LA-UR-17-27965
Approved for public release; distribution is unlimited.

Title: Fragment Impact Toolkit (FIT)

Author(s): Shevitz, Daniel Wolf
Key, Brian P.
Garcia, Daniel B.

Intended for: Report

Issued: 2017-09-05

Fragment Impact Toolkit (FIT)

Daniel Shevitz, Brian Key, Daniel Garcia

Los Alamos National Laboratory: P.O. Box 1663, Los Alamos, NM 87545

Introduction

The Fragment Impact Toolkit (FIT) is a software package used for probabilistic consequence evaluation of fragmenting sources. The typical use case for FIT is to simulate an exploding shell and evaluate the consequence on nearby objects. FIT is written in the programming language Python and is designed as a collection of interacting software modules. Each module has a function that interacts with the other modules to produce desired results.

Design Philosophy

The design philosophy of FIT is that there are enormous uncertainties in fragmentation problems. Numerically generating fragmentations of objects from first principles is computationally expensive and subject to variables that are unknowable such as the exact microstructure of the object or defect locations. The effective result is that from a realistic perspective the fragmentation itself may as well be considered random, albeit with certain known statistical properties, such as the mean fragment size, the fragment distribution, etc.

FIT uses a Monte Carlo approach to simulate fragmentation problems. By avoiding first principle simulations FIT can run orders of magnitude faster. FIT uses empirical inputs such as fragment size distribution and velocity to generate many fragmentations and evaluate their consequences. In essence, FIT uses either real data or first principles simulation as initial conditions for running many simulated case breakups.

FIT is written with a flexible class structure that allows the user to easily customize all aspects of the simulation. A simulation typically has the following stages:

1. source fragmentation,
2. assigning initial conditions to the fragments,
3. transporting them through space,
4. defining target geometries,
5. checking for impacts of the fragments on the targets,
6. screening those impacts for things such as sufficient kinetic energy or proximity in space and time,
7. post processing the results to compute images, movies, or graphs.

Example FIT simulations and results are shown in Figures 1 and 2.

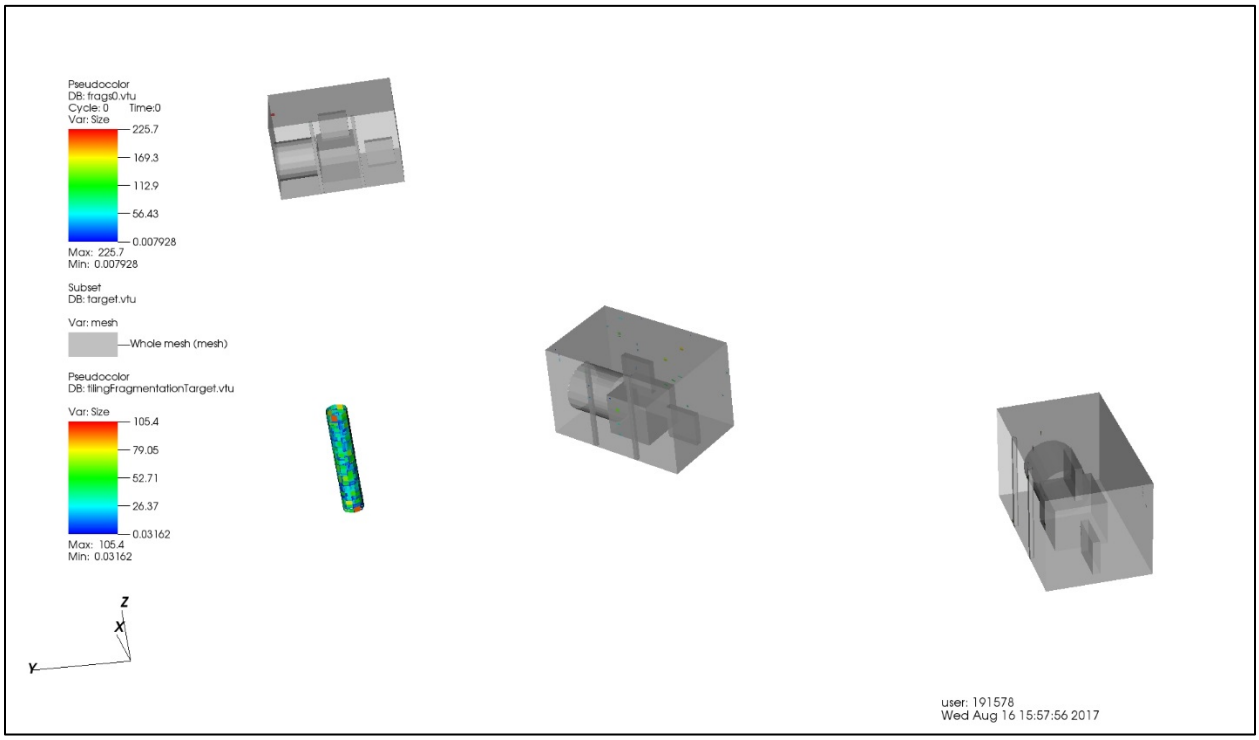

Figure 1. Example FIT Simulation. Source fragmentation (cylinder) and targets (boxes)

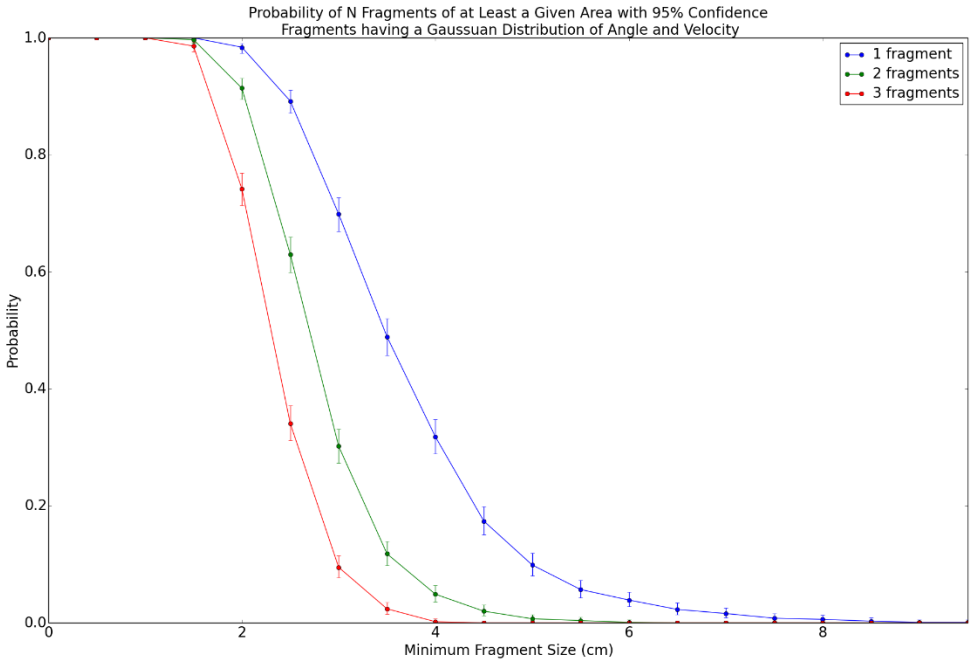

Figure 2. Example Post Processed Results

Users are not locked into any particular physics. Running a FIT simulation consists of writing a driver script that does whatever the user requires. FIT consists of the following primary classes: Node, Tree, Fragment, Fragmentation, Shape, and Simulation. What each of these classes does and how they inter-operate will be the subjects of the following sections.

Classes

Node Class:

A Node is an abstract entity that corresponds one to one with fragments. In other words, each fragment has a corresponding Node in a binary tree. The use of binary trees will be explained in the next section. Nodes (unless they are leaf nodes) have two children which are the sub-fragments that this fragment splits into. Each Node has a weight which will correspond to its mass, and a "fulcrum" which is used to describe how much mass should be in either of the child fragments. The fulcrum will also be explained further in the next section.

Tree Class:

Just as with Nodes and Fragments, Trees and Fragmentations correspond to abstract and concrete representations of the fragmentation. Trees describe the relationships between the nodes, and fragmentations correspond to physical positioning of the fragments. One of the primary realizations of the FIT code is that fragmentations can be described both conceptually and mathematically by binary trees. The conceptual idea is that two fragments can be combined into a "super fragment". Those super fragments can be combined, and that process can be continued until all we are left with is the original shape. This ignores stretching of the fragment and shape distortions which are obviously present, but when we consider damage assessment we really only care about the mass and velocity, i.e. its kinetic energy. The shape matters less to us. In this way, the original shell can be considered tiled by the fragments.

This process as described presently can always be constructed for a thin shell. While the binary tree uniquely determines a fragmentation it should be noted that the converse is not true. There are many, many ways to pairwise combine fragments in a hierarchy.

There are so many uncertainties in fragmentation that FIT takes a high level approach and assumes that the shapes of the fragments do not matter. What matters is the fragment size distribution and velocities. To simplify the fragment generation process, FIT creates rectangular fragments because they are very easy to tile flat shapes. The actual fragmentation is always a rectangular patch which could in principle be a small tangent plane to a complex shape, but in practice up to this point, we have always just wrapped the rectangle around a cylinder to represent the source. The notion of a binary tree can always be applied to the fragmentation of a thin shell. It would be useful to be able to map the fragmented rectangle onto more complex shapes, but unfortunately this is impossible, because more

complex mathematical shapes cannot be mapped in a way that preserves area, hence the fragment size distribution would get distorted.

Trees start with a fragment size distribution. Typically this distribution is usually specified in terms of mass. Converting from the fragmented rectangle described in the preceding paragraph and mass is straightforward using the relationship the thickness times area times density equals mass. To acknowledge that the fragment distribution could be specified in terms of either area or mass, we use the neutral term “weight” to describe the distribution. Fragment weights are sampled from the specified distribution until the weight exceeds the desired total. For instance, in the typical use case, fragment masses are sampled until the total mass exceeds the desired mass of the source. The last fragment weight is then adjusted to exactly total the desired weight. This last step is a little ad-hoc, but presents no systematic bias to the result especially considering that most fragmentations have hundreds or thousands of fragments, each a small fraction of the total.

The question becomes where to put the fragment in the original rectangle. This is where the “fulcrum” comes into play. The obvious thing to do would be to flip a coin at each node in the tree until we come to a new leaf. The problem with this method is that due to the Central Limit Theorem [1], as we sample many fragments on average half will go to the left and half will go to the right and this keeps happening recursively. This leads to a biasing of the fragmentation so that there is split down the middle, and then each of the halves is again split down the middle, and so on. As the fragments travel through space this leads to artifacts including artificial gaps in the fragment field. To prevent this from happening, FIT uses a slightly different approach. At each node we sample a distribution, typically a uniform distribution between two numbers less than one, for example [.3, .7]. This is desired fraction of the weight to keep on each side of the division of the two sub-fragments. A conceptual way of thinking about the fulcrum is on a child’s teeter-totter we always add the weight to the high side, which tends to improve the balance. By making the fulcrum a random number, we prevent systematic or reproducible artifacts. The idea is that for each fragment weight we progress down the tree, at each node deciding to add the fragment to the left or right depending on whether the cumulative weight below the current fragment is above or below the desired fulcrum. In aggregate we end up with a nearly balanced tree (for a visual image, think of a child’s mobile) where the balance points are determined by the fulcrum.

Fragment Class:

The relationship between a node and a fragment is simple and straightforward. The Fragment class appends information to the Node class that corresponds to the real fragment in three dimensional space. In addition to the weight (typically mass), the fragment has a birth position, an initial velocity, and an initial time when it starts moving.

Fragmentation Class:

Just as the Fragment class turns an abstract Node into a physical entity, the Fragmentation class turns the Tree class into a physical manifestation. The Fragmentation class has methods for iterating over the fragments and performing various operations including assigning the positions, transporting the fragments along their ballistic trajectories, intersecting them with targets, persisting the data, etc. There

is very little implemented in the Fragmentation class. Mostly the Fragmentation class keeps track of global constants, such as the size of the initial shell. In addition, the Fragmentation class keeps track of all the fragments and iterates over them.

Shape Class:

Even though a single simulation may use two cylinders, one for the source and one for the target, the representations of source and target are totally different. The source has been extensively discussed and consists of a collection of fragments. The various Shape classes and subclasses (Shape, PlanarPatch, ConicSectionPatch, Ellipsoid, etc.) are used to represent targets. The targets are described analytically. The reason for this is that with an analytic representation for the targets, the intersections of fragments and the target in its entirety can likewise be computed analytically at a dramatic savings in computational cost.

At this point we need to talk about what it means for a fragment to hit a target shape. In FIT, we typically (although it's not strictly required, but much more efficient) assume the fragments travel in straight lines starting from their center of mass and moving in the direction of their initial velocity. We neglect the force of gravity, tumbling, air resistance, etc. In other words, we assume that the target is sufficiently close to the source that the flight of the fragment is ballistic. The reason for this assumption is that this allows an analytic solution of the intersection. With these assumptions, if we consider the motion of the fragment through time, the fragment's trajectory is assumed to be a line parametrized by the flight time. A fragment impact is considered to be an intersection of the fragment trajectory line with the target shape. In an intersection, we need to find the intersection time and make sure that it is positive. A negative time would correspond to going backward in time as would happen if the target were on the opposite side of the source. These negative time solutions need to be discarded.

Typically realistic targets are complicated assemblies of multiple shapes. The Shape class is actually a container for other Shapes. The idea is that we build up more complex shapes out of simpler shapes. Each shape keeps track of where it is in space and any constraints that must be satisfied for a fragment to hit. For example if the shape is half a sphere, then the shape needs to know which half is excluded.

Shapes are created as the solution to equations. There are two types of primitive shapes: PlanarPatch and ConicSectionPatch. Mathematically, a plane can be described as the set of points satisfying:

$$N^T x = c,$$

where N is the normal vector to the plane and c is how far up normal vector the plane sits. In general every line intersects every plane somewhere. This is no longer true if the planar patch has finite size. Then we need to make sure that the intersection point satisfies the constraint of actually hitting within the finite domain. All this can be done analytically. The trajectory of the fragment is represented by the equation:

$$x = x_0 + v_0 * t,$$

where x_0 is the birth position of the fragment and v_0 is the birth velocity of the fragment and t is time. Therefore the intersection of the fragment with the plane can be written as the solution (in t) to the simple equation:

$$N^T(x_0 + v_0 * t) = c,$$

which is a simple linear equation in t. Then we need to check whether the point $x_0 + v_0 * t$ satisfies the constraints of the finite size.

When working with composite shapes, it is frequently easier to define the target in a convenient reference frame. For example, define the plane as having a normal vector in the z-direction. We can then rotate and translate the plane to wherever we need. The convention we use is that these transformations are parametrized by the formula:

$$x \to R_z(\varphi)R_x(\theta)x + x_0.$$

In words, we first rotate around the x-axis by θ degrees. Then we rotate around the z-axis by φ degrees and finally we translate by x_0. This may seem like an over simplification but for objects with axial symmetry, this is in fact maximally general. Transformations of this form can be composed to form another transformation of the same class. The shapes defined in FIT keep track of their composite transformations. The preceding discussion can be slightly modified to account for the added complexity of the transformation and the intersection can still be computed quickly as the solution to a linear equation in t.

The second type of primitive shape is encapsulated in the ConicSectionPatch class. This class is quadratic in x and is represented by solutions to the equation:

$$x^T Q\, x = c,$$

where Q is a quadratic form (a symmetric 3x3 matrix). This generalization gives a surprisingly rich set of additional shapes depending on the particular Q we chose. The most useful shapes we can generate by varying Q are cylinders, spheres, and ellipsoids. The discussion of fragment impact proceeds similarly to that for the PlanarPatch except that the intersection:

$$(x_0 + v_0 * t)^T Q\, (x_0 + v_0 * t) = c$$

is now a quadratic equation in t, which can also be solved analytically. To determine if an intersection exists, one only needs to compute the discriminant of the resulting quadratic equation and see if it is greater than zero. The discussion of constraints and transformations is identical to the PlanarPatch.

Given these two primitive shapes, we can approximate a surprisingly wide range or shapes and scenes as shown in Figure 1. Shapes are composed of other shapes. We can use a single planar patch to represent a witness plate, or a set of patches to represent a system of witness plates. We can combine a cylinder with a circular bottom and ellipsoidal cap to represent a piece of artillery. Combining a cylinder with two ellipsoidal caps and trapezoids for wings and tail assemblies gives us an airplane, shown in Figure 3.

These shapes can themselves be combined to create more complicated scenes or theaters. The possibilities are limitless.

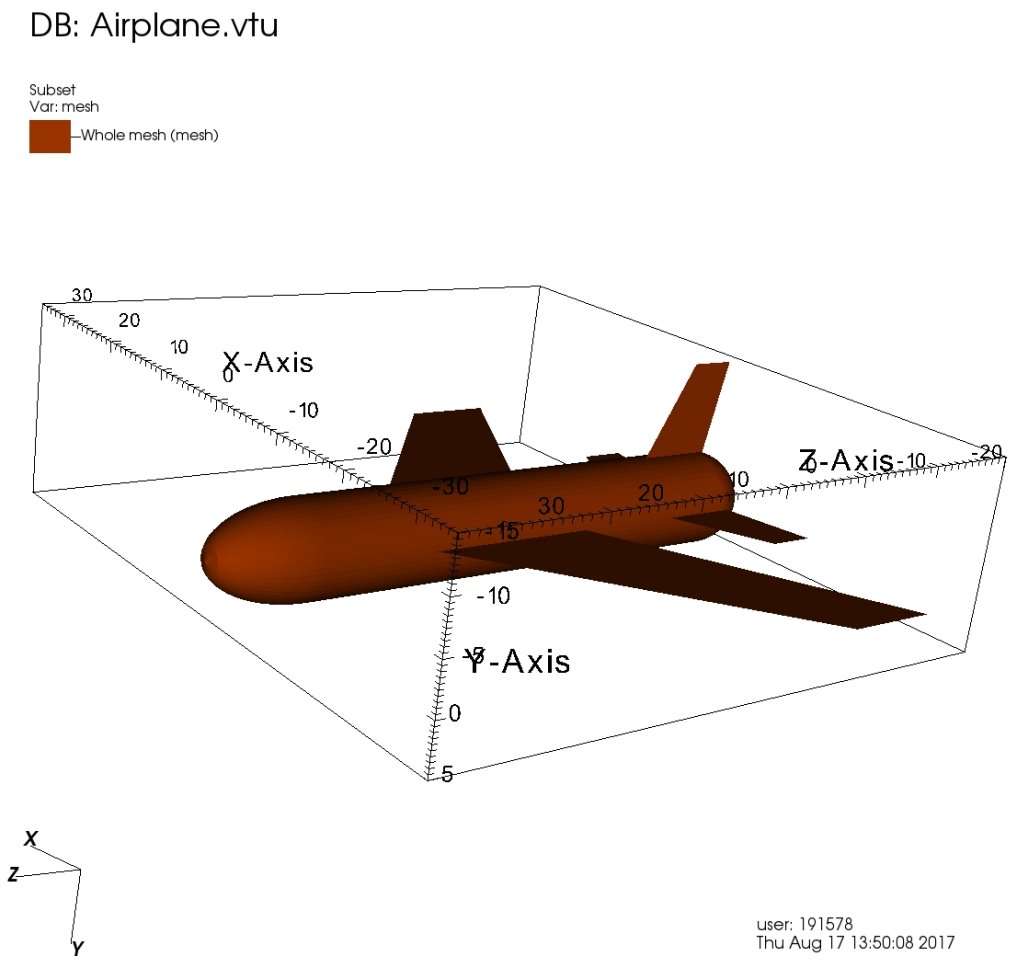

Figure 3. Airplane made from compound primitive shapes

A final note on complex targets is that a fragment could potentially impact multiple patches and we need a way to disambiguate these to determine the actual impact. The final determinant is simply the physical impact corresponds to the lowest positive impact time. In other words, the first impact is the actual impact.

Simulation Class:

The Simulation class is a very simple wrapper that orchestrates operations that are typically done together, such as fragment generation, assignment of positions and velocities, setting up the target geometry, computing impacts, and finally post processing.

One of the advantages of FIT is that post processing is implemented by scripting and therefore can be made completely customizable. The simplest post processing is to run a single simulation and look at the results. FIT has built in routines to convert fragmentations into VTK files which is a standard graphics file format that allows viewing in software such as Paraview [2] and VisIT [3]. Viewing the fragmentation allows the user to see the initial fragmentation of the source, which fragments impact the target, viewing them at either the initial positions or where they impact the target, see Figure 4.

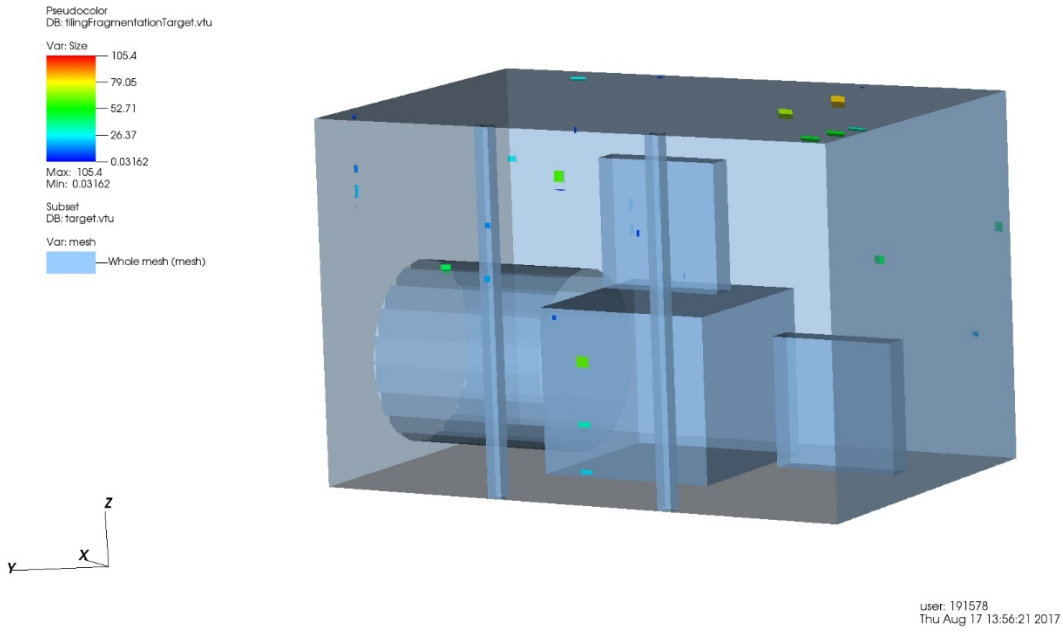

Figure 4. Fragment Impacts on Target

Similarly movies can be made of the explosion of the fragment field moving away from the source as shown in Figure 5. All of these types of results help provide a sanity check on any statistics that might ultimately be generated.

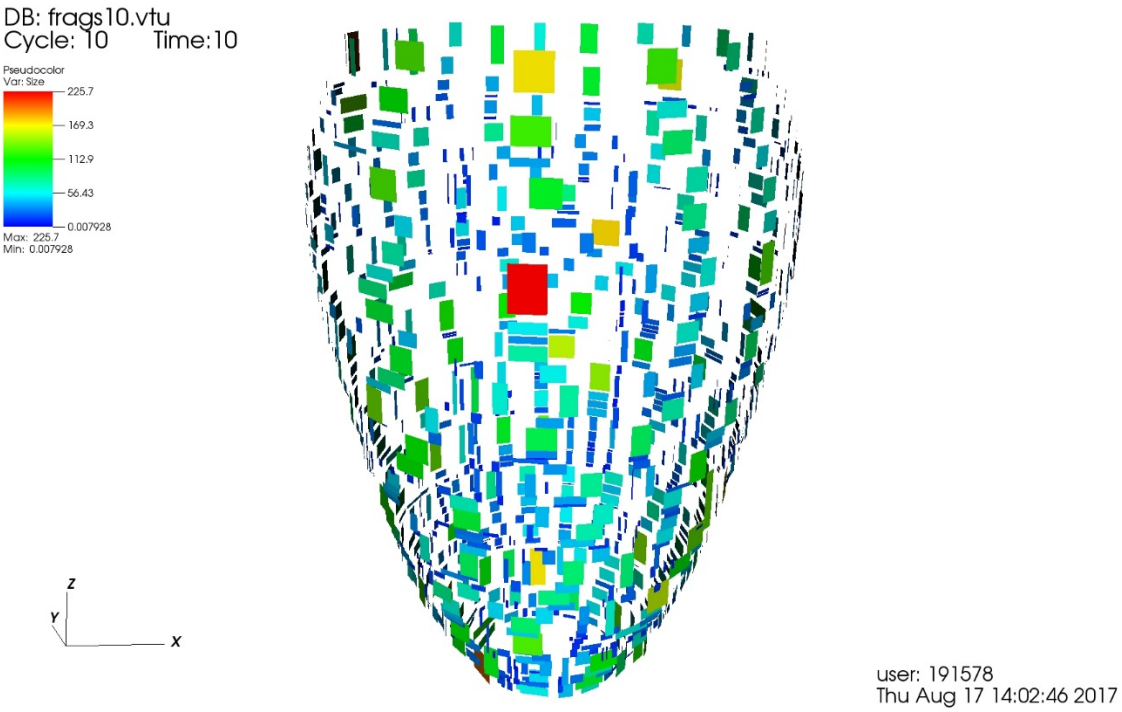

Figure 5. Moving fragment field and fragmentation

Because of intrinsic unknowns, fragmentation is considered a random process. While any single predicted fragmentation is possible, the likelihood of that fragmentation will be essentially zero. To counteract the randomness, FIT is typically used in a Monte Carlo fashion. Many fragmentations are run to simulate shell fragmentation. Depending on the complexity of the problem, many can mean hundreds, thousands, or potentially millions of simulations.

Assessing probabilities is usually a matter of counting outcomes. We estimate the probability of an event by simple counting. The probability estimate is defined as the number of times an event occurs divided by the total number of trials. The event can be defined by the user. The simplest event to consider is: did the target sustain an impact or not? This probability would estimate the question of when the target survived unscathed or not.

One aspect of FIT not discussed up to this point is the concept of filtering. Filtering is when we remove a fragment from an analysis. There are many reasons why we might want to filter a fragment. We will list three here. The first is the case of shielding. A fragment may be stopped by another element of the scene before impacting the target. This is actually implemented as a compound target, but only impacts on the desired patch are counted. A second type of screening is even if the fragment impacts the target, it may not have sufficient size or kinetic energy to do any damage. Conceptually, if a piece of dust hits the target it will not do any damage. The user can control the threshold for considering an impact sufficiently large to count. The third and most subtle form of filtering implemented at present is implemented when the target is an explosive. It is of interest to count how many impacts can ignite the

explosives in the target. First, the fragments need to have sufficiently large kinetic energy to light the explosive but there is a second requirement. The first impact that ignites the target creates a detonation wave in the explosive. If the second fragment hits the target but is behind the detonation wave of the first impact, then it is of no consequence because the explosive is already lit at the point of impact, hence such impacts can be ignored.

Conclusion

FIT is a set of interacting modules implemented as Python classes. These modules are used to represent different physics in consequence evaluation. Modules include case breakup with a known fragment size distribution, assignment of initial conditions including fragment mass, velocity, and time of flight. FIT has modules for implementing the flight of the fragments including filtering which removes fragments for reasons such as hitting shields, creating complex target geometries composed of patches of planes and 3D generalizations of conic sections. Simulations can then be defined as scripts to view the fragmentations and consequences in a graphics viewer or to compute probabilities to inform policy makers. Acting together the classes of FIT can simulate many different kinds of impact assessment problems and rapidly give answers that would otherwise be impossible.

References

[1] Wikipedia, "Central Limit Theorem," Wikipedia, 2017. [Online]. Available: https://en.wikipedia.org/wiki/Central_limit_theorem.

[2] Sandia Corporation, Kitware Inc., "ParaView," Kitware Inc., 2017.

[3] Lawrence Livermore National Laboratory, "VisIt: An End-User Tool For Visualizing and Analyzing Very Large Data," Livermore, 2012.