defined time. For such a problem one can construct a wrapping routine `orbit_timestep_3dgeoint` which is given the initial conditions of the particle and iteratively calls the pusher subroutine `pusher_tetra_orbit` until the set time is reached. Since generally the set flight time of the particle will lead to an orbit position inside the final tetrahedron, the remaining time of the trajectory must also be given to the pusher routine. The pusher routine then computes the time it takes until the particle exits the current tetrahedron and compares this value to the remaining time of the orbit integration step which was given to the wrapper routine. In case the time it takes to leave the tetrahedron is smaller than the remaining time, the pushing is computed, then the remaining time is reduced by this value and the next pushing through the adjacent tetrahedron is started. In case there is not sufficient time to complete the pushing, the orbit is instead integrated up to the value of the orbit parameter `tau` corresponding to the remaining time, leading to an arbitrary final position inside the tetrahedron. The code structure of the wrapping routine `orbit_timestep_3dgeoint` and the components of the module `pusher_tetra_orbit_mod` is given in figure 4.1.
Due to this wrapping routine, one can directly start the computation of a single particle orbit for a given flight time by calling subroutine `orbit_timestep_3dgeoint` with arguments `(x,vpar,vperp,t_step,boole_initialized,ind_tetr,iface)`. This list of parameters is explained in tab. 4.1.

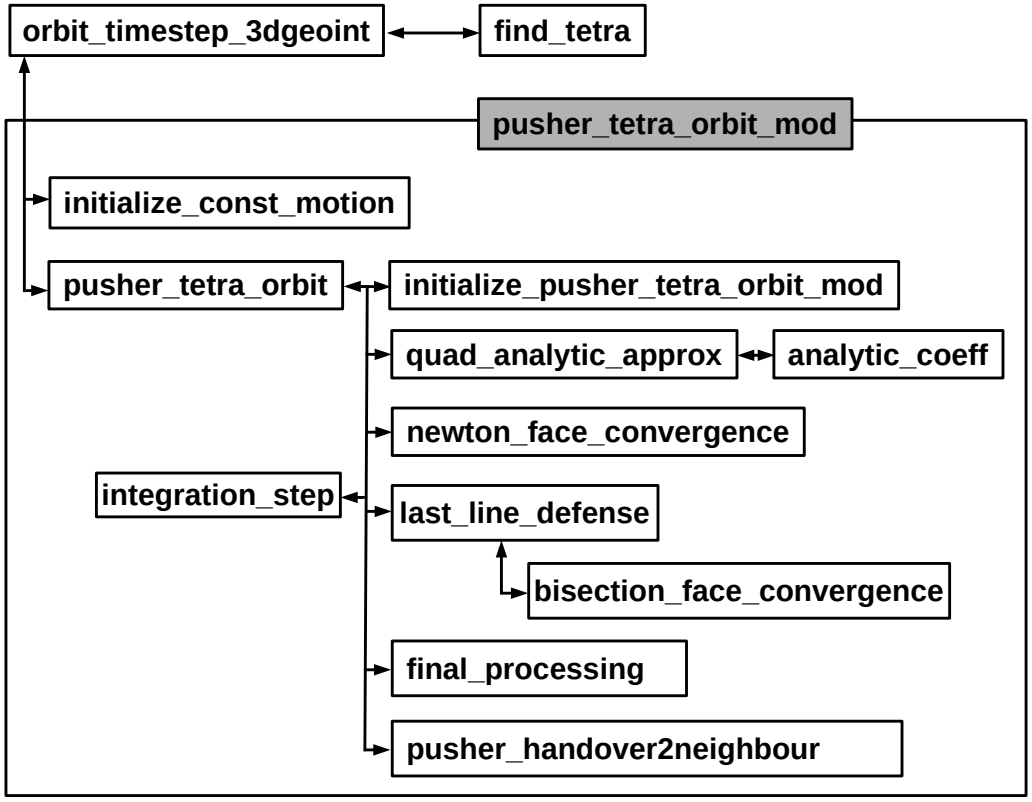


Figure 4.1: Code structure of `pusher_tetra_orbit_mod` and associated subroutine

Table 4.1: Parameter overview for wrapper subroutine `orbit_timestep_3dgeoint`, defines initial conditions and duration of particle motion, orientations of velocities are taken with respect to the orientation of the magnetic field $\vec{B}$.

Data type	Name	Description
`double precision, dimension(3),intent(inout)`	`x`	particle position
`double precision, intent(inout)`	`vpar`	parallel velocity
`double precision, intent(inout)`	`vperp`	perpendicular velocity
`double precision,intent(in)`	`t_step`	defined flight time
`logical,intent(inout)`	`boole_initialized`	sets initialization of constants of motion
`integer,intent(inout)`	`ind_tetr`	tetrahedron index at position `x`
`integer,intent(inout)`	`iface`	index of face if `x` lies on face, 0 otherwise

4.1.1 Initializing constants of motion

In figure 4.1, one can see the code diagram which gives an overview of the different subroutines. When starting a calculation in `orbit_timestep_3dgeoint` for a defined step length, the subroutine that is called first is `initialize_const_motion` which sets the constants of motion for the given initial conditions. These constants of motion are `E_tot`, `perpinv` and `perinv2` which denote the total energy E, the negative perpendicular adiabatic invariant $-J_\perp$ and the squared value thereof, respectively. Since these quantities are saved with attributes `public,protected`, the subroutine `initialize_const_motion` must be saved within the current module `pusher_tetra_orbit_mod`, otherwise it would not be allowed to set the values. The constants of motion will retain their set values for a number of tetrahedral pushings until the next time step is executed. Usually, between time steps collision events will occur when performing Monte Carlo simulations, as a consequence the constants of motion may change and have to be defined anew.

4.1.2 Particle pushing algorithm

For a given time step, after initializing the constants of motion, the subroutine `pusher_tetra_orbit` is called with the initial conditions of the current pushing. At the end of the subroutine execution it returns the new starting conditions for the next pushing as well as the remaining time of the current step. An overview of the call parameters of the subroutine is given in tab. 4.2.

Table 4.2: Parameter overview for `pusher_tetra_orbit`

Data type	Name	Description
`integer, intent(inout)`	`ind_tetr_inout`	current tetrahedron index
`integer, intent(inout)`	`iface`	current face index of tetrahedron where orbit converged, 0 if not converged
`double precision, dimension(3), intent(inout)`	`x`	current particle position in *global* coordinates, i.e. not with respect to the first node of a tetrahedron
`double precision, intent(inout)`	`vpar`	parallel velocity of the particle with respect to **B**
`double precision, dimension(3), intent(out)`	`z_final`	final particle position in *local* coordinates, needed for calculation of the flux tube volume used in another application
`double precision, intent(in)`	`t_remain_in`	remaining time of the current integration step, which consists of many pushings
`double precision, intent(out)`	`t_pass`	flight time of the current pushing step
`logical, intent(out)`	`boole_t_finished`	boolean stating if the remaining step time has been reached in the current pushing
`integer, intent(out)`	`iper_phi`	+1,-1 if the particle travels through the $\varphi = 0$-plane in $-\varphi, +\varphi$ direction, 0 otherwise

Initialize pusher

In the `pusher_tetra_orbit` subroutine, first an initializer subroutine `initialize_pusher_tetra_orbit_mod` is called. Here, the initial conditions are used to compute the coefficients a_l^i, b^i for the ordinary differential equation set 3.1, representing the equations of motion [4]. One should emphasize here, that this ode set is solved within a shifted coordinate system, where the coordinate origin $\mathbf{z}_0$ lies on the first vertex of a given tetrahedron. By convention in this project, when referring to a position in the global coordinate system one denotes the variable `x`, when referring to a position inside the local shifted coordinate system one uses `z` instead.

Initial guess of exit plane

Now, that the necessary components of the ODE set 3.1 have been initialized, the next occurring orbit intersection needs to be computed by the pusher routine. Since this must be done efficiently, a numerically inexpensive approximative quadratic solution is first evaluated by subroutine `quad_analytic_approx` to compute the guess for the orbit parameter `tau` at the first intersection of the particle trajectory with the cell boundary. Based on the result for the orbit parameter `tau`, an integration step is performed for the given step length using an RK4 solver, this integrator type is explained in more detail in appendix B. The RK4 integrator subsequently returns the evaluated position for the specified value for `tau`. In general, due to inaccuracies in the approximation, this value does not correspond to a converged orbit position. On a sidenote, in the context of the pusher routine, converged simply means that the particle position is within a defined convergence distance to a given tetrahedral plane. This distance is given by 10^{-10} times the normal distance of the first vertex within a given tetrahedron to its opposing cell boundary spanned by vertices 2,3 and 4 of the given tetrahedron. In addition, the normal velocity, which can also be computed from the output of the RK4 step, must have a negative sign in order for the convergence to be valid. The negative sign merely states that the particle is flying outwards of the tetrahedron, if there particle flew inwards, it would therefore not be accepted. Now, since the orbit position is generally not yet converged after the quadratic approximation, one next applies Newton's method for the face convergence by calling the subroutine `newton_face_convergence`. A detailed description of this approach is given by M. Eder *et al* [4].

4.1.3 Convergence and validation loop `conv_val_loop`

What one has obtained so far is a proposal for the exit plane and a converged orbit position on this plane. There may still be some problems, however, since for example Newton's method can fail if the orbit in fact turns before it intersects with the plane. Furthermore, it might happen that it does converge on the suggested plane but at the point of convergence it had already left the tetrahedron through another plane which is not allowed. Such cases need to be checked and handled appropriately. For this purpose, the *convergence and validation loop* `conv_val_loop` was implemented. This loop starts directly after the above mentioned quadratic approximation just before the convergence using Newton's method. Here, in each iteration of the loop, the algorithm tries to converge the particle orbit position on the currently proposed intersection face. Next, if convergence is reached the algorithm checks for the remaining planes

if the particle lies outside the tetrahedron. If this is not the case furthermore the normal velocity is checked to see if the particle flies outside. In case this is also correct, the particle is considered converged and accepted. In any case, where an error is detected, an appropriate approach is suggested. In most cases this involves using the quadratic approximation to suggest a different face, however, in some special cases this is not sufficient. Therefore, one calls inside the loop `conv_val_loop` an additional convergence routine `last_line_defense`, which no longer opts for high computational efficiency but rather for a reliable way of finding the intersection point. This subroutine is very comprehensive, but a central piece of it is a bisection scheme. For a short overview, in this scheme one computes the relative particle positions to all four faces and furthermore checks the normal velocities. Here, if the particle is inside the tetrahedron, the current step length is doubled and an integration step is performed. If the particle is now outside the tetrahedron, the last integration step is halved and integrated back in negative τ-direction. This is done in an iterative scheme until a converged particle position has been found and albeit computationally expensive, the *lastlineofdefense* solver remains an indispensable element of the algorithm due to its high reliability.
The structure of the loop `conv_val_loop` is presented in pseudo-code below.

```
conv_val_loop:  do i = 1,5
     boole_converged = .true.
     call newton_face_convergence(z,tau,iface_new,..)
     if (Newton's method failed) then
          allowed_faces(iface_new)=.false.
          if (all allowed_faces forbidden) then
               call last_line_defense(z,tau,iface_new,..)
               cycle conv_val_loop
          endif
          call quad_analytic_approx(z,allowed_faces,dtau,..)
          if (quadratic approximation failed) then
               call last_line_defense(z,tau,iface_new,..)
               boole_converged = .false.
```

```
          cycle conv_val_loop
     endif
     call integration_step(z,dtau,..)
     tau = tau + dtau
     boole_converged = .false.
     cycle conv_val_loop
endif
three_planes_loop:  do j=1,3
     k = modulo(iface_new+j-1,4)+1
     if (particle is outside face k) then
          allowed_faces(iface_new)=.false.
          if (all allowed_faces forbidden) then
               call last_line_defense(z,tau,iface_new,..)
               boole_converged = .false.
               cycle conv_val_loop
          endif
          if (face k is not forbidden in allowed_faces) then
               iface_new = k
               boole_converged = .false.
               cycle conv_val_loop
          endif
     endif
enddo three_planes_loop
     if (normal velocity at iface_new points inwards) then
          allowed_faces(iface_new)=.false.
          if (all allowed_faces forbidden) then
               call last_line_defense(z,tau,iface_new,..)
               boole_converged = .false.
               cycle conv_val_loop
          endif
          call quad_analytic_approx(z,allowed_faces,dtau,..)
```

```
            if (quadratic approximation failed) then
                call last_line_defense(z,tau,iface_new,..)
                boole_converged = .false.
                cycle conv_val_loop
            endif
            call integration_step(z,dtau,..)
            tau = tau + dtau
            boole_converged = .false.
            cycle conv_val_loop
        endif
        if (tau is negative) then
            allowed_faces(iface_new)=.false.
            z = z_init
            tau = 0.d0
            if (all allowed_faces forbidden)
                call last_line_defense(z,tau,iface_new,..)
                boole_converged = .false.
                cycle conv_val_loop
            endif
            call quad_analytic_approx(z,allowed_faces,dtau,..)
            if (quadratic approximation failed) then
                call last_line_defense(z,tau,iface_new,..)
                boole_converged = .false.
                cycle conv_val_loop
            endif
            call integration_step(z,dtau,..)
            tau = tau + dtau
            boole_converged = .false.
            cycle conv_val_loop
        endif
        exit conv_val_loop
enddo conv_val_loop
```

From the pseudo-code one can see, that a lot of thought has gone into efficiently computing the next intersection. However, due to the used approach of guessing the exit face with an approximation, many special cases of particle trajectories had to be taken into account such that the logic deals with them correctly.

4.1.4 Final processing

The last steps of the orbit integration are to first check if the computed time is in fact smaller than the remaining flight time. If this is not the case, instead of the converged orbit position, the corresponding position at the remaining time is evaluated and assumed. In this case the current tetrahedron index will be returned with `iface_new` being set to 0. If the computed time is smaller than the remaining time, the remaining time is reduced by the current value and the tetrahedron index adjacent to `iface_new` will be returned by calling the subroutine `pusher_handover2neighbour`. Furthermore, `iface_new` is changed to `neighbour_face(iface_new)` of the adjacent tetrahedron to mark the new entry face. Now, one must check if the intersection face is at a periodic boundary of the coordinate system. In this case the corresponding value of `i_per_theta/phi` times 2π is added to the respective coordinate component. Finally, the values for the current position `z` in local coordinates are converted to `x` in global coordinates, then the output values are returned and the orbit pushing is completed.

4.2 Pusher routine `pusher_tetra_orbit_analytic`

Implementation of analytic pusher from analytical solution

4.3 Search routines for tetrahedra with starting points

depending on the level of detail, the numerical pusher and the analytic pusher can be anything from a short overview to a lengthy description

Chapter 5

Monte Carlo simulation of particle transport using Gorilla

possibly shortly describe MC simulation and calculation of diffusion coefficient

Chapter 6

Conclusion and outlook

conclusion and outlook text

Appendix A

Lagrange polynomial interpolation

A.1 Introduction

In this chapter, a short introduction to Lagrange polynomial interpolation is presented. This writeup is based on the NIST Library of Mathematical Functions [2].
The nodes z_k are real or complex valued, the function values are $f_k = f(z_k)$. Given $(n+1)$ distinct points z_k with their corresponding function values f_k, the Lagrange interpolation polynomial is the unique polynomial $P_n(z)$ satisfying $P(z_k) = f_k$ while not exceeding order n, with $k = 0, 1, ..., n$. The Lagrange polynomial is given by

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

with Lagrange coefficients

$$\mathscr{L}_k(z) = \prod_{j=0, j\neq k}^{n} \frac{z - z_j}{z_k - z_j}$$

where the factor for $j = k$ is omitted in the product. The Lagrange coefficients are again polynomials with the property

$$\mathscr{L}_k(z_j) = \delta_{k,j},$$

thus, each $\mathscr{L}_k(z)$ has a weight of 1 if $z = z_k$ or 0 if $z = z_j$ with $j \neq k$. For this property, $P_n(z)$ goes exactly through all data points (z_k, f_k).

A.2 Application for a simple exponential

In practice one applies a low order polynomial interpolation for a small set of points lying close to the target position, this approach guarantees smoothness of the interpolated curve even for non-smooth data, thus effectively reducing high order polynomial oscillations. For demonstrative purposes, some Lagrange polynomials (line style: solid,

black) for a basic exponential function (line style: dotted, blue) are depicted below.

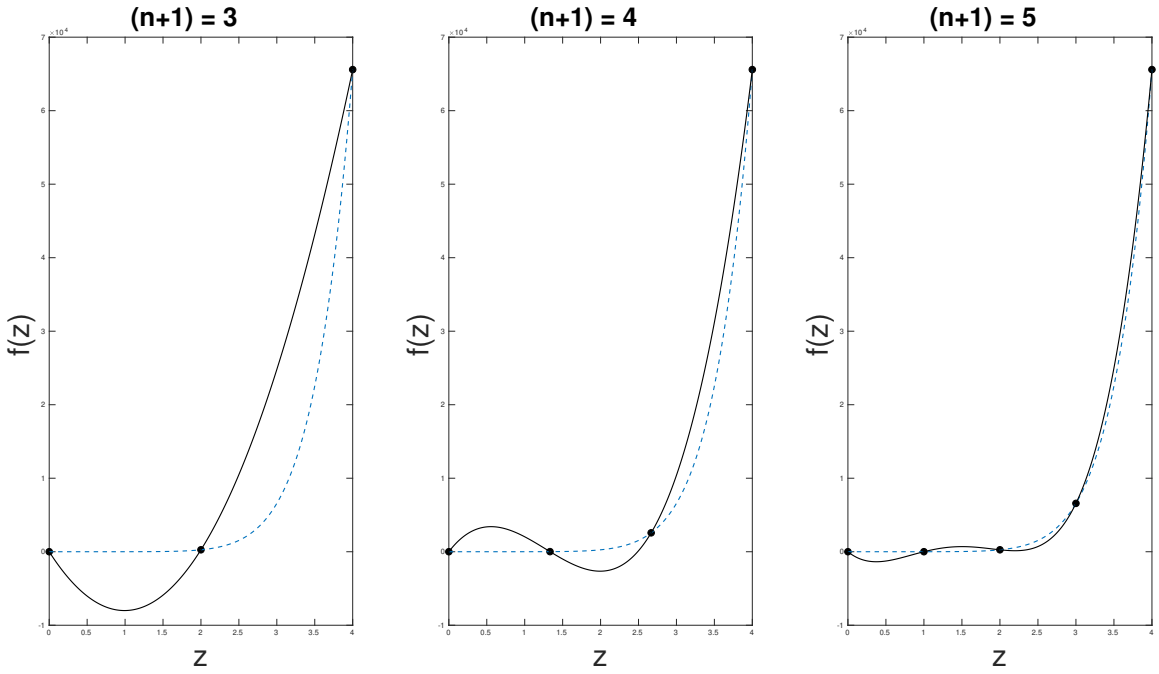


Figure A.1: Lagrange polynomials of order n with equidistant z_k for $f(z) = e^z$

Appendix B

Runge-Kutta integration

B.1 General formulation

In this chapter, a brief summary of Runge-Kutta integration is presented, based on the comprehensive summary by E. Hairer in [6]. In numerical analysis, efficient integration methods for solving initial value problems of the form $y' = f(x, y)$ were initially implemented by Euler (1768), although later further methods based on his work were developed by Runge (1895) and Kutta (1905). The most widely known algorithm is the so-called fourth order Runge-Kutta solver (commonly abbreviated *RK4*), however, an entire generalized class of integrators has since been derived. Such an integrating scheme is fully described by the coefficients of the corresponding *Butcher tableau* given in B.1:

Table B.1: *Butcher tableau* for a general s-stage *Runge-Kutta method*

0					
c_2	a_{21}				
c_3	a_{31}	a_{32}			
$\vdots$	$\vdots$	$\vdots$	$\ddots$		
c_s	a_{s1}	a_{s2}	$\dots$	$a_{s,s-1}$	
	b_1	b_2	$\dots$	b_{s-1}	b_s

Generally, the condition

$$c_i = \sum_{j=1}^{i-1} a_{ij} \tag{B.1}$$

is further imposed, which greatly simplifies the problem of deriving order conditions for higher order methods.
Using the coefficients of B.1 one can explicitly calculate the approximate solution to

the initial value problem for a single step $\Delta x = h$ by computing

$$y_1 = y_0 + h(b_1 k_1 + \ldots + b_s k_s) \tag{B.2}$$

with

$$\begin{aligned} k_1 &= f(x_0, y_0) \\ k_2 &= f(x_0 + c_2 h, y_0 + h a_{21} k_1) \\ k_3 &= f(x_0 + c_3 h, y_0 + h(a_{31} k_1 + a_{32} k_2)) \\ &\ldots \\ k_s &= f(x_0 + c_s h, y_0 + h(a_{s1} k_1 + \ldots + a_{s,s-1} k_{s-1})). \end{aligned} \tag{B.3}$$

B.2 RK4 with application

For the specific case of the *RK4*-method, which is also implemented in the *GORILLA* code, the corresponding *Butcher tableau* is given in B.2.

Table B.2: *Butcher tableau* for the *RK4*-method

0				
$\frac{1}{2}$	$\frac{1}{2}$			
$\frac{1}{2}$	0	$\frac{1}{2}$		
1	0	0	1	
	$\frac{1}{6}$	$\frac{1}{3}$	$\frac{1}{3}$	$\frac{1}{6}$

One is especially interested in the application of this scheme to an ODE system of the shape

$$f(\tau, \mathbf{z}(\tau)) = \mathbf{f}(\mathbf{z}(\tau)) = \frac{\mathrm{d}\mathbf{z}(\tau)}{\mathrm{d}\tau} = \hat{\mathbf{a}} \cdot \mathbf{z}(\tau) + \mathbf{b} \tag{B.4}$$

with initial conditions $\mathbf{z}(0) = \mathbf{z}_0$. Note that $\mathbf{f}(\tau, \mathbf{z}(\tau))$ does not explicitly depend on τ,

thus, (B.2) and (B.3) yield for a single *RK4* step with $h = \tau$

$$\begin{aligned}
\mathbf{z}_{RK4} &= \mathbf{z}_0 + \tau\left(\frac{1}{3}k_1 + \frac{1}{6}k_2 + \frac{1}{6}k_3 + \frac{1}{3}k_4\right) \\
k_1 &= \mathbf{f}(\mathbf{z}_0) \\
k_2 &= \mathbf{f}\left(\mathbf{z}_0 + \frac{\tau}{2}\mathbf{f}(\mathbf{z}_0)\right) \\
k_3 &= \mathbf{f}\left(\mathbf{z}_0 + \frac{\tau}{2}\mathbf{f}\left(\mathbf{z}_0 + \frac{\tau}{2}\mathbf{f}(\mathbf{z}_0)\right)\right) \\
k_4 &= \mathbf{f}\left(\mathbf{z}_0 + \tau\mathbf{f}\left(\mathbf{z}_0 + \frac{\tau}{2}\mathbf{f}\left(\mathbf{z}_0 + \frac{\tau}{2}\mathbf{f}(\mathbf{z}_0)\right)\right)\right),
\end{aligned} \tag{B.5}$$

which allows to explicitly write the approximate *RK4*-solution for this ODE system as

$$\begin{aligned}
\mathbf{z}_{RK4}(\tau) &= \mathbf{z}_0 + \frac{\tau}{6}\mathbf{f}(\mathbf{z}_0) + \frac{\tau}{3}\mathbf{f}\left(\mathbf{z}_0 + \frac{\tau}{2}\mathbf{f}(\mathbf{z}_0)\right) + \frac{\tau}{3}\mathbf{f}\left(\mathbf{z}_0 + \frac{\tau}{2}\mathbf{f}\left(\mathbf{z}_0 + \frac{\tau}{2}\mathbf{f}(\mathbf{z}_0)\right)\right) \\
&+ \frac{\tau}{6}\mathbf{f}\left(\mathbf{z}_0 + \tau\mathbf{f}\left(\mathbf{z}_0 + \frac{\tau}{2}\mathbf{f}\left(\mathbf{z}_0 + \frac{\tau}{2}\mathbf{f}(\mathbf{z}_0)\right)\right)\right) .
\end{aligned} \tag{B.6}$$

It is important to note that the *RK4*-method has the property that for sufficiently smooth functions, the approximate *RK4*-solution $\mathbf{z}_{RK4}(\tau)$ coincides with the fourth order Taylor expansion of the analytical solution for $\mathbf{z}(\tau)$. The associated errors are therefore of order $\mathcal{O}(\tau^5)$.

B.3 Runge-Kutta-Fehlberg - RK45

Although the introduced *RK4*-method is a very useful tool, its accuracy greatly depends on the chosen step size h, while the routine generally does not yield an estimate for the error, if not computed separately. A clever way to circumvent this problem was introduced by E. Fehlberg (1969), namely, to evaluate a given step successively with proposed fourth-order and fifth-order routines and then compute the difference of these two results. If the difference is smaller than a set tolerance, the step is accepted, if not, the step is discarded and the previous step size h is halved for the next attempt. The Butcher tableau for the *RK45* method is given by

Table B.3: *Butcher tableau* for the *RK45*-method

0						
$\frac{1}{4}$	$\frac{1}{4}$					
$\frac{3}{8}$	$\frac{3}{32}$	$\frac{9}{32}$				
$\frac{12}{13}$	$\frac{1932}{2197}$	$-\frac{7200}{2197}$				
1	$\frac{439}{216}$	-8	$\frac{3680}{513}$	$-\frac{845}{4104}$		
$\frac{1}{2}$	$-\frac{8}{27}$	2	$-\frac{3544}{2565}$	$\frac{1859}{4104}$	$-\frac{11}{40}$	
	$\frac{16}{135}$	0	$\frac{6656}{12825}$	$\frac{28561}{56430}$	$-\frac{9}{50}$	$\frac{2}{55}$
	$\frac{25}{216}$	0	$\frac{1408}{2565}$	$\frac{2197}{4104}$	$-\frac{1}{5}$	0

Here, the first row at the bottom gives the coefficients for the fifth order method, the second row gives the coefficients for the fourth order method.

Bibliography

[1] W.D. D'haeseleer,*Flux Coordinates and Magnetic Field Structure - A Guide to a Fundamental Tool of Plasma Theory*, , Springer Verlag, 1991

[2] Frank W. J. Oliver, Daniel W. Lozier, Ronald F. Boisvert, Charles W. Clark, *NIST Handbook of Mathematical Functions*, Cambridge University Press, 2010

[3] Sergei Kasilov, private communication 2019

[4] Eder Michael, *Three-dimensional geometric integrator for charged particle orbits in toroidal fusion devices*, Diploma Thesis, 2018

[5] Reviews of Plasma Physics, Volume 2 - Solov'ev Morozov ,...

[6] Solving Ordinary Differential Equations I, Nonstiff Problems, E. Hairer, S.P. Nørsett, G. Wanner, Springer Series in Computational Mathematics, 2008

[7] 46th EPS Conf. on Plasma Physics, 2019, ECA Vol. 43C, P5.1100.

[8] Delaunay mesh construction, Dyer, Ramsay and Zhang, Hao and Möller, Torsten, Eurographics Symposium on Geometry Processing, Barcelona, Spain, 2007, ISSN/ISBN 978-3-905673-46-3