Final Report on Award DE-SC0000834
Simulations of alpha wall load in ITER

Johan Carlsson, P.I.
Tech-X Corporation
5621 Arapahoe Avenue, Suite A
Boulder, CO 80303

Performance Period: July 20, 2009 - July 19, 2010
Grant supported by DOE Office of Fusion Energy Sciences

October 19, 2010

Executive Summary

To perform realistic simulations of wall load in tokamaks from energetic ions, such as fusion alphas in ITER, one needs to have both a model of the relevant physics, including interaction with MHD modes, and an engineering-grade model of the plasma-facing surface of the vacuum vessel, loaded from a CAD file. Also, in the presence of a realistic, non-axisymmetric magnetic field, the validity of the predominantly used drift-orbit approximation for energetic-ion dynamics is currently being questioned. Given these challenges, there currently is no simulation code capable of making a reliable, quantitative prediction of the wall load in ITER from fusion-born alpha particles.

The DOE OFES SBIR Phase I project "Simulations of alpha wall load in ITER" (award number DE-SC0000834) made good progress on the latter two issues by i) developing a wall library that loads a CAD file and provides an interface for particle-wall collisions and ii) adding full gyro-orbit dynamics to the DELTA5D [1] drift-orbit Monte-Carlo code.

Technical Results

The Phase I project had four specific technical objectives: 1) Find the optimal drift-orbit integrator, 2) Get gyro-orbit integrator working, 3) Demonstrate feasible approach to model realistic ITER wall, and 4) Determine efficiency of retrograde Monte Carlo for reducing statistical noise. All four Phase I objectives have been met (and well exceeded for the 3rd objective) and will be discussed in detail below.

Task 1: Find the optimal drift-orbit integrator

Before the Phase I project, DELTA5D had four different drift-orbit integrators: LSODE, Gill's method and 2nd and 4th order Runge-Kutta. We have now added five more: CVODE [2], Rome-Cary symplectic [3, 4], Bulirsch-Stoer, leap frog and another version of Runge-Kutta.

The CVODE integrator [2] is part of the LLNL SUite of Nonlinear and DIfferential/ALgebraic equation Solvers (SUNDIALS) [5]. It is an implementation of the VODE [6] algorithm, a general purpose solver similar to LSODE, but which uses variable-coefficient methods instead of the fixed-step-interpolate methods in LSODE. CVODE is therefore generally recommended as a replacement for LSODE, which is part of the SUNDIALS predecessor ODEPACK [7].

The Rome-Cary symplectic integrator is a 2nd order symplectic integrator. It is often advantageous to use algorithms where the discretized equations have the same quantities conserved as the original equations. For orbit integration, the symplectic integrators have such a conservation property: they conserve phase-space volume. As a consequence, constants of motion are preserved and a periodic orbit can be followed for thousands of periods and still remain periodic, despite temporal discretization errors.

To compare CVODE and Rome-Cary symplectic with the already used LSODE, we first installed `cvode-2.6.0` and added the necessary code to DELTA5D to call CVODE through SUNDIALS FCVODE Fortran interface. Implementing the Rome-Cary symplectic integrator in DELTA5D was less straightforward. The integrator was implemented as a stand-alone C application called `symporbit` that we converted into a library we call `RoCaSymp`. The refactoring and

`i_ode_meth`	orbit integrator
1	LSODE
2	Gill's method
3	4th order Runge-Kutta
4	2nd order Runge-Kutta
5	CVODE
6	Rome-Cary symplectic
7	Bulirsch-Stoer
8	Leap frog
9	Runge-Kutta

Table 1: Values of input parameter `i_ode_meth` for the different orbit integrators. 1-4 were available before the Phase I project, 5-9 were added during the project.

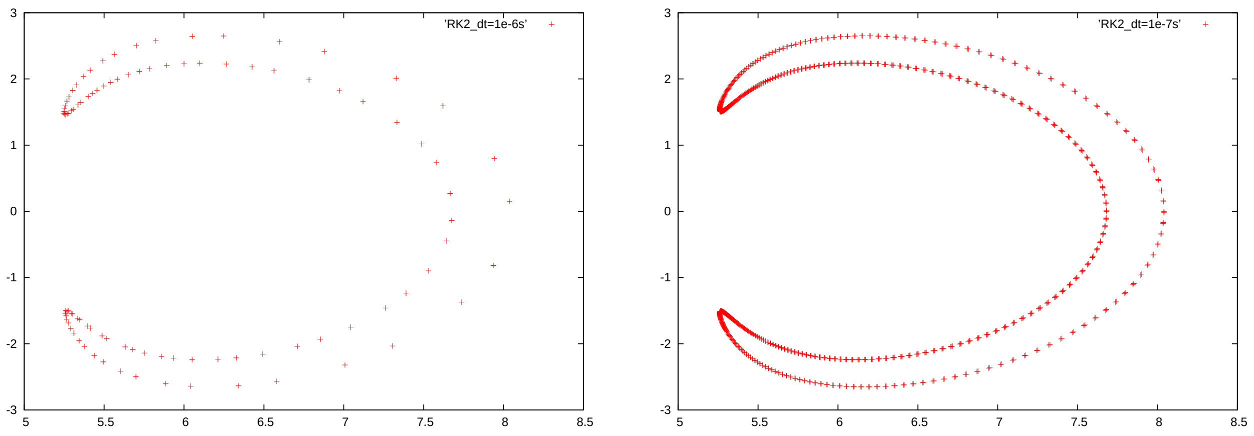


Figure 1: $Z(R)$ [both in meters] for orbit integration with 2nd order Runge-Kutta (`i_ode_meth`=4) for $\Delta t = 1\mu s$ (left) and $\Delta t = 0.1\mu s$ (right)

resulting debugging took somewhat longer than expected. We discovered that Bulirsch-Stoer, leap frog and Runge-Kutta had all been implemented in `symporbit` so we modified the library interface to allow a driver to call also these other integrators. We also wrote Fortran wrappers to make this library callable from DELTA5D. With the library and Fortran wrappers completed and working correctly, adding the necessary calls to the `RoCaSymp` library in DELTA5D was quick and easy. The orbit integrator invoked in DELTA5D is determined by the value of the input parameter `i_ode_meth`, as shown in Table 1.

To compare orbit integrators, we use an axisymmetric ITER equilibrium and follow an orbit for 100 μs with two different time steps: $0.1\,\mu s$ and $1.0\,\mu s$. We then plot $R(Z)$ and the energy change in eV over time, $\delta W(t)$. We first use the three existing, lightweight, explicit integrators.

The drift orbits in the poloidal plane look virtually the same for all the integrators, and so are only shown for 2nd order Runge-Kutta in Fig. 1. The three explicit integrators (`i_ode_meth`=2,3,4) are fast and have an acceptable error at the shorter time step. Gill's method (Fig. 4) and 4th order Runge-Kutta (Fig. 3) have very similar performance, but 2nd order Runge-

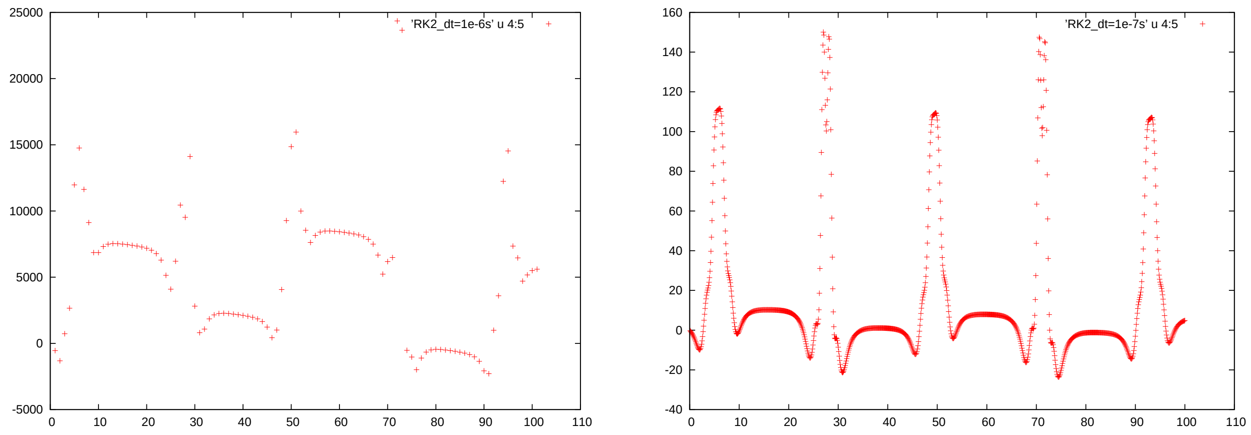


Figure 2: $\delta W(t)$ [energy change in eV vs. time in μs] for orbit integration with 2nd order Runge-Kutta (i_ode_meth=4) for $\Delta t = 1\mu s$ (left) and $\Delta t = 0.1\mu s$ (right)

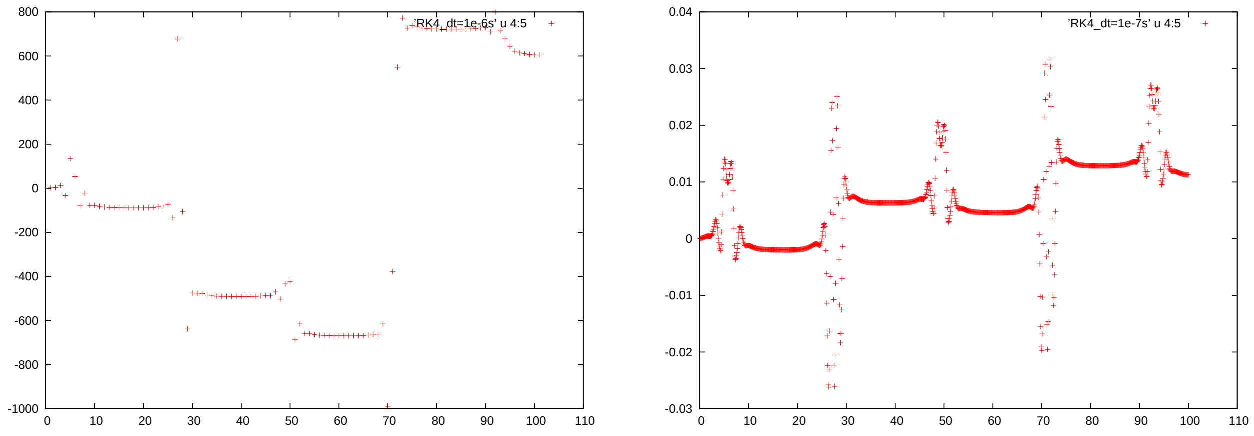


Figure 3: $\delta W(t)$ [energy change in eV vs. time in μs] for orbit integration with 4th order Runge-Kutta (i_ode_meth=3) for $\Delta t = 1\mu s$ (left) and $\Delta t = 0.1\mu s$ (right)

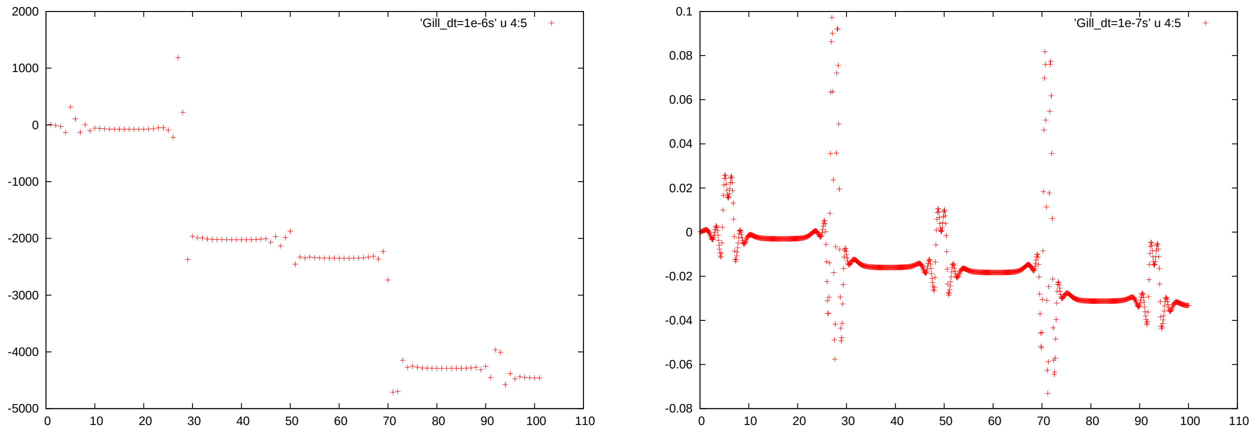


Figure 4: $\delta W(t)$ [energy change in eV vs. time in μs] for orbit integration with Gill's method (`i_ode_meth`=2) for $\Delta t = 1\mu s$ (left) and $\Delta t = 0.1\mu s$ (right)

Kutta (Fig. 2) is clearly inferior. The three implicit integrators, the old LSODE (`i_ode_meth`=1) and the new CVODE (`i_ode_meth`=5) and Rome-Cary symplectic (`i_ode_meth`=6) are all considerably slower. For LSODE (Fig. 5) and CVODE (Fig. 6) the increased computational cost pays off by allowing much larger time steps without increased energy error. CVODE is slightly faster than LSODE and also better at keeping the error within bounds independent of the time step.

The Rome-Cary symplectic integrator (Fig. 7) does work as expected. The error has no secular term, but is purely periodic. A periodic orbit might then become slightly deformed, but it will remain periodic even for very long integration times. However, the symplectic integrator is much slower than CVODE and the amplitude of its error is much larger.

The optimal drift-orbit integrator is therefore clearly CVODE.

Task 2: Get gyro-orbit integrator working

To simulate a realistic wall load, the full gyro dynamics are needed. To facilitate the wall/orbit-intersection computation, the same Cartesian coordinates were chosen for the gyro dynamics as for the wall model. Because it was the best drift-orbit integrator, CVODE was also used for the gyro-orbit integration. Implementation was straightforward. When a drift orbit reaches a specified flux surface (one or a few gyro radii from the wall), one (or more) gyro orbits are created and launched. We keep evolving the drift orbit to allow for later comparison between drift and gyro dynamics. Sub time stepping is used to resolve gyro period. For ITER the alpha-particle gyro period is around 30 ns compared to the typical time steps for drift-orbit integration of 100–1000 ns. 10–100 sub steps are therefore normally used for the gyro-orbit integration.

The transformation from the magnetic coordinates of a drift orbit to the Cartesian coordinates of a gyro orbit requires some computation. First the Cartesian components of the magnetic field, B_x, B_y and B_z, are computed from B and ι $(= 1/q)$ via finite differencing of the cylindrical coordinates as functions of the magnetic coordinates, $R(\psi, \theta, \phi)$, $Z(\psi, \theta, \phi)$ and $\phi_{cyl}(\psi, \theta, \phi)$. The total and parallel speeds, v and $v_{\|}$, respectively, are then trivially calculated from the energy W

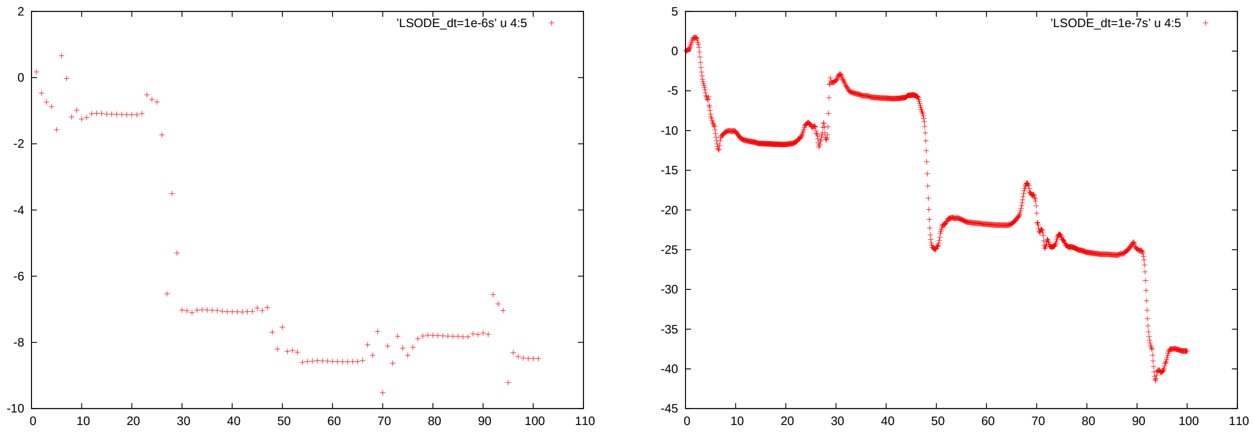


Figure 5: $\delta W(t)$ [energy change in eV vs. time in μs] for orbit integration with LSODE (i_ode_meth=1) for $\Delta t = 1\mu s$ (left) and $\Delta t = 0.1\mu s$ (right)

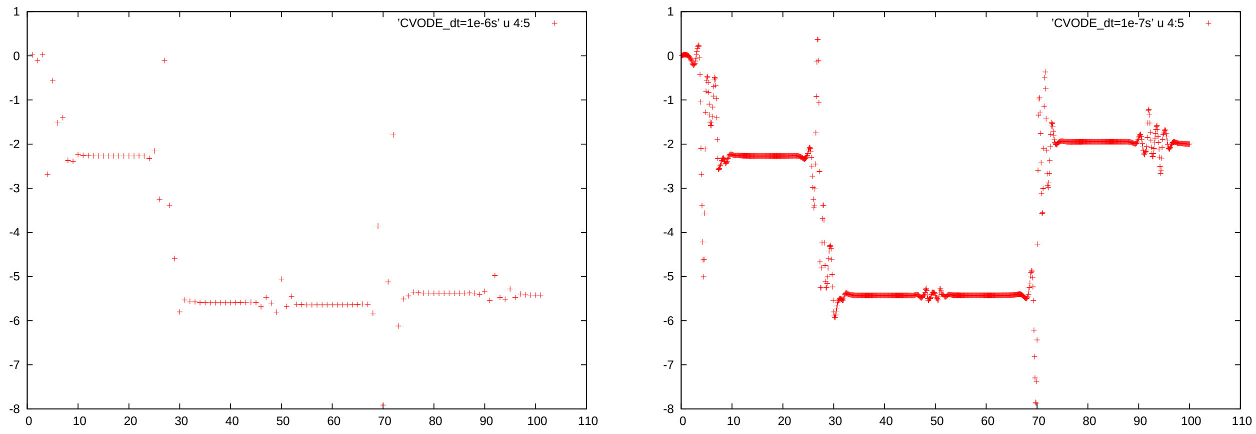
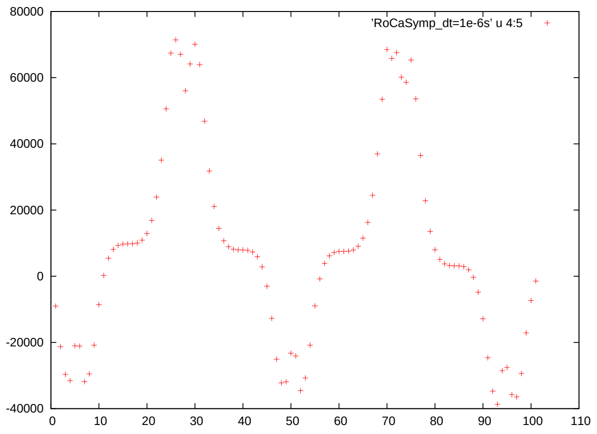


Figure 6: $\delta W(t)$ [energy change in eV vs. time in μs] for orbit integration with CVODE (i_ode_meth=5) for $\Delta t = 1\mu s$ (left) and $\Delta t = 0.1\mu s$ (right)



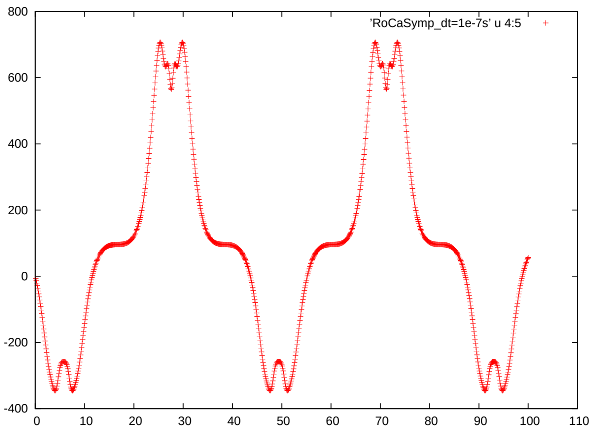


Figure 7: $\delta W(t)$ [energy change in eV vs. time in μs] for orbit integration with the Rome-Cary symplectic integrator (`i_ode_meth`=6) for $\Delta t = 1\mu s$ (left) and $\Delta t = 0.1\mu s$ (right)

and magnetic moment μ of the drift orbit. The Cartesian velocity components are the solution to the nonlinear, underdetermined system

$$\left\{ \begin{array}{l} B_x v_x + B_y v_y + B_z v_z = B v_\parallel \\ v_x^2 + v_y^2 + v_z^2 = v^2 \end{array} \right.$$

To solve for the Cartesian velocity of the gyro orbit, we use quasi-Newton iteration with Singular-Value Decomposition to invert the Jacobian. That is, we find the root of

$$\left\{ \begin{array}{l} F_1(v_x, v_y, v_z) = B_x v_x + B_y v_y + B_z v_z - B v_\parallel = 0 \\ F_2(v_x, v_y, v_z) = v_x^2 + v_y^2 + v_z^2 - v^2 = 0 \end{array} \right.$$

by iteratively solving the linear system $\mathbf{J}^n \cdot \boldsymbol{\delta}\mathbf{v}^n = -\mathbf{F}^n$, where the Jacobian $\mathbf{J} = d\mathbf{F}/d\mathbf{v}$ is a 2×3 matrix, and updating the approximate root $\mathbf{v}^{n+1} = \mathbf{v}^n + \boldsymbol{\delta}\mathbf{v}^n$. We use an approximate analytic Jacobian where the inhomogeneity of the magnetic field is neglected. We invert the underdetermined, 2×3 Jacobian with Singular-Value Decomposition (SVD). SVD finds the increment $\boldsymbol{\delta}\mathbf{v}$ that takes us toward the closest point on the gyro ring in velocity space. This algorithm proved quite robust and converges in half dozen iterations for initial guess $v_x = v_y = v_z = v$.

To calculate also the Cartesian spatial coordinates of the gyro orbit, we need the vector gyro radius, which is given by

$$\boldsymbol{\varrho} = -\frac{m}{eZ}\frac{\mathbf{v} \times \mathbf{B}}{B^2}$$

The particle location on the gyro orbit is then given by $\mathbf{r} = \mathbf{r}_{gc} + \boldsymbol{\varrho}$.

We now have all the six Cartesian phase-space coordinates of a gyro orbit. We note that the gyro phase of this orbit is arbitrary and is determined by the initial guess for the velocity coordinates found by quasi-Newton iteration. A fairly obvious generalization is then to split the drift orbit into an ensemble of gyro orbits with uniformly spaced gyro phases. The idea being that the ensemble better represents the drift orbit than a single gyro orbit would, and that the statistical noise in the wall-load simulations is reduced.

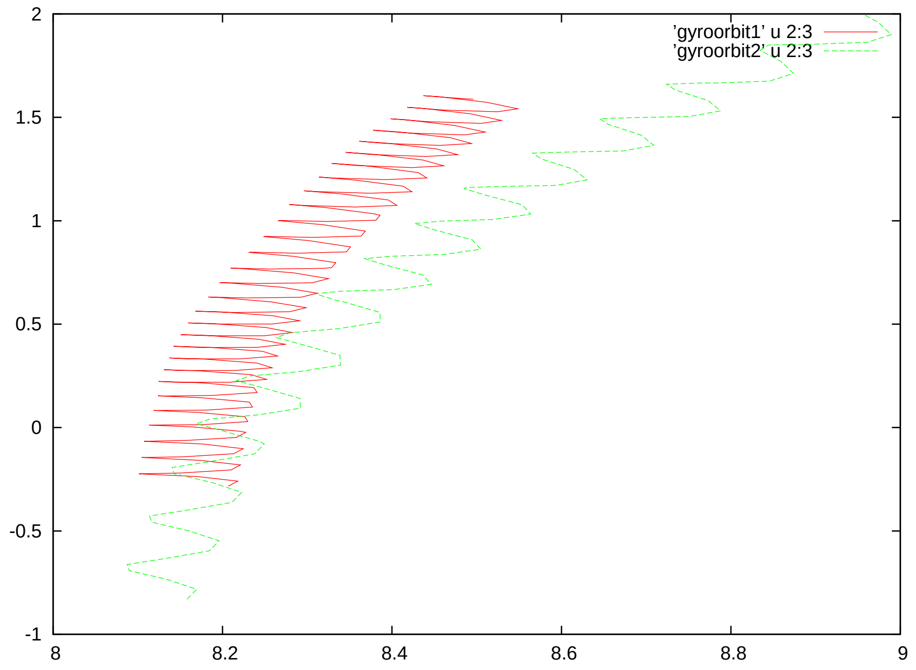


Figure 8: $Z(R)$ for two sample gyro orbits in DELTA5D created from two drift orbits (i.e. the ensemble size is one)

To create an ensemble of N gyro orbits, one can find the $\mathbf{v}_i$ that satisfy $\boldsymbol{\varrho}_i \cdot \boldsymbol{\varrho}_0 = \cos(i\, 2\pi/N)$, where $\boldsymbol{\varrho}_0$ is the vector gyro radius of the first gyro orbit that was found. Adding this equation to the two equation that define the shape of the gyro ring in velocity space, one gets the nonlinear system for the ith gyro orbit $(i = 1, \ldots, N-1)$

$$\left\{ \begin{array}{l} F_1(v_x, v_y, v_z) = B_x v_x + B_y v_y + B_z v_z - B v_\parallel = 0 \\ F_2(v_x, v_y, v_z) = v_x^2 + v_y^2 + v_z^2 - v^2 = 0 \\ F_3(v_x, v_y, v_z) = \varrho_x^0 v_y B_z - \varrho_x^0 v_z B_y + \varrho_y^0 v_z B_x - \varrho_y^0 v_x B_z + \varrho_z^0 v_x B_y - \varrho_z^0 v_y B_x + \\ \ \varrho^0 \sqrt{(v_y * B_z - v_z * B_y)^2 + (v_z * B_x - v_x * B_z)^2 + (v_x * B_y - v_y * B_x)^2}\, \cos\left(i\, 2\pi/N\right) = 0 \end{array} \right.$$

This system can also be solved with Newton iteration. As before, we use an approximate analytic Jacobian where the inhomogeneity of the magnetic field is not taken into account. In principle, the 3×3 Jacobian could be inverted using a simpler method, but we again use SVD because of its nice robustness. For the initial guess of the Cartesian velocity coordinates of each orbit i in the ensemble, we evolve gyro orbit $i-1$ for $1/N$ gyro period. Convergence is then typically reached in only two iterations. The spatial coordinates are then calculated just like for the first gyro orbit $(i = 0)$.

Gyro-orbit creation and evolution is thus now working in DELTA5D. An example of DELTA5D gyro orbits is shown in Fig. 8.

Task 3: Demonstrate ability to model realistic ITER wall

This task was to determine a good way to load a CAD model of the ITER wall into DELTA5D and which algorithm to use to determine if an alpha particle intersects the wall during a given time step. However, the task was more interesting than expected so we got carried away and also did an implementation of a wall library that does both. Our version of DELTA5D links in this wall library, and is already able to do inefficient alpha wall-load simulations, but with a realistic ITER wall. The inefficiency is due to the fact that the Phase I version of the wall library check all particles against all the triangles defining the surface of the wall, several times per gyro period. An obvious Phase II improvement of the wall library will be to implement a rapid search algorithm that figures out which subsets of particles and triangles to cross check for intersections, and discard the rest.

Overview of particle-wall-intersection task We did discuss the idea of calculating intersections between the true gyro orbit and the wall in the Phase I proposal. Wisely enough, we next said "We will also investigate if it is possible to approximate the nonlinear solve with a number of linear solves by representing the orbit as a number of straight lines". After further thought, we did indeed decide to take the latter, more feasible approach.

To resolve the gyro motion, one needs to take at least half a dozen or so time steps per gyro period. Connecting these points along the gyro orbit with straight line segments, one gets an approximation that does not deviate more than a few millimeters from the true gyro orbit. Even with the most realistic CAD model of the ITER wall, even the smallest details will be larger than this. The linear orbit approximation will therefore be more than good enough for the wall-load simulations.

For the wall it is clear that one only wants to deal with the surface of the wall that faces the plasma. It would be inefficient to load parts of the wall that cannot be reached by plasma particles. One should therefore first mesh the plasma-facing surface of the CAD model. By using flat polygons, the solve for an intersection becomes a linear solve, a big enough advantage to justify possible having to use more and smaller flat polygons than what might have been necessary with curved polygons to get a good wall-surface description.

A secondary goal for this task is to make the code and methods we develop for realistic wall modeling available to the wider fusion community. We therefore only want to use widely and freely available software tools. We found a suitable meshing tools, GMSH [8]. It is conceivable that we in the future might incorporate a surface mesher into the wall library, but at least for now we are running the mesher as a stand-alone application in a pre-processing step. GMSH loads the more open CAD formats (for example STEP, IGES, STL), but unfortunately no open-source software can load the highly proprietary CATIA files that ITER has chosen as their standard.

GMSH produces an output file in the MSH format, which is basically a set of triangles. The wall library can load MSH files, as well as ASCOT wall files. The ASCOT wall files are handcrafted sets of triangles and quadrangles and are more coarse grained than the CAD files used at ITER.

The wall library was implemented in C for maximal callability from all the languages used for fusion-simulation codes (Fortran, C, C++, Python). It provides a very simple interface to the calling code. Given the Cartesian coordinates for two points, it will determine if the line segment between them intersects any of the triangles representing the wall surface, and if so, where. As used from DELTA5D, the two points will be the end points of a gyro-orbit segment. However, we can see many other uses of the wall library for fusion applications. We will propose a Phase II task for an edge ray-tracing code built on the wall library and used within FACETS to study plasma heating by wall modes during EC and LH heating.

Wall model We only managed to get an incomplete CAD model of the ITER wall, consisting of only a small portion of the wall, see Fig. 9. The partial wall was from a CAD file in the CATIA format, which is exclusively used at ITER. The CATIA format is highly proprietary and cannot even be read without a commercial software license. Eventhough incomplete, this wall model still proved useful. Since we cannot directly read CATIA files, we were given the same (partial) wall model converted into three different CAD formats (STEP, IGES and STL). The mesher GMSH was able to open all the converted wall files, but only the STL file format (which is basically a list of triangles and is commonly used in stereolitography) was loaded flawlessly. The files in the other two formats (STEP and IGES) would require further processing or "CAD cleaning" before being used in wall-load simulations.

Since we could not get access to a complete ITER CAD file, we asked the ASCOT team for one of their ITER wall files, and kindly and promptly received it (see Fig. 10). The ASCOT file has lower resolution than a real CAD file. To compensate for this in our tests, we used GMSH to refine the mesh in the MSH file converted from the ASCOT file format by our wall library. Fig. 11 shows a wire-frame figure of this high-resolution wall model. The remeshing can of course not recreate the details of a real CAD file, but is merely intended to produce a data set (the set of triangles defining the wall) of realistic size.

Intersection algorithms Even with the wall surface approximated by a set of triangles and with straight-line orbit segments, finding the intersection is still a non-trivial numerical problem. Two large data sets (particles and triangles, respectively) must be cross-checked every time step. Grazing incidence is typical for gyro orbits and we thus need to find the intersection between an almost parallel line and plane.

There are several approaches for finding the intersection between a line segment and a triangle. The fastest method to determine if an intersection occurs is to solve a linear system for three parameters (r, s, t), where t is normalized distance along the line segment and r and s are normalized positions along two sides of the triangle, known as barycentric coordinates.

We call the three vertices of a triangle $\mathbf{x}_1 = (x_1, y_1, z_1)$, $\mathbf{x}_2 = (x_2, y_2, z_2)$ and $\mathbf{x}_3 = (x_3, y_3, z_3)$, respectively. The ends of the line segment, approximating the particle orbit from time t to $t + \Delta t$, are called $\mathbf{x}_4 = (x_4, y_4, z_4)$ and $\mathbf{x}_5 = (x_5, y_5, z_5)$, respectively. To determine intersection, one can solve the linear system

$$\begin{pmatrix} x_2 - x_1 & x_3 - x_1 & x_4 - x_5 \\ y_2 - y_1 & y_3 - y_1 & y_4 - y_5 \\ z_2 - z_1 & z_3 - z_1 & z_4 - z_5 \end{pmatrix} \begin{pmatrix} r \\ s \\ t \end{pmatrix} = \begin{pmatrix} x_4 - x_1 \\ y_4 - y_1 \\ z_4 - z_1 \end{pmatrix} \tag{1}$$

Then, one checks if $0 \leq r, s, t \leq 1$ and $0 \leq r + s \leq 1$. If these inequalities are satisfied, the line segment has an intersection with the triangle.

The 3×3 intersection matrix can be very ill-conditioned, and we tried several methods for solving it, analyzing both their robustness and speed.

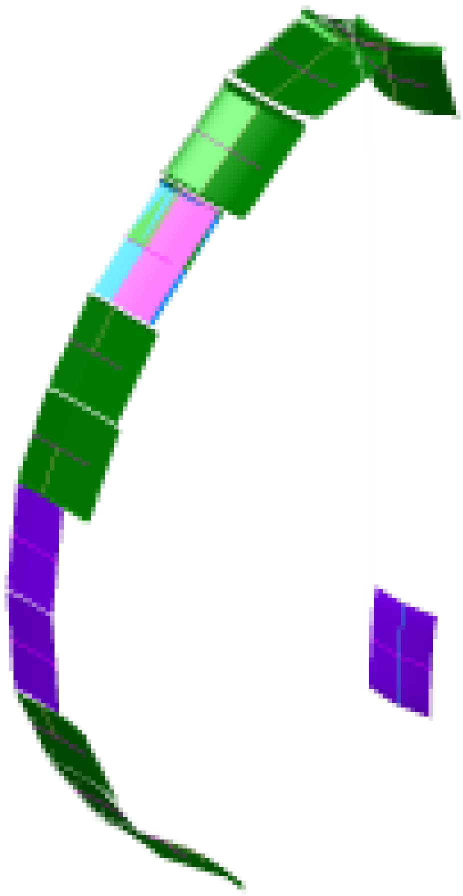


Figure 9: Partial CAD model of ITER wall

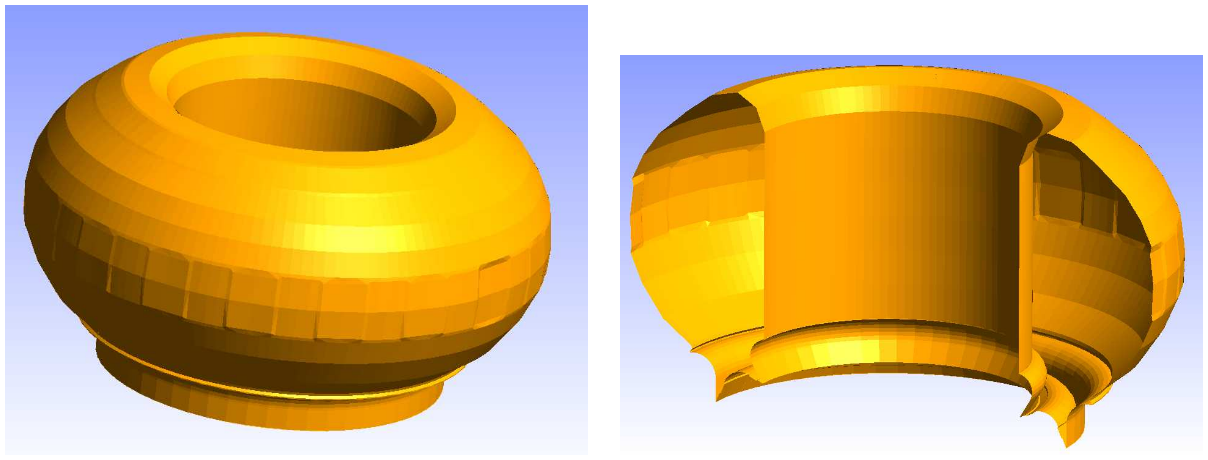


Figure 10: ASCOT wall model, converted to MSH format by our wall library

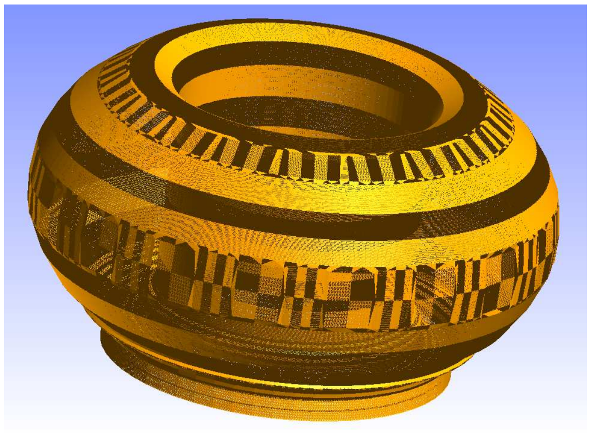


Figure 11: Wire-frame plot of ASCOT wall model with refined mesh

Algorithm	Pairs per second
LU	7.69×10^6
DGELSS	0.14×10^6
DGELSD	0.11×10^6
SVDSOLVE	0.14×10^6
Möller-Trumbore	14.3×10^6

Table 2: Performance of the different intersection algorithms

Standard LU decomposition is fast, but unreliable for large condition numbers. We tried several versions of Singular-Value Decomposition (SVD). The DGELSS LAPACK routine uses SVD to find the minimal-norm solution to a linear system. DGELSD is similar to DGELSS, but applies a Divide and Conquer algorithm. The Divide and Conquer was found to be inefficient for the rank-3 intersection systems. Our own implementation SVDSOLVE, which uses LAPACK DGESVD to compute the Singular Value Decomposition of the linear system matrix, and BLAS routines to compute the minimum-norm solution to the system. We also looked into using deflated decomposition, which could be a faster alternative to SVD large condition numbers, but which cannot handle infinite condition numbers (singular systems). Finally, Möller-Trumbore [9], a clever way to solve the 3×3 intersection system using Cramer's rule.

In Table 2 is listed how many line-segment/triangle pairs each solver can check for intersection in one second. Möller-Trumbore is clearly the fastest, two orders of magnitude faster than SVD. However, only SVD is robust enough to avoid false negatives, which result in particles escaping through the wall.

To speed up the SVD intersection detection, we tried first rejecting line segments not even intersecting the infinite plane on which the triangle lies, inside or outside the triangle. Such line segments can be found without the numerically costly singular-value decomposition. Two algorithms were implemented to quickly identify and discard such line segments. The first of the two algorithms computes the plane equation coefficients, in the form $Ax+By+Cz+D=0$. Given the three triangle corners $\mathbf{x}_1 = (x_1, y_1, z_1)$, $\mathbf{x}_2 = (x_2, y_2, z_2)$ and $\mathbf{x}_3 = (x_3, y_3, z_3)$, the coefficients are given by

$$A = \begin{vmatrix} 1 & y_1 & z_1 \\ 1 & y_2 & z_2 \\ 1 & y_3 & z_3 \end{vmatrix}, \; B = \begin{vmatrix} x_1 & 1 & z_1 \\ x_2 & 1 & z_2 \\ x_3 & 1 & z_3 \end{vmatrix}, \; C = \begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix}, \; D = -\begin{vmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ x_3 & y_3 & z_3 \end{vmatrix} .$$

The plane equation is then evaluated for the line-segment ends $\mathbf{x}_4 = (x_4, y_4, z_4)$ and $\mathbf{x}_5 = (x_5, y_5, z_5)$. If the signs of the results differ for the two line-segment end points, then the line segment does intersect the triangle plane. In such a case, one still then needs to use SVD to solve for the two barycentric coordinates to determine if the intersection with the triangle plane was actually inside the triangle.

The second algorithm computes the following two scalars:

$$a = ((\mathbf{x}_2 - \mathbf{x}_1) \times (\mathbf{x}_3 - \mathbf{x}_1)) \cdot (\mathbf{x}_4 - \mathbf{x}_1)$$

$$b = ((\mathbf{x}_2 - \mathbf{x}_1) \times (\mathbf{x}_3 - \mathbf{x}_1)) \cdot (\mathbf{x}_5 - \mathbf{x}_1)$$

If $ab < 0$ then the line segment crosses the triangle plane.

Algorithm	Checks per second
Plane crossing, determinants	14.3×10^6
Plane crossing, cross and dot products	20.0×10^6

Table 3: Performance of the different intersection algorithms

In Table 3 the performance of the algorithms is compared by testing how many plane-crossing checks each can make in one second. The second algorithm thus proved to be faster than the first. Both algorithms could be subject to errors due to finite floating-point-precision arithmetic. However, during our tests with the refined ASCOT model, neither any false positives nor negatives occurred.

We have thus found a workable two-step approach. First do a quick check to reject line segments that do not intersect the triangle plane anywhere.

For the few line segments that pass this first-step filter, use SVD to solve for the two barycentric coordinates r and s, and possibly also for t. We already know that $0 < t < 1$ because the line segment intersected the triangle plane, but to find its exact value, which gives the exact intersection time, we still need to do the linear solve, or equivalent computation. With the values of the barycentric coordinates known, we can check the inequalities $0 \leq r, s \leq 1$ and $0 \leq r + s \leq 1$ and if they are satisfied the intersection with the triangle plane did indeed occur inside the triangle and qualify as a true intersection.

When the line segment approximating an orbit between times t and $t + \Delta t$ is exactly parallel to the plane of a triangle, the 3×3 intersection system is exactly singular and the third singular value from the SVD is exactly zero. This extreme case is illustrated in Fig. 12. To avoid getting false positives in such cases we demand that the smallest singular value is at least `DBL_MIN` times the largest singular value for an intersection to count, where `DBL_MIN` is the smallest representable floating-point value larger than zero. Fig. 13 shows that the intersection detection modified to include this additional check does work as expected. The cost of avoiding false positives is that false negatives can occur when the angle between the line segment and the plane is less than 10^{-307} and 10^{-304} radians, depending on the resolution of the wall model. In practice, the number of orbits impinging on the wall at angles less than this threshold value should be exceedingly small. The default LAPACK SVD solvers give false positives for angles larger than our threshold value by a factor of `DBL_EPSILON` over `DBL_MIN`, where `DBL_EPSILON` is the smallest value x for which $1.0 + x \neq 1.0$. Our tuning of the algorithm has thus resulted in a reduction of the smallest detectable angle by roughly 292 orders of magnitude ($10^{-16}/10^{-308}$).

SVD excels in getting correct answers from even nearly singular systems. Unfortunately SVD cannot get correct answers from incorrect systems. Because both the left-hand side matrix and the right-hand side vector of the intersection linear system are computed by subtracting vector components, severe loss of accuracy can occur when two almost identical numbers are subtracted. The limited precision of double-precision (64-bit) floating-point numbers then does limit the accuracy of the end result. We believe we have found a feasible solution to this problem and are proposing a Phase II task to prove it.

If one can determine if the line between two points intersects the wall of an object, then one can estimate the volume of that object. This is the principle behind Monte-Carlo volume quadrature. Simply as a way to start testing the wall library before the gyro-orbit integration

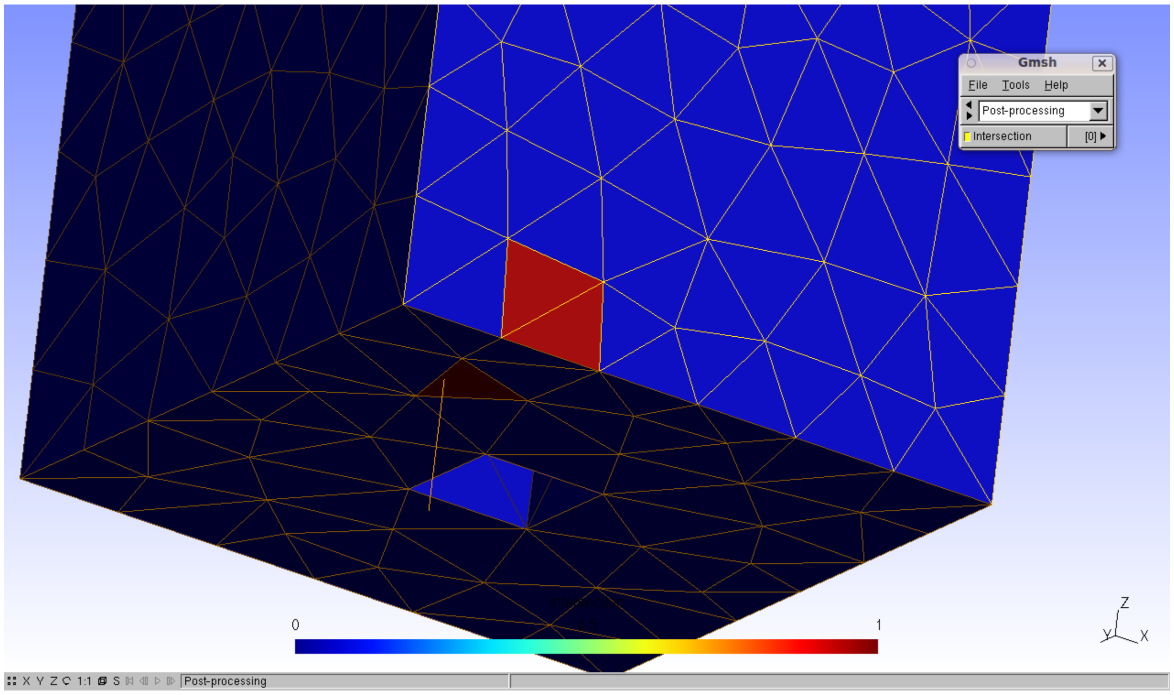


Figure 12: Line segment (yellow line) giving two false positives (the two bright red triangles on side of cube) when line segment is exactly parallel to a triangle.

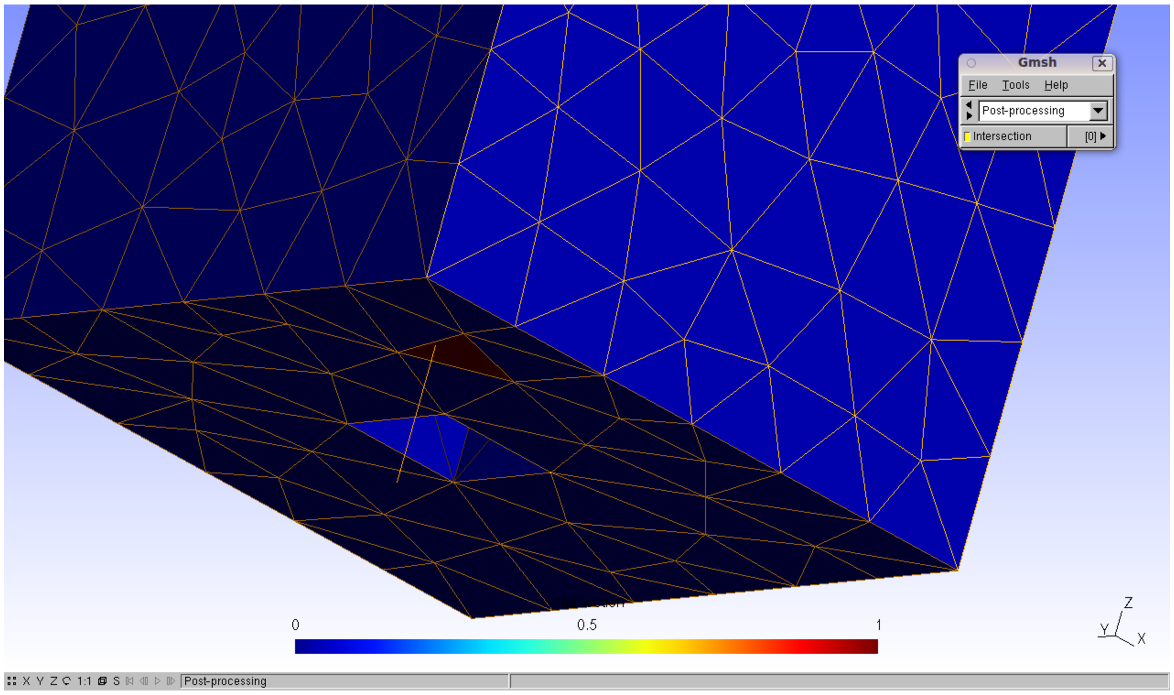


Figure 13: Line segment (yellow line) no longer giving false positives for parallel triangles with modified algorithm that excludes intersections when smallest singular value is virtually zero

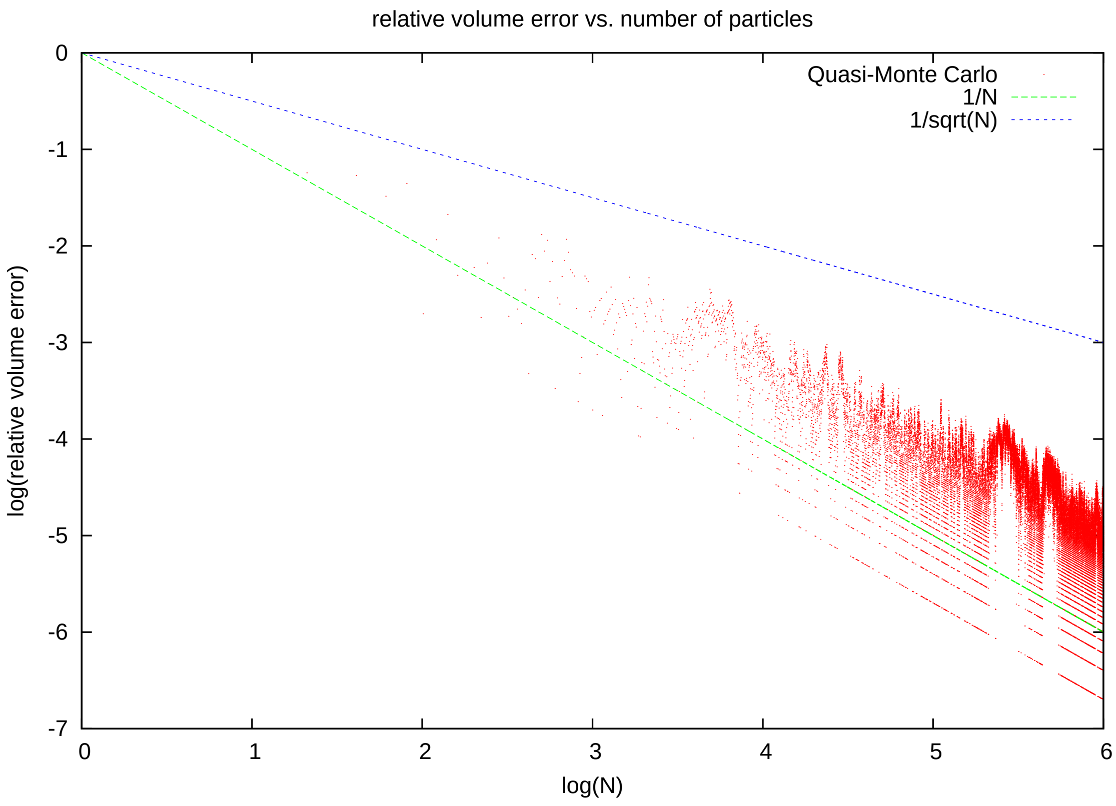


Figure 14: Statistical error vs. number of samples for Monte Carlo quadrature using quasi-random numbers

was working, we implemented such Monte-Carlo quadrature to compute the volume of the ITER vacuum vessel as modeled by the ASCOT wall file. The algorithm works as follows. First define a box with known volume that encloses the object. Then pick a point anywhere on the surface of the enclosing box and a second point anywhere inside it. If the line between these two points intersects the wall of the object an odd number of times, then the second point was inside the object. By sampling a large number of points inside the enclosing box, the ratio of volumes of the object and the enclosing box can be estimated by the fraction of points inside the object. Instead of using real or pseudo random numbers to sample points, we used quasi-random numbers (QRNs). A QRN sequence is designed to sample a volume in an optimal way. With a scrambled Halton sequence, which generates a low-discrepancy distribution, the sampling error for N samples is reduced from the the usual $\mathcal{O}(N^{-1/2})$ to more like $\mathcal{O}(N^{-1})$, as shown in Fig. 14. For the ASCOT wall file, the volume of the ITER vacuum vessel was found to be 1050.5 m^3.

We also looked into a possible GPU implementation of Möller-Trumbore, that is an implementation that can execute most of the computations on the graphics processor on a video card instead of on the CPU. Fortunately GPUs have recently gotten much better ability to handle `if` statements. In fact the relatively small amount of computation in Möller-Trumbore to compute the values of r, s and t makes the algorithm well-suited for predication, the most efficient way to handle conditional code on modern CPUs [10]. Further literature search did indeed find several

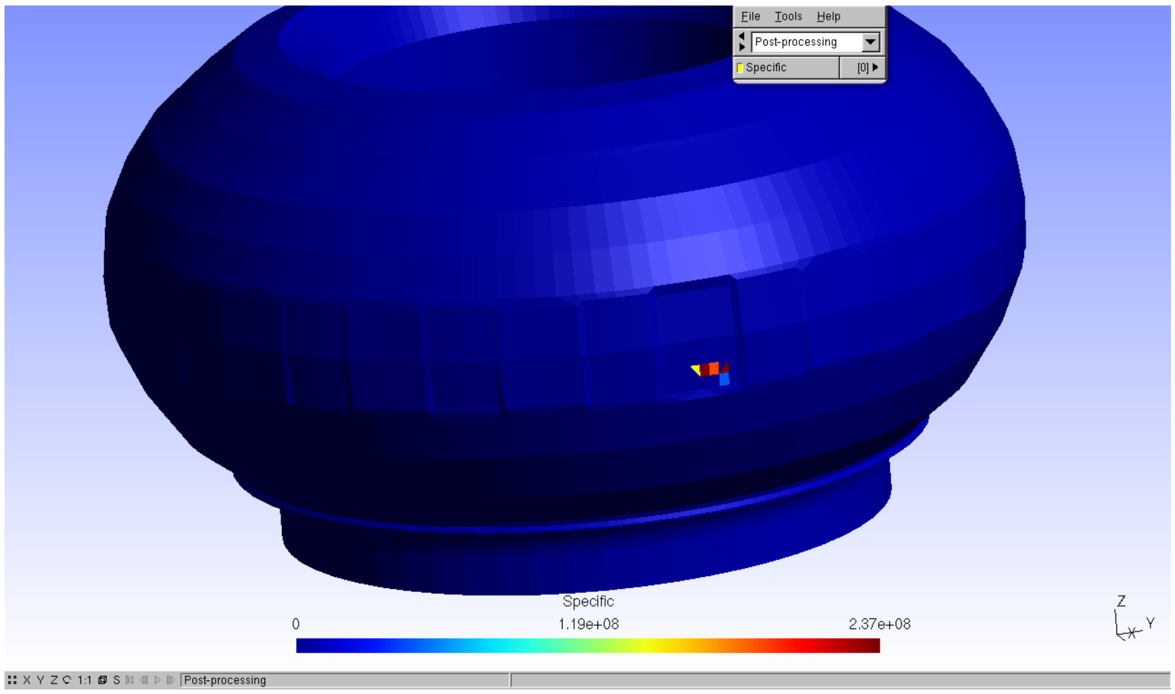


Figure 15: First simulation result for particle wall load with updated version of DELTA5D

examples of efficient GPU implementations of Möller-Trumbore.

First alpha wall-load simulation Finally, we did manage to do a first alpha wall-load simulation with the updated version of DELTA5D. The ASCOT wall model was loaded via the wall library, drift orbits were split as they got close to the wall and the intersections between the evolving gyro orbits and the wall were computed each sub time step by calls to the wall library. Only a handful of gyro orbits were followed and the particle, not thermal, wall load was computed. The results is shown in Fig. 15.

Task 4: Determine efficiency of noise-reduction techniques

The problem with conventional Monte Carlo is that the use of random numbers make it impossible to aim the test particles at the points one wants to know the solution. Therefore binning has to be used, which has an intrinsic conflict between resolution and statistical noise.

Retrograde Monte Carlo does not need binning. It launches the test particles from the point(s) where the solution is wanted, and walks them backward in time. When the initial time is reached, the initial condition is sampled and the solution at the final time can be computed by averaging the sampled initial conditions. This technique works well when the initial condition is spread out in phase space. Unfortunately it is not for fusion-born alpha particles which have an initial distribution function that is a delta function in the energy coordinate.

Fortunately the ensemble averaging over gyro orbits uniformly distributed over gyro phase does work. We have tentatively seen a reduced level of statistical noise both for a fixed number of drift orbits and for fixed run time.

References

[1] D. A. Spong, J. Carlsson, D. B. Batchelor, L. A. Berry, S. P. Hirshman, and J. F. Lyon. Heating, Energetic Particle Confinement, and Transport in Compact Stellarators. *Bull. Am. Phys. Soc.*, 44:215, 1999.

[2] S. Cohen and A. Hindmarsh. CVODE, a stiff/nonstiff ODE solver in c. *Computers in Physics*, 10(2):138–143, 1996.

[3] John R. Cary. General symplectic integration algorithms through fourth order. *Bulletin of the American Physical Society*, 34:1927, 1989.

[4] James A. Rome. Orbit topology in conventional stellarators in the presence of electric fields. *Nuclear Fusion*, 35(2):195–206, 1995.

[5] `https://computation.llnl.gov/casc/sundials/main.html`.

[6] Peter N. Brown, George D. Byrne, and Alan C. Hindmarsh. VODE: A Variable-Coefficient ODE Solver. *SIAM Journal on Scientific and Statistical Computing*, 10(5):1038–1051, 1989.

[7] A.C. Hindmarsh. ODEPACK, A Systematized Collection of ODE Solvers. In R.S. et al. Stepleman, editor, *Scientific Computing: Applications of Mathematics and Computing to the Physical Sciences*, volume 1, pages 55–64, Amsterdam, Netherlands; New York, U.S.A., 1983. North-Holland.

[8] `http://geuz.org/gmsh/`.

[9] Tomas Möller and Ben Trumbore. Fast, minimum storage ray/triangle intersection. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses*, page 7, New York, NY, USA, 2005. ACM.

[10] `http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter34.html`.