Chapter 1

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 [**?**], 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 [**?**], 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.

1.1 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 1.1.
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. 1.1.

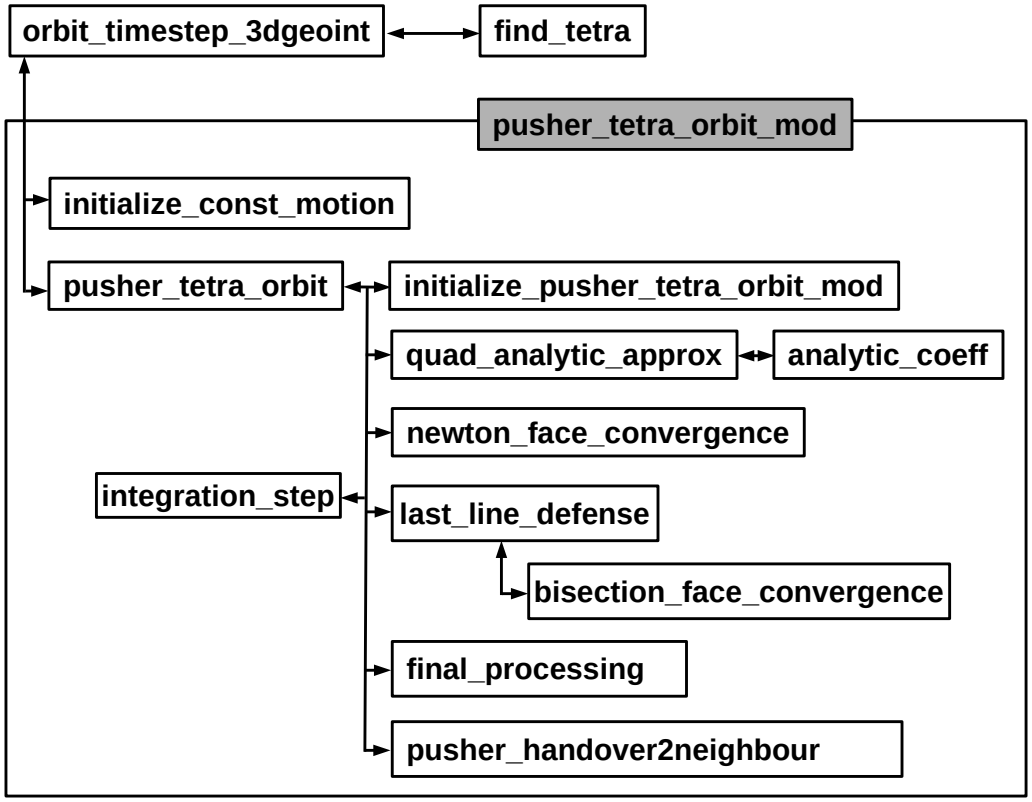

Figure 1.1: Code structure of `pusher_tetra_orbit_mod` and associated subroutine

Table 1.1: 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

1.1.1 Initializing constants of motion

In figure 1.1, 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.

1.1.2 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. 1.2.

Table 1.2: 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 **??**, representing the equations of motion [**?**].

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. 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. 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* [**?**].

1.2 Pusher routine `pusher_tetra_orbit_analytic`

Implementation of analytic pusher from analytical solution

1.3 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