Chapter 1

Pusher algorithms for particle orbits

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. To illustrate, a schematic figure of a particle orbit intersecting with the tetrahedral cell boundaries, taken from [**?**], is shown in Fig. 1.1.

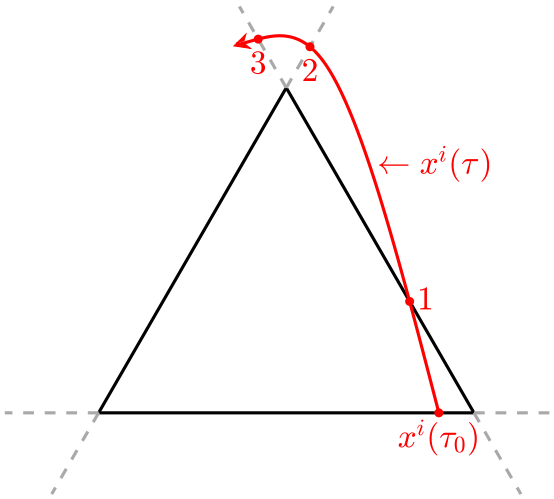

Figure 1.1: Intersections of the particle orbit $x^i(\tau)$ with planes confining the cell are displayed. For demonstrative purposes, the tetrahedral cell is depicted as a two-dimensional triangle. The particle enters the cell at $x^i(\tau_0)$ and exits again at ①. This figure is taken from [**?**].

In this figure, one can see the particle orbit $x^i(\tau)$ intersects with the planes lying at the cell boundaries several times. Here, only the first intersection is of interest, as the particle leaves the tetrahedron at this position. This first intersection must now be efficiently found by the algorithm. Since such an algorithm can be thought of as a pushing of the particle orbit through the tetrahedron from one cell boundary to the next, 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 [?], 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_rk`, an overview of the code is given below, however, due to large similarities with the previous approach discussed in [?], the new routine will be presented in less detail. Apart from this routine, a second routine named `pusher_tetra_poly` 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 polynomial order of (??) 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 for efficiently finding the corresponding tetrahedron index to start a calculation.

1.1 Pusher routine `pusher_tetra_rk`

As discussed, the pusher subroutine `pusher_tetra_rk` 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_gorilla` which is given the initial conditions of the particle and iteratively calls the pusher subroutine `pusher_tetra_rk` until the set time is reached, meaning that the particle is pushed consecutively through each cell. 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 to reach the set time 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_gorilla` and the components of the module `pusher_tetra_orbit_mod` is given in figure 1.2.

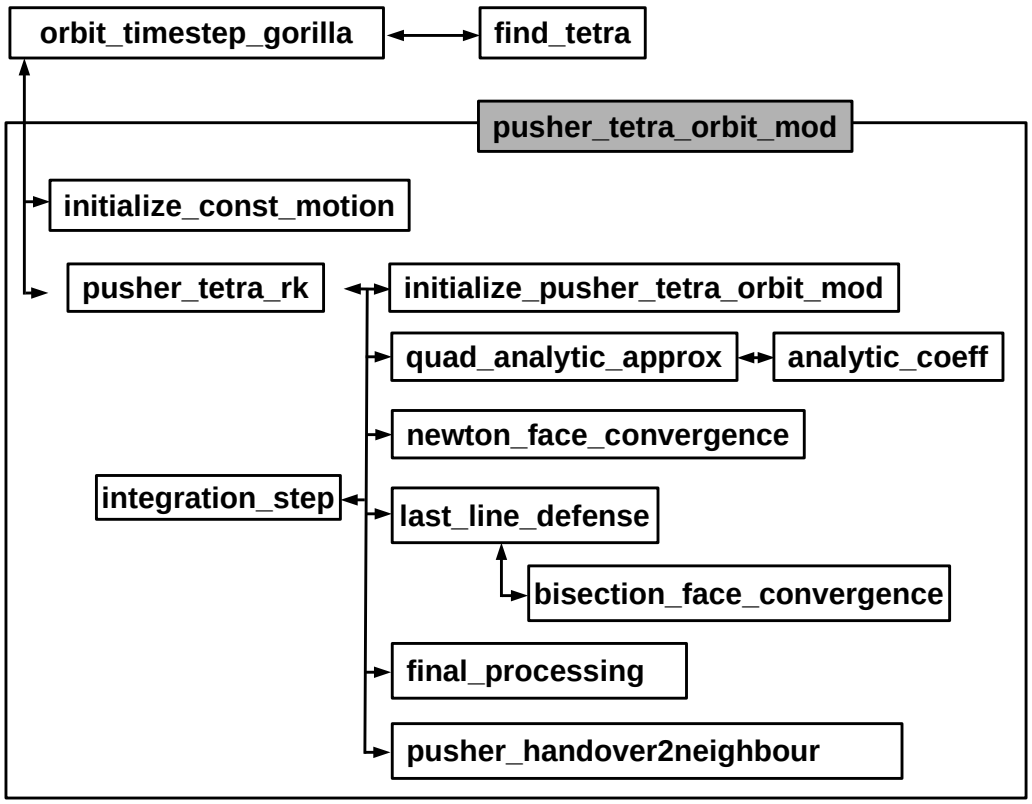

Figure 1.2: Code structure of `pusher_tetra_orbit_mod` and associated subroutine

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_gorilla` with arguments `(x,vpar,vperp,t_step,boole_initialized,ind_tetr,iface)`. This list of parameters is explained in tab. 1.1.

Table 1.1: Parameter overview for wrapper subroutine `orbit_timestep_gorilla`, 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

1.1.1 Initializing constants of motion

In figure 1.2, one can see the code diagram which gives an overview of the different subroutines. When starting a calculation in `orbit_timestep_gorilla` 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.

1.1.2 Particle pushing algorithm

For a given time step, after initializing the constants of motion, the subroutine `pusher_tetra_rk` 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. 1.2.

Table 1.2: Parameter overview for `pusher_tetra_rk`

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 $\mathbf{B}$
`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 **??**, representing the equations of motion [**?**]. 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 **??** 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 **??**. 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. 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 assures here, that only outflowing particles (i.e. with negative normal velocity) are accepted as a solution. 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* [**?**].

1.1.3 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 as it would be the actually correct exit plane instead of the proposed one. 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 *last line of defense* solver remains an indispensable element of the algorithm due to its high reliability. This effect on performance remains small, however, as only a small portion of particle pushings actually need to be solved by the *last line of defense* solver.
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 instead of computing all intersections with all four planes, which would be computationally much more expensive, many special cases of particle trajectories had to be taken into account such that the logic deals with them correctly.

1.1.4 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.

1.2 Pusher routine `pusher_tetra_poly`

In chapter **??**, the analytical solution to the linearized equations of motion was derived. There, a power series expansion of the solution was presented in equation (**??**), where the RK4 method corresponded to the same expansion, only up to fourth order. With this analytical expansion, one actually has many new possibilities in computing the next orbit intersection. Unlike with the RK4 method, one has here an explicit expression of the orbit in orders of τ. This allows to use the expansion for the position in the Hesse normal form of the tetrahedral planes. By doing this, one obtains a polynomial of τ for each plane, where the smallest positive value of τ corresponds to the next exiting position. For low enough orders of this expansion, and therefore of the polynomial, one can in fact use analytical formulas to find the solutions for τ. On this working principle, an alternative pushing routine `pusher_tetra_poly` is therefore implemented. Due to the stringent requirements on computational efficiency,

however, this subroutine uses a similar approach as the above described subroutine `pusher_tetra_orbit`. A short overview of the analytical approach is given below.
Using equation (**??**), one can write the expansion of the solution as a polynomial of a defined order.

$$\mathbf{z}_{poly}(\tau) = \mathbf{z}_0 + \mathbf{a}\tau + \mathbf{b}\tau^2 + \dots$$

The three spatial components (first three components) of $\mathbf{z}_{poly}$ can now be inserted into the plane equations of the four tetrahedral faces. The plane equations are hereby given by the Hesse normal form using the normal vectors of the plane and the normal distance of the first vertex with respect to the current plane. Since the first vertex of a given tetrahedron lies by convention on the tetrahedral planes {2,3,4} but not on plane 1, one can write this explicitly as

$$\text{Plane 1: } 0 = d_\perp + \vec{n}_1 \cdot \mathbf{z}_{poly}(\tau),$$

$$\text{Planes i=\{2,3,4\}: } 0 = \vec{n}_i \cdot \mathbf{z}_{poly}(\tau).$$

Here, $d_\perp$ denotes the normal distance of the first tetrahedron vertex to the first plane. The normal vectors of the respective planes i are given by $\mathbf{n}_i$. Since $\mathbf{z}_{poly}(\tau)$ is in the *local* coordinate system, with the first vertex representing the coordinate origin, the coordinates of the first vertex do not appear in these equations.
Using this form, one obtains a polynomial of the same order as $\mathbf{z}_{poly}(\tau)$. Here, the roots of the polynomial represent the intersection points of the orbit with the given plane. The task is now, to efficiently find the smallest positive root τ_{exit} of all four equations. This value of the orbit parameter τ corresponds to the exit position of the particle through the tetrahedral cell boundary. In order to obtain the same level of accuracy as with RK4, one must furthermore compute orbits with a Taylor expansion of order four. This is the highest order of a polynomial for which analytical solutions for the roots of a polynomial exist, based on case differentiations of the parameters. Higher order polynomials are therefore not suggested nor currently supported. Moreover, lower order polynomial roots are much more efficiently solvable, for this reason additional logics is implemented for the analytic pusher, as computational efficiency remains a key criterion and computing all possible intersections does slow down the algorithm. The current approach is therefore to first evaluate the polynomial coefficients of the Taylor expansion for the desired order. Using this solution up to only the quadratic order, one can evaluate a guess for the exit plane of the particle by computing the roots of the polynomial obtained by the Hesse normal form for all planes and taking the smallest positive root corresponding to the guess for the exit position. The value for τ corresponding to the smallest positive root is then used to compute the positions of

the particle using the expansion coefficients of the desired order to check convergence more accurately and to be able to take measures to reach convergence. Based on this principle, additional logics is implemented for the subroutine `pusher_tetra_poly`, like for the subroutine `pusher_tetra_rk`, however, this is very comprehensive and therefore not within the scope of this thesis.

1.3 Search routine for tetrahedra with starting points

Since generally, a user defines a particle starting position `x` in global coordinates, rather than specifying a tetrahedron index and a local position, the search routine `find_tetra` is implemented for finding the corresponding tetrahedron index. For this, moreover, a function `is_inside` is implemented which allows to check if the particle position lies inside a proposed tetrahedron. This function uses the Hesse normal form to compute the distances to all tetrahedral planes. The necessary quantities are the coordinate position of the first vertex, the normal distance of this vertex to the opposing plane and the four normal vectors of the planes, these are all accessable in module `tetra_grid_mod`. Next, due to the axisymmetry of the grid, based on the current phi position one can vastly reduce the number of possible tetrahedra by allowing only tetrahedra of the current φ-slice. Now, a loop over all possible tetrahedron indices is implemented to check if the particle lies inside. Once the correct tetrahedron has been found, the distances to the four planes need to be checked for random convergence on a plane. If this is the case, also the normal velocity with respect to that plane must be evaluated, since the pusher always assumes that a given particle flies inwards which could lead to errors in the logics. If the particle, however, is converged on a plane and flies outwards, instead of the current tetrahedron index the adjacent neighbor at this face index is returned with the corresponding value for `iface_new`.