Pyg4ometry: a Python library for the creation of Monte Carlo radiation transport physical geometries

Stewart T. Boogert[a,*], Andrey Abramov[a], Laurence Nevay[a], William Shields[a], Stuart Walker[a]

[a]*John Adams Institute at Royal Holloway, Department of Physics, Royal Holloway, Egham, TW20 0EX, Surrey, UK*

Abstract

Creating and maintaining computer readable geometries for use in Monte Carlo Radiation Transport (MCRT) simulations is an error-prone and time-consuming task. Simulating a system often requires geometry from different sources and modelling environments, including a range of MCRT codes and computer-aided design (CAD) tools. Pyg4ometry is a Python library that enables users to rapidly create, manipulate, display, read and write Geometry Description Markup Language (GDML)-based geometry used in simulations. Pyg4ometry provides importation of CAD files to GDML tessellated solids, conversion of GDML geometry to FLUKA and conversely from FLUKA to GDML. The implementation of Pyg4ometry is explained in detail along with small examples. The paper concludes with a complete example using most of the Pyg4ometry features and a discussion of extensions and future work.

Keywords: Geant; FLUKA; GDML; CAD; STEP; Monte Carlo; Particle; Transport; Geometry;

PROGRAM SUMMARY

Program Title: Pyg4ometry
Licensing provisions: GPLv3
Programming language: Python, C++
External routines/libraries: ANTLR, CGAL, FreeCAD, NumPy, OpenCascade, SymPy, VTK

*Corresponding author.
E-mail address: stewart.boogert@rhul.ac.uk

Nature of problem:
Creating computer readable geometry descriptions for Monte Carlo radiation transport (MCRT) codes is a time-consuming and error-prone task. Typically these geometries are written by the user directly in the file format used by the MCRT code. There are also multiple MCRT codes available and geometry conversion is difficult or impossible to convert between these simulation tools.
Solution method:
Create a Python application programming interface to read and write files of and process the geometry objects and specification language used by Geant4 and FLUKA. Form triangular meshes to represent geometrical objects for both visualisation of the geometry and advanced approximate geometrical algorithms. Triangular mesh process allows the loading and use of STL and CAD/CAM files. Converting from FLUKA to Geant4 requires algorithms to decompose solids to a set of unions of convex solids. Converting from FLUKA to Geant4 requires the replacement of infinite surfaces with finite solids.

1. Introduction

There are numerous different software codes to simulate the passage of particles through material, such radiation transport (RT) programmes include MCNP [1], FLUKA [2, 3], Geant3 [4] and Geant4 [5]. All these codes are based on the Monte Carlo technique but each code either has a particular specialism, simulation methodology or target user community. Monte Carlo RT (MCRT) simulations have diverse uses including shielding calculations for radiological protection, detector performance, medical imaging and therapy, and space radiation environment simulations are some examples. A fundamental requirement of all of the codes is to supply a computer-readable description of the physical three-dimensional geometry that the particles are passing through. The creation of geometry files is typically a very time-consuming activity and the simulation validity and performance is directly dependent on the quality of the geometry. There is no standard geometry format used across MCRT codes, with each code typically using its own unique format. A user will typically not have geometry in FLUKA and Geant4 for example. A geometry system that allows the conversion between files prepared for different codes will enable cross-checks of the physics processes in different particle transport codes. The file formats used for geometry are generally focused on the computational efficiency of a particle tracking task and not ease of use. In addition to the creation of geometry files for RT programs, usually computer-aided design (CAD) files exist for systems which need to be simulated. The fundamental geometric representations in CAD files are usually not amenable to MCRT programs. For these reasons it is advantageous to create a software tool that allows particle transport code users to rapidly develop error-free geometry files, convert between common MCRT geometry formats and load CAD models.

This paper describes a geometry creation and conversion package called Pyg4ometry, written in Python and internally based on the Geant4 application programming interface (API) and the Geometry Description Markup Language (GDML) for file persistency [6]. The main features of Pyg4ometry are a Python scripting API to rapidly design parametrised geometry; conversion to and from FLUKA geometry descriptions; conversion from CAD formats (STEP and IGES) based on FreeCAD [7] and OpenCascade [8]; and powerful geometry visualisation tools based on VTK [9]. The origin of Pyg4ometry was a set of utilities to prepare geometry for an accelerator beamline simulation program based on Geant4 called BDSIM [10]. Accelerator physicists, like specialists in other areas, need a tool to quickly model specialist geometry and the subsequent interaction of the charged particle beam. Pyg4ometry allows the rapid creation and adaptation of geometry. Figure 1 depicts various workflows possible with Pyg4ometry. Pyg4ometry is not an executable software package but a toolkit, a user would typically write a very small Python program to use to use the classes and functions provided by Pyg4ometry. This paper describes version 1.0 of Pyg4ometry, which is freely available as a git repository and via the Python Package Index (PyPi).

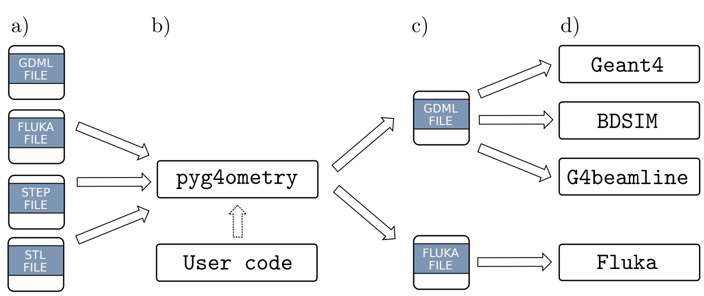

Figure 1: Schematic of Pyg4ometry workflow, showing (a) the different input file formats, (b) Python processing, (c) output file targets and (d) MCRT codes which use the geometry.

There are existing codes that have functionality similar to Pyg4ometry.

ROOT [11], the high-energy physics data analysis framework, can load, display and manipulate GDML-like geometry. Tools exist to convert CAD files to GDML, open source examples are GUIMesh [12], commercial products include ESABASE2 [13] and FASTRAD [14]. There are also tools from the fusion and neutronics community that can convert CAD geometry into formats usable by MCRT codes, with examples including DAGMC [15] and McCAD [16]. In principle CAD software can export shape data to STL (or other similar mesh formats), which can be used by Geant4 [17]. In practice, using a lot of CAD models is difficult if that model is comprised of a large number of parts. Exporting, assigning material to, and placing the STL components into the MCRT code can be a cumbersome task. Almost all modern CAD tools like CATIA, Inventor, SolidWorks have a scripting language to allow users to programatically generate geometry. This scripting functionality is not available with either GDML or FLUKA were users write the geometry files directly. The existent set of software does not provide a complete set of tools to efficiently create complex geometries.

This paper is structured as follows, first a brief radiation transport-focused introduction to computer descriptions of geometry, followed by an explanation of the design and implementation of Pyg4ometry. The sections that follow describe how Pyg4ometry can be used to perform rapid geometric modelling, conversion from FLUKA to GDML, GDML to FLUKA and CAD to GDML. The paper concludes with an example of a composite, complex system consisting of components drawn from all the supported geometry input files formats.

2. Computer descriptions of geometry

Central to a computer-readable geometry is how a solid is defined in three dimensions. There are numerous different ways to describe a geometry, including constructive solid geometry (CSG), boundary representation (BREP) and tessellated polygons, which are described briefly in the following sections. The Geant4 geometry specification is a mixture of geometry modelling techniques and described in detail last.

Constructive solid geometry uses Boolean operations (subtraction, intersection and union), between simple solid shapes (e.g. cube, cylinder, sphere, etc.) or infinite volumes (e.g. a plane-defined infinite half-space) to model complex surfaces which represent a solid. Boolean operations and solids can be combined to form a CSG tree to model complex geometry. FLUKA uses

CSG to model solids, the form of Boolean expression used by FLUKA is not a general CSG tree but a logical expression in disjunctive normal form.

Boundary representation consists of two parts, topology and geometry. Topological elements are faces, edges and vertices and the corresponding geometrical elements are surfaces, curves and points. No current MCRT applications use native file formats employed by CAD systems. The conversion of CAD BREP formats for loading in MCRT applications is typically performed via a tessellated format, although it is possible to decompose BREP descriptions to bounded or infinite mathematical surfaces and subsequently solids as used in CSG descriptions. This type of conversion is complex and error-prone, although recent progress has been made [18].

Solid volumes can be defined using triangular, quadrilateral or tetrahedral meshes. Numerous formats exist to describe meshes, the ubiquitous being STL with more modern examples including PLY and OBJ. For solids with curved faces a tessellated mesh will always give an approximate description. As the mesh deviation distance from the solid decreases the number of polygons increases and with it the memory consumption and execution time of the MCRT simulation.

Geant4 geometry description is a mixture of BREP, CSG and tessellated concepts. Geant4 includes 27 basic solids, although it does not store a sense of topology present in traditional CAD BREP systems. One of the fundamental solids is a tessellated solid which can be used to represent STL or PLY files. Geant4 also provides the ability to perform Boolean operations on these primitive solids. The richest and most flexible geometry description is currently used by Geant4. Not only do solid objects need to be defined but also placed in a world coordinate system. Geant4 has two concepts which facilitate this *logical volumes* and *physical volumes*. A logical volume is a region of space that is defined by an outer solid but also other attributes like material, magnetic field and zero or more *daughter* physical volumes. A physical volume is a placement (or instance) of a logical volume. This design permits large reuse of objects, minimising memory footprint for largely repetitive structures such as detectors that Geant4 was created to simulate.

To exchange geometry descriptions between software packages the Geometry Description Markup Language (GDML) was developed [6]. GDML is an XML-based description of Geant4 geometry. Geant4 and ROOT [19] can read and write GDML and it is commonly used as an exchange format for Geant4 geometries.

3. Pyg4ometry design and layout

Pyg4ometry is a Python package consisting of semi-independent sub-packages. The sub-package `pyg4ometry.geant4` contains all classes for Geant4 detector construction and `pyg4ometry.gdml` provides the functionality for reading and writing GDML files. There are sub-packages for importing and exporting other geometry formats: `pyg4ometry.fluka`, `pyg4ometry.stl` and `pyg4ometery.freecad`. Lastly, the sub-package `pyg4ometry.convert` is used for conversions between formats.

The core of Pyg4ometry consists of Python classes that mimic Geant4 solids, logical volumes, physical volumes, GDML parameters and material classes. The constructors of the Python classes are kept as close to the original Geant4 C++ implementation as possible so that Pyg4ometry users do not have to learn a new API. For example the `G4Box` class in Geant4, has the XML tag `Box` in GDML and is represented by the `Box` class in Pyg4ometry. The Python object initialisers are very similar to their corresponding Geant4 C++ constructors, but the length definitions are those used by GDML. For example, GDML uses full lengths whilst Geant4 uses half lengths. Geometry construction in Python proceeds in a way which is very similar to geometry construction in Geant4. A user relatively familiar with Geant4 should be able to start creating geometry in Pyg4ometry immediately. In the following sections novel or important developments in Pyg4ometry are described.

For each input format available in Pyg4ometry (GDML, STL, FLUKA and STEP) a dedicated *Reader* class is implemented: `gdml.Reader`, `stl.Reader`, `fluka.Reader` and `freecad.Reader`. Each reader constructs the appropriate Geant4 classes and provides a `Registry` instance which can be used or manipulated by the user. Output consists of taking the registry and writing to file with the desired format.

The internal data representation closely follows the structure of GDML. A `Registry` class aggregates Python ordered dictionaries that are used to store the main elements of a GDML file. As a Pyg4ometry user instantiates the geometry the registry is correspondingly updated. When a user is finished with the geometry, the registry can be written to disk as a GDML file. Multiple `Registry` instances can be aggregated to form a composite geometry or volumes can be removed and added.

In GDML symbolic expressions can be used to parametrise solids and their placement. These expressions are evaluated when the GDML is loaded into Geant4. In order to fully replicate the functionality of GDML an expression engine was implemented using ANTLR [20]. The GDML is loaded using standard XML modules and parsed using ANTLR to create an abstract syntax tree (AST). GDML allows for the definition and assignment of variables. GDML expressions are not much more complicated than binary operators $+, -, \times, /$ and common trigonometric and special functions $\sin, \cos, \tan$, etc. The AST terminates on either expressions which evaluate to numbers or GDML variables. Internally, all PYG4OMETRY classes use GDML expressions and not floating-point numbers. Storing internal data as expressions allows for deferred evaluation (or re-evaluation) of solid parameters and placements. This allows a user to update variables whilst defining geometry and the expression engine will update all internal values. An example of GDML expressions is shown in Listing 1.

Listing 1: A simple Python script using PYG4OMETRY to create GDML variables.

```
# Import modules
import pyg4ometry

# Create empty data storage structure
reg = pyg4ometry.geant4.Registry()

# Expressions
v1 = pyg4ometry.gdml.Constant("v1","0",reg)
v2 = pyg4ometry.gdml.Constant("v2","sin(v1+pi)",reg)
```

A powerful feature of Geant4 and hence GDML is the ability to either repeat, divide or parametrise geometry. The class which enables the creation of multiple replicas of a volume in a Cartesian, cylindrical or spherical grid is known as a Replica Volume. A Division Volume breaks a primitive into segments in either Cartesian or cylindrical polar coordinates. A parametrised volume allows for the arbitrary multiple placement of solids where the parameters are allowed to vary for each placement. Another way in GDML to create parametrised solids or volumes is GDML loops, where sections of GDML can be repeated with varying parameters based on the loop index. GDML loop loading and expansion are not supported by PYG4OMETRY but will be implemented in a future release.

3.1. Tessellation of solids (meshing)

Creating a uniform three dimensional mesh description of all solids (including Booleans) is exceptionally useful for visualisation and other algorithms, such as overlap detection. For each Geant4 solid instance a triangular tessellated vertex-face mesh is generated and cached. This mesh is then used to determine the extent of placed instances of geometry (physical volumes) and meshes for CSG-derived solids. CSG mesh calculations are performed using a Binary Space Partitioning (BSP) tree technique in pure Python [21] or the Computational Geometry Algorithms Library (CGAL) surface meshes [22] in C++. In general the CGAL implementation is one to two orders of magnitude faster than the pure Python CSG implementation and must be used for large geometries. Triangular meshes based on CSG operations involving curved surfaces often contain large numbers of triangles. Before meshes are visualised or written to file various polygon mesh algorithms from CGAL [23] can be employed to give the meshes more desirable features.

If a user is creating and placing multiple daughter volumes within a *mother* volume then it is the user's responsibility to create a solid which fully encompasses the daughter volumes. Overlaps between daughter volumes and the mother can be detected, but it is desirable to have a mother volume shape that efficiently holds its daughters. However, there are exceptions to this rule in the form of assembly volumes.

3.2. Visualisation

When implementing geometry a rapid and robust visualisation system is key to produce error-free and efficient simulation input. A Pyg4ometry geometry hierarchy can be viewed using the popular Visualisation Toolkit (VTK). No separate scene graph is required as the Geant4 volume hierarchy is sufficient to place the meshes associated with each physical volume. A daughter volume is placed within a logical volume with a rotation $\mathbf{R}_d$, scale $\mathbf{S}_d$ and translation $\mathbf{T}_d$.

The transformation $\mathbf{M}$ and translation $\mathbf{T}$ from mother to daughter is

$$\mathbf{M} = \mathbf{S}_d\mathbf{R}_d\,, \tag{1}$$
$$\mathbf{T} = \mathbf{T}_d\,. \tag{2}$$

If the mother volume is placed in the world then the placement transformation $\mathbf{M}_w$ and translation $\mathbf{T}_w$ are expressed as

$$\mathbf{M}_w = \mathbf{M}_m\mathbf{M}_d = \mathbf{S}_m\mathbf{R}_m\mathbf{S}_d\mathbf{R}_d\,, \tag{3}$$
$$\mathbf{T}_w = \mathbf{M}_m\mathbf{T}_d + \mathbf{T}_m = \mathbf{S}_m\mathbf{R}_m\mathbf{T}_d + \mathbf{T}_m\,, \tag{4}$$

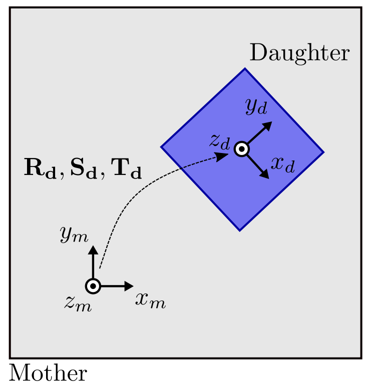

Figure 2: The placement of a physical volume inside a logical volume.

where the subscript m indicates mother volume and d indicates daughter volume. Given a hierarchy of logical and physical volumes, Equations 3 and 4 can be used recursively to place an arbitrary number of nested volumes.

The physical volume class (`pyg4ometry.geant4.PhysicalVolume`) is also used to store visualisation attributes like the solid's colour, surface or wire-frame representation and visibility. Overlaps detected in the mesh geometry are stored in the Logical Volume and can be displayed separately to allow a user to visually identify and debug the the overlaps.

Geometry needs to be augmented with other information for a complete MCRT simulation. Often other attributes need to be assigned to regions of space, for example material definition, magnetic field or optical properties. These physical properties can be used to define the visualisation attributes of a volume.

3.3. Overlap detection

All MCRT codes cannot handle spatial overlap between two geometric objects and will have ill-defined behaviour when tracking particles in such a situation. A key feature of PYG4OMETRY is the detection of potential overlaps in a way which is most useful to the user, it does this by performing an intersection operation between solid instances and determining if the resultant mesh is empty. Figure 3 shows three different types of possible overlaps, (a) protrusion of a daughter from the mother, (b) finite volume intersection between two daughters and (c) an overlap where two daughters share a face.

If the resulting intersection is non-null then the overlaps can be displayed side-by-side in the visualisation.

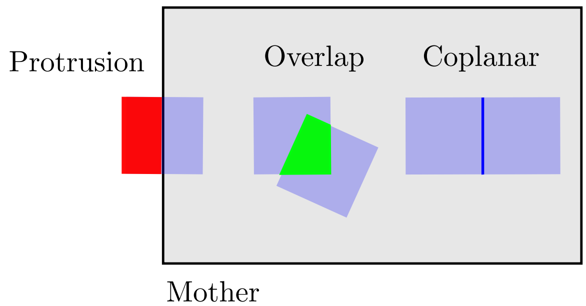

Figure 3: Schematic of the three different types of overlap between daughters of a mother logical volume.

Overlap detection in PYG4OMETRY relies of the meshes generated for each solid, any detection algorithm will be approximate. Generally the overlap detection algorithm proceeds as shown in Algorithm 1. For a logical volume with $n_{\text{daughters}}$ physical volumes, assuming meshes for solids have an average number of faces n, clearly this algorithm has complexity $\sim \mathcal{O}(n^2 n_{\text{daughters}}^2)$. This might seem like a prohibitive computational cost, but worth it considering the potential waste if small overlaps are present in the final MCRT simulation and large amounts of cluster CPU time is wasted. This algorithm clearly favours geometry descriptions which have a high degree of logical volume reuse, however this is also true of Geant4 as a whole, so the user will likely be inclined to design along such lines regardless. Due to the discrete nature of triangular meshes it is not possible to have perfect detection of overlaps, especially when curved surfaces are considered. The deviation of the meshes created from a solid can be controlled by the user reducing the chances of missing a potential overlap, but the algorithm presented here cannot capture all overlap cases. The overlap detection algorithm can present the potential overlaps quickly and easily to the user, thus significantly aiding their modelling process.

4. Rapid geometry modelling

Given the Python scripting interface, expression and tessellation engines it is possible for a user to rapidly specify the geometrical layout of the RT

Algorithm 1: The overlap checking algorithm employed in PYG4OMETRY.

Data: Logical volume v with mesh m and daughter volume meshes $d \in D$.
Result: Set S of non-null mesh intersections.
Function `Intersection(`n_1, n_2`)`
  Data: CSG meshes n_1 and n_2.
  Result: The mesh intersection of n_1 and n_2.
$V \longleftarrow \emptyset$; `// Cache tried mesh pairs.`
$S \longleftarrow \emptyset$;
for $d_1 \in D$ **do**
  $p \longleftarrow$ `Intersection(`m, d_1`)`;
  if *p is not null* **then**
    $S \longleftarrow S \cup \{p\}$;
  for $d_2 \in D$ **do**
    if $d_1 = d_2$ **or** $(d_2, d_1) \in V$ **then**
      continue;
    $q \longleftarrow$ `Intersection(`a,b`)`;
    if *q is not null* **then**
      $S \longleftarrow S \cup \{q\}$;
    $V \longleftarrow V \cup \{(d_1, d_2)\}$;

problem, vary the parameters of the geometry and visualise it. When a user has achieved the desired geometry without geometry overlaps, a GDML file can be written to file from the internal memory representation. Some example code is presented in Listing 2. The structure should be familiar to regular users of Geant4 or GDML, apart from the new class described in the previous section called the `Registry`. First the `Registry` is created to store all the PYG4OMETRY objects; followed by constants; then materials and properties, solids and logical and physical volumes; finally the whole geometry can be saved as a GDML file or visualised using VTK.

Listing 2: A simple Python script using PYG4OMETRY to create a simple Geant4 geometry.

```
# import modules
import pyg4ometry.gdml as gd
import pyg4ometry.geant4 as g4
```

```python
import pyg4ometry.visualisation as vi

# create empty data storage structure
reg = g4.Registry()

# expressions
wx = gd.Constant("wx","100",reg)
wy = gd.Constant("wy","100",reg)
wz = gd.Constant("wz","100",reg)
bx = gd.Constant("bx","10",reg)
by = gd.Constant("by","10",reg)
bz = gd.Constant("bz","10",reg)
br = gd.Constant("br","0.25",reg)

# materials
wm = g4.MaterialPredefined("G4_Galactic")
bm = g4.MaterialPredefined("G4_Fe")

# solids
wb = g4.solid.Box("wb",wx,wy,wz,reg)
b = g4.solid.Box("b",bx,by,bz,reg)

# structure
wl = g4.LogicalVolume(wb, wm, "wl", reg)
bl = g4.LogicalVolume(b, bm, "b", reg)
bp1 = g4.PhysicalVolume([0,0,0],
                        [0,0,0],
                        bl, "b_pv1", wl, reg)
bp2 = g4.PhysicalVolume([0,0,-br],
                        [-2*bx,0,0],
                        bl, "b_pv2", wl, reg)
bp3 = g4.PhysicalVolume([0,0,2*br],
                        [2*bx,0,0],
                        bl, "b_pv3", wl, reg)

# define world volume
reg.setWorld(wl.name)
```

```
# physical volume vistualisation attributes
bp1.visOptions.color = (1,0,0)
bp1.visOptions.alpha = 1.0
bp2.visOptions.color = (0,1,0)
bp2.visOptions.alpha = 1.0
bp3.visOptions.color = (0,0,1)
bp3.visOptions.alpha = 1.0

# gdml output
w = gd.Writer()
w.addDetector(reg)
w.write("output.gdml")

# visualisation
v = vi.VtkViewer(size=(1024,1024))
v.addLogicalVolume(wl)
v.addAxes()
v.view()
```

An example of the VTK output for code Listing 2 is shown in Figure 4. Significantly more complex geometries can be developed using a structure similar to that shown.

The PYG4OMETRY Python code in the example is approximately as expressive as the GDML it writes. The benefit of wrapping GDML in Python is that it allows very rapid prototyping of geometry without concerns of C++ compilation (in the case of implementing the geometry directly in Geant4) or writing well formed XML (in the case of GDML). Effectively the Python interpreter checks input for syntax errors when using PYG4OMETRY classes. Another key benefit is the ability to use the PYG4OMETRY code to create programmatic converters between different geometry languages or more generally manipulation and transformation of the geometry stored in memory. The rapid modelling example given in Listing 2 and Figure 4 is rather trivial, a significantly more complex example is shown in Figure 7.

5. FLUKA to GDML conversion

FLUKA geometry is based upon a limited set of primitives (referred to as *bodies*) which can be combined using Boolean operations. A *zone* consists

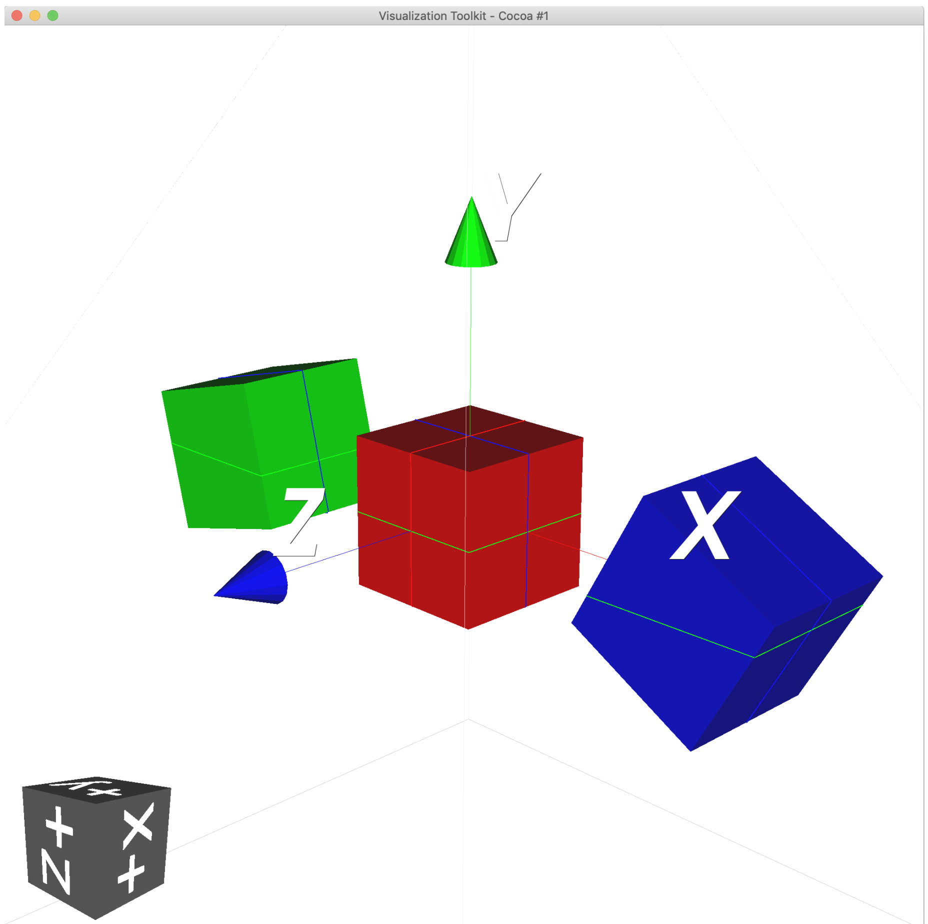

Figure 4: VTK visualisation output from code Listing 2.

of one or more bodies or *subzones* combined using intersections and subtractions. Zones may then be further combined using union operations to form *regions*, which is defined as the union of one or more zones, as well as a material.

Each FLUKA body is represented in PYG4OMETRY with a corresponding class, and in turn each class has methods which returns a GDML primitive solid and that solid's rotation and position such that it matches its FLUKA equivalent. The expansion, translation and transform geometry directives are each folded into one or more of these three methods. The mapping of FLUKA bodies to GDML solids is shown in Table 1. It is worth noting that many of the FLUKA bodies are infinite in extent, but are mapped to finite GDML solids. The translation of infinite bodies to equivalent finite solids is one of the main and most involved steps in the conversion process. This mapping is possible because whilst FLUKA bodies can be infinite in extent, all zones and regions must be finite. Zones and regions are then composed by instantiating these classes and adding body instances to them. Each Zone and Region instance can then return its equivalent GDML Boolean solid.

The FLUKA CSG ASCII format is parsed using an ANTLR4-generated parser, producing an AST. The resulting AST is then inspected sequentially (walked) to populate Region instances with zones and bodies. With the region instances populated they can then be manipulated and translated into GDML. The translation involves a number of special steps to bridge the two disparate formats and ensure the resulting GDML is well-formed and usable in Geant4. Some of these steps are simple, for example in FLUKA unions can be disconnected, but in Geant4 specifically only MultiUnions can be disconnected, such that MultiUnions are used throughout the converted geometry instead of the usual binary unions. Other procedures are more involved and are discussed in the rest of this section.

Listing 3: A simple PYG4OMETRY Python script to load a FLUKA file.

```
reader = pyg4ometry.fluka.Reader("FlukaFileName.inp")
g4_reg = pyg4ometry.convert.fluka2Geant4(reader.flukaregistry)
logical = g4_reg.getWorldVolume()
```

5.1. Infinite bodies

The majority of bodies in FLUKA are infinite in extent, and fall broadly into four categories, half-spaces, infinitely-long cylinders, infinitely-long elliptical cylinders and quadric surfaces. Translating these bodies to Geant4

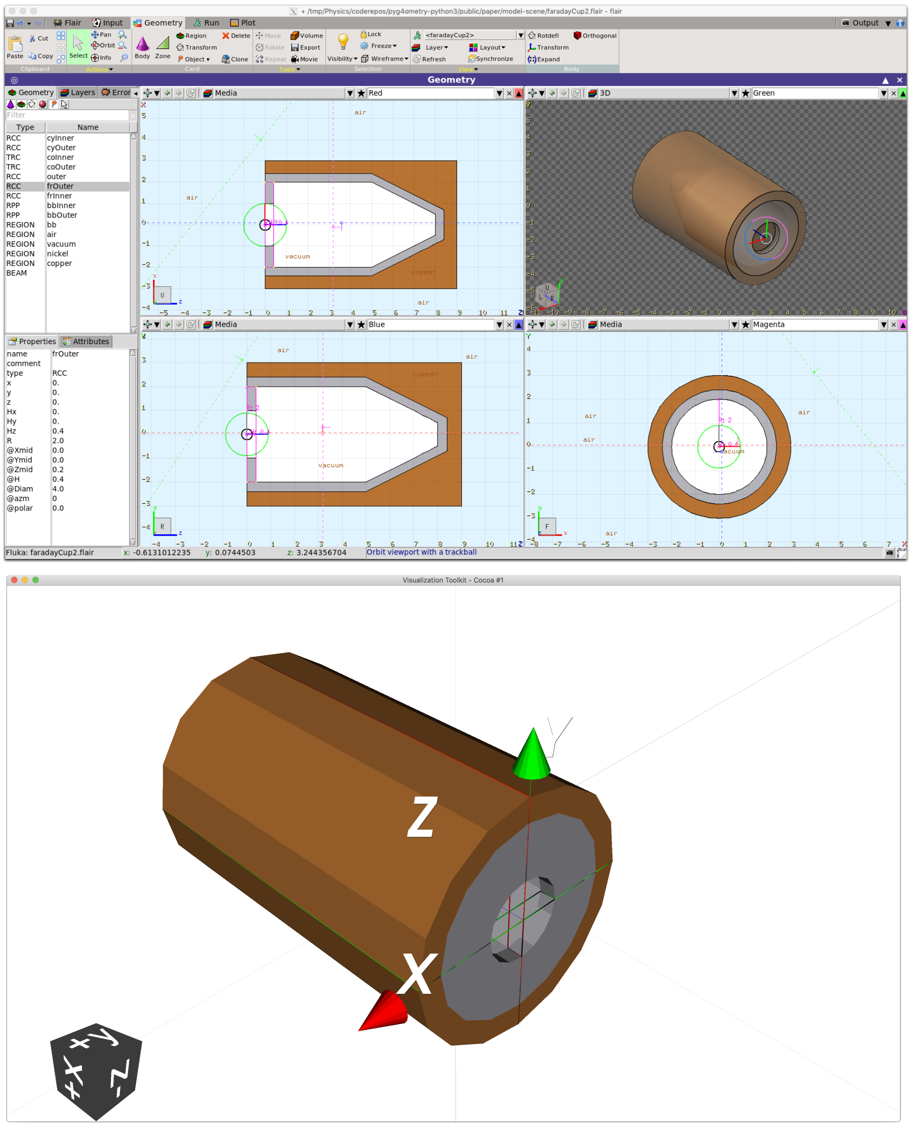

Figure 5: Example conversion of a simple FLUKA geometry to GDML. Above: the original FLUKA geometry displayed in flair, FLUKA's graphical user interface. Below: the GDML geometry viewed using Pyg4ometry. The example is a Faraday cup used to capture and measure accelerator beam charge.

Table 1: FLUKA bodies and their corresponding Pyg4ometry classes.

FLUKA body	Pyg4ometry class
RPP (Rectangular parallelepiped)	Box
BOX (General rectangular parallelepiped)	Box
SPH (Sphere)	Orb
RCC (Right circular cylinder)	Tubs
REC (Right elliptical cylinder)	EllipticalTube
TRC (Truncated Right Angle Cone)	Cons
ELL (Ellipsoid of Revolution)	Ellipsoid
WED/RAW (Right Angle Wedge)	ExtrudedSolid
ARB (Arbitrary Convex Polyhedron)	TessellatedSolid
XYP (X-Y Infinite half-space)	Box
XZP (X-Z Infinite half-space)	Box
YZP (Y-Z Infinite half-space)	Box
PLA (Generic infinite half-space)	Box
XCC (X-axis Infinite Circular Cylinder)	Tubs
YCC (Y-axis Infinite Circular Cylinder)	Tubs
ZCC (Z-axis Infinite Circular Cylinder)	Tubs
XEC (X-axis Infinite Elliptical Cylinder)	EllipticalTube
YEC (Y-axis Infinite Elliptical Cylinder)	EllipticalTube
ZEC (Z-axis Infinite Elliptical Cylinder)	EllipticalTube
QUA (Quadric surface)	TessellatedSolid

requires generating the equivalent finite solid whilst retaining the same final finite Boolean shape. This is achieved with the use of axis-aligned bounding boxes (AABBs), in which the FLUKA body is translated to a finite solid with with dimensions slightly larger than the AABB. The lengths of infinite (elliptical) cylinders are reduced to finite equivalents with lengths slightly greater than the bounding box. Similarly, half-spaces are reduced to boxes with one face acting as that of the half-space face, and quadric surfaces are sampled only over the volume denoted by the AABB. Furthermore, the positions of these solids are moved as close to the bounding box as possible whilst retaining an identical final Boolean geometry.

Generating these bounding boxes over which the bodies should be translated involves first evaluating each region with very large Tubs, EllipticalTubes and Boxes (by default 50 km in length), such that they are effectively infinite for most reasonable use cases. Pyg4ometry's CSG meshing is then used to generate a mesh for each region from which the axis-aligned bounding box can be extracted. Each region is then evaluated a second time with respect its respective bounding box, with all of its constituent infinite solids being reduced as described above. The implementation is described in Algorithm 2. This algorithm works robustly for infinite (elliptical) cylinders and half-spaces as the number of facets is independent of the size of the solid, so generating the initial mesh from large solids works well. Generating a quadric surface over a very large volume in space whilst retaining topological information is computationally very expensive, so to resolve this the user must provide the approximate axis-aligned bounding box of any region in which a quadric is used.

5.2. Removing redundant half-spaces

The above algorithm for replacing infinite FLUKA bodies with finite Geant4 solids works well in most cases, but additional care must be taken for redundant infinite half-spaces. A redundant infinite half-space is defined as one which has no effect on the shape of the final Boolean solid. Whilst this may be true of arbitrary bodies, it is most problematic for half-spaces as after the infinite body reduction has been performed. If the half-space is far away from the region's AABB, then it can result in a malformed Boolean. Such half-spaces are filtered from their respective regions during the conversion process by calculating the nearest distance from the centre of the AABB to the half-space face. If this distance is greater than the centre-to-corner distance of the AABB, then that half-space is removed from the region during conversion.

5.3. Coplanar faces

Coplanar faces, in which the faces of two union components or that of two regions are perfectly coplanar in FLUKA are ubiquitous and present no difficulties to the operation of the program. However, in Geant4 these will generally result in tracking errors. These must be handled robustly to ensure the resulting geometry is usable. Coplanar faces are resolved automatically by slightly decreasing the size of every body that is used in an intersection, and increasing the size of every body used in a subtraction. These rules are inverted for nested subtractions and work well for guaranteeing well-formed geometry that is free from tracking errors.

Algorithm 2: The infinite-body minimisation algorithm employed in the conversion of Fluka to GDML.

Data: FLUKA regions to be converted to GDML.
Result: GDML solids equivalent to the FLUKA regions built from minimally-sized primitive solids
Function `ToGDMLSolid`(b, a)
  Data: FLUKA body b with axis-aligned bounding box a.
  Result: GDML solid equivalent to b bounded by the volume a.
Function `RegionAABB`(r)
  Data: FLUKA region r.
  Result: Axis-aligned bounding box (AABB) of the FLUKA region.
$B \longleftarrow \emptyset$; `// Map of regions to AABBs.`
for *r in regions to be converted* **do**
  $B[r] \longleftarrow$ `RegionAABB`(r)
`// Map of bodies to minimal bounding boxes.`
$E \longleftarrow \emptyset$;
for *b in bodies in regions to be converted* **do**
  for *r in regions in which body b is used* **do**
    $E[b] \longleftarrow E[b] \cup$ `RegionAABB` (r);
$G \longleftarrow \emptyset$;
for *body b and AABB a in E* **do**
  `// Map the fluka bodies to minimal GDML solids by converting with the minimal AABB,` a.
  $G \longleftarrow$ `ToGDMLSolid`(b, a);
for *r in regions to be converted* **do**
  build the corresponding GDML for r from the set of minimal GDML primitives in G.;

5.4. Materials

Any useful translation between geometry description formats must also account for materials, and accordingly PYG4OMETRY correctly translates FLUKA materials to GDML. FLUKA materials can be divided into built-in, single-element and compound materials. Built-in materials are simply those that are predefined by FLUKA, single-element materials are described with a single `MATERIAL` card, and compound materials are described with one `MATERIAL` card followed by one or more `COMPOUND` cards. These three alternatives are represented in PYG4OMETRY with the `BuiltIn`, `Material`, and `Compound` classes. Populating a hierarchy of instances using these classes is involved due to the fact that recursively-defined materials in FLUKA input files need not be defined in a logical order. Namely, a given compound may be defined before the materials that it consists of are themselves defined. To account for this it is necessary to correctly compute the instantiation order so that the above classes and be instantiated correctly. To do this a directed acyclic graph is populated with the materials and their constituents, after which a topological sort is performed so that the compound materials are sequenced after their constituents. Mapping this set of nested FLUKA material instances to GDML material instances is then straight forwards as the GDML material semantics are slightly more expressive than FLUKA's, so a one-to-one mapping is trivial.

5.5. Lattice

FLUKA supports modular geometries with the use of the `LATTICE` command. Figure 6 demonstrates this capability. The arbitrarily complex *basic unit* can be defined once and used multiple times by placing one or more empty *lattice cells* with the associated rototranslation from that lattice cell to the basic unit. The lattice cells themselves will generally lack structure and simply serve as a reference to the basic unit. The rototranslated lattice cell must fully contain the basic unit and all of the regions within it. When a particle steps into the lattice cell, the rototranslation is applied to that particle and it is is taken into the basic unit, and the simulation continues within the basic unit. Any particles leaving the basic unit will be translated back to the lattice cell with the inverted rototranslation. Up to two levels of lattice nesting the are supported in FLUKA.

This feature is clearly analogous to the logical/physical volume feature in Geant4, although it is less explicit as the contents of a given logical volume are clearly stated, whereas the contents of a lattice cell are implied by the

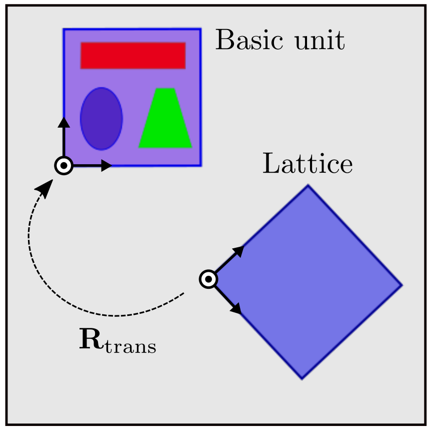

Figure 6: The lattice feature demonstrating a lattice cell referring to its basic unit with the rototranslation R_{trans}. Any particle entering the cell will be transformed onto the basic unit with R_{trans}, and when leaving the basic unit, back to the cell with R_{trans}^{-1}.

combination of the rototranslation and the locations of its lattice cell and basic unit.

Translating the lattice construct into a logical volume (basic unit) with many physical volumes (lattice cells) requires associating each lattice cell with the full contents of its corresponding basic unit (typically several regions). This is achieved by meshing the complete FLUKA geometry and then rototranslating the lattice cell mesh with its associated rototranslation. By construction this will translate the lattice cell mesh directly onto the full basic unit mesh. Finally, in checking for overlaps, the regions within the the basic unit can be determined. In the final conversion step, the lattice cell can simply be replaced with a physical volume that refers back to logical volumes located in the previous step. Thus a basic unit with one or more lattice cells can be translated into Geant4's logical volume with one or more physical volumes. As has been stated, FLUKA supports two levels of nesting, but currently the conversion to GDML supports only one. However, extending to an extra level is simple in that it only involves an extra application of a rototranslation matrix before checking for overlaps.

5.6. Discussion

Figure 5 shows a Faraday cup implemented in FLUKA and accurately translated to GDML using PYG4OMETRY. Many of the steps described above were directly applied in this model and all the features are tested and demonstrated in the repository. This set of algorithms for bridging FLUKA with GDML covers a very broad range of geometries, however a number of possible improvements for the future remain. For example, Boolean solids in FLUKA can in general be disconnected, and this typically manifests itself in the form of disconnected unions, but is also allowed for intersections and subtractions. Disconnected unions are readily available in GDML with the use of the MultiUnion solid type, however there is no valid means to construct a disconnected intersection or subtraction in GDML. One possible solution would be to detect these two cases, and if found, split them into their constituent parts and place separately as TesselatedSolid instances.

Furthermore, the quadric surface conversion can be further improved by specialising on some of the individual forms of the quadric. Simply converting every quadric into a TesselatedSolid comes at a potential performance cost in the tracking, as well as usability in PYG4OMETRY as the user must provide an AABB. In some cases tessellation in unavoidable (for example a hyperbolic paraboloid), but parabolic cylinders could for example be translated to an ExtrudedSolid. This could provide both a performance improvement in the tracking time and make PYG4OMETRY easier to use as an AABB would not need to be provided beforehand. This has not been implemented as quadrics are relatively rarely used, but where quadrics are used it is often in the form of parabolic cylinders (e.g. magnet pole tips) and this specialisation in particular would be worth implementing.

6. GDML to FLUKA conversion

It is relatively straight forward to convert Geant4 geometry to FLUKA. Each of the Geant4 solids can be mapped to a FLUKA region. A region is a region of space defined by a material and the Boolean disjunction (a union using the operator | in free format geometry) of one or more zones. Each zone is then defined in terms of the conjunction (intersection with +, subtraction with −) of one or more primitive bodies, as well as parentheses to determine the order of operations within the zone. FLUKA has 20 of these primitive bodies, listed in Table 1 and, in general, infinite-extent bodies have tracking accuracy and efficiency benefits over finite ones. Key for conversion are XY-, XZ-, YZ-Planes (XYP, XZP, YZP), arbitrary plane (PLA), Z-axis aligned cylinder (ZCC), Z-axis aligned elliptical cylinder (ZEC), sphere (SPH), truncated right-angle cone (TRC) and general quadric surface (QUA). Some solids in Geant4 directly map to a single FLUKA body, others require the construction of a simple FLUKA CSG tree combining these primitives. Table 6 lists the mapping between Geant4 solids and the bodies used to compose a FLUKA region.

There are some important issues which must be considered to perform an accurate conversion. A Geant4 logical volume solid might need to be created from a FLUKA region consisting of many zones (i.e. unions), so $R_1 = +z_1 \,|+ z_2$, similarly a void is required to locate daughter volume placements which again might also be converted to a region of multiple zones $R_2 = +z_3 \,|+ z_4$. So creating the FLUKA region is then

$$\begin{aligned} R_1 - R_2 &= (+z_1 \,|+ z_2) - (+z_3 \,|+ z_4) && (5) \\ &= (+z_1 - z_3 - z_4) \,|\, (+z_2 - z_3 - z_4)\,. && (6) \end{aligned}$$

Geant4 has the Boolean solids associated with difference, union and intersection, so in addition to Equation 6, both $R_1 \cup R_2$ and $R_1 \cap R_2$ is required in FLUKA notation, so

$$\begin{aligned} R_1 \cup R_2 &= (+z_1 \,|+ z_2) \cup (+z_3 \,|+ z_4) && (7) \\ &= +z_1 \,|+ z_2| + z_3 \,|+ z_4 && (8) \end{aligned}$$

and

$$\begin{aligned} R_1 \cap R_2 &= (+z_1 \,|+ z_2) \cap (+z_3 \,|+ z_4) && (9) \\ &= +z_1 + z_3 \,|+ z_1 + z_4 \,|+ z_2 + z_3 \,|+ z_2 + z_4\,. && (10) \end{aligned}$$

FLUKA (apart from the `LATTICE` directive) has no sense of a volume hierarchy. Each body is placed with translation, rotation and expansion geometry directives in global coordinates. A transformation from world coordinates to a physical volume is built up by recursively applying daughter volume transformations and this is used to place FLUKA bodies. This in practice is very similar to the procedure to create the VTK visualisation already described in Section 3.2.

In FLUKA every single point in space needs to be associated with one and only one region. This presents a problem when converting Geant4 logical volumes to FLUKA regions, as the logical volume outer solid S_{logical} need to

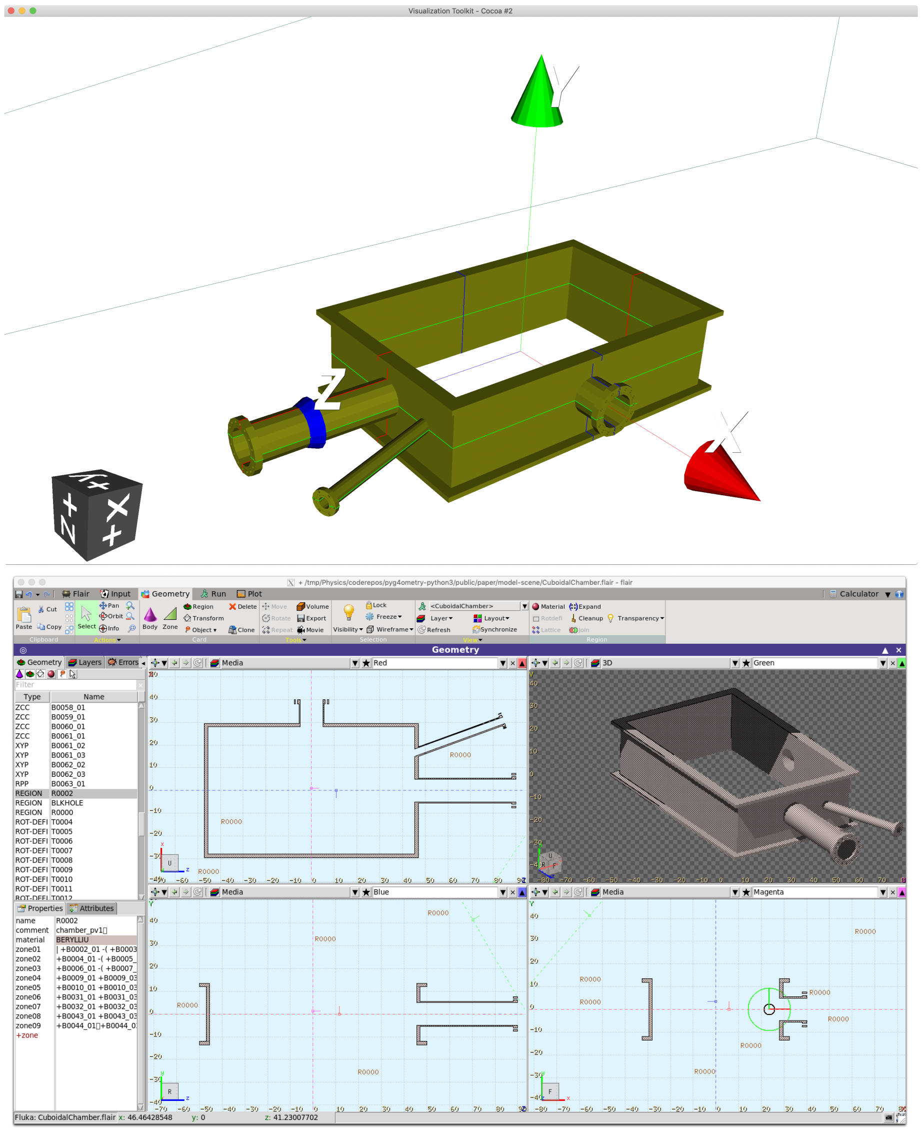

Figure 7: Example conversion of a simple GDML geometry to FLUKA, above: the original model in Pyg4ometry, below: the converted FLUKA geometry viewed in flair. The example is a sector bend dipole electromagnet.

Geant4 solid	FLUKA region construction
Box	+2 XYP + 2 XZP + 2 YZP
Tube	+ZCC - 2 PLA -2 XYP - ZCC
CutTube	+ZCC - 4 PLA -ZCC
Cone	+TRC - TRC - 2 PLA
Para	+ 6 PLA
Trd	+ 6 PLA
Trap	+ 6 PLA
Sphere	+SPH - SPH - 2 PLA - 2 TRC
Orb	+SPH
Torus	+N ZCC - N PLA
Polycone	+N TRC -2 PLA
Polyhedra	+N PLA
Eltube	+ZEC - 2 XYP
Ellipsoid	+ELL - 2 XYP
Elcone	+QUA - 2 XYP
Paraboloid	+QUA - 2 XYP
Hype	+QUA - QUA - 2 XYP
Tet	+4 PLA
Xtru	+N PLA
TwistedBox	+N PLA
TwistedTtap	+N PLA
TwistedTrd	+N PLA
TwistedTube	+N PLA
Arb8	+N PLA
Tessellated	+N PLA
Union	$R_1 \cup R_2$
Subtraction	$R_1 - R_2$
Intersection	$R_1 \cap R_2$
MultiUnion	$R_1 \cup R_2 \cup R_3 \cup$

Table 2: GDML/Geant4 solids and the mapping to FLUKA regions.

have the daughter solids $S_{\mathrm{daughter},i}$ subtracted. A solid which can be converted to a region S_{region} is then

$$S_{\mathrm{region}} = S_{\mathrm{logical}} - S_{\mathrm{daughter},1} - S_{\mathrm{daughter},2} - S_{\mathrm{daughter},3} \dots \tag{11}$$

If a logical volume has a number of daughter volumes which are also possibly

Boolean solids, then computing S_{region} can become very complex because of Equations 6, 8 and 10.

Figure 7 shows an example conversion from GDML to FLUKA. The example is a vacuum chamber with three ConFlat flange (CF) beam pipes connected to CF flanges. The top and bottom plates have been removed to display the geometry more clearly. The model is formed of `G4Box` and `G4Tubs` and Boolean operations of subtraction, intersection and union.

6.1. Non-Convex solid decomposition

In general the BREP solids used by Geant4 include non-convex solids, such as Polycone, Xtru, and Twisted. These particular solids are problematic when converting them to a FLUKA-readable format, as non-convex solids can only be created by the union of convex zones. There are two groups which need to considered, firstly those where 2D polygonal section needs to be decomposed, these include Polycone, Polyhedron and Xtru, secondly those where three-dimensional convex decomposition is required TwistedBox, TwistedTrap, TwistedTrd, TwistedTube and Tessellated. The second classification of non convex solids are converted to CGAL Nef polyhedra [24] and decomposed to convex polyhedra [25].

6.2. Disjunctive normal form and degenerate surfaces

Typically FLUKA will decompose a region into disjunctive normal form (DNF), this normal form is characterised as the union of intersections and subtractions,

$$R = z_1 \mid z_2 \mid z_3 \mid z_4 \ldots \tag{12}$$

Defining regions in terms of the DNF allows the rapid test of whether a point is inside that region. Testing each zone z_i of R can terminate if a point is determined to be inside any of the convex zones z_i. In general Equation 11 does not have the form of a DNF. If there are many levels of logical-physical volume placement, then recursive application of Equation 11 will create a nested set of parentheses. There are some conditions where a general Boolean expression can yield an exponential explosion of the final DNF. There are well known algorithms to convert logical expressions to its DNF. PYG4OMETRY can simplify parentheses from a region by creating a corresponding SymPy [26] Boolean expression and using the `to_dnf` method. Further simplification of the CSG tree leverages PYG4OMETRY's meshing capabilities combined with CSG pruning algorithms based on [27]. FLUKA by default will try to expand all regions to their DNF at runtime, which inevitably can result in the sort of exponential explosion already mentioned. Until version 4.0, if FLUKA identified such an explosion, it would report an error and exit, thus making such models impossible to run. As of version 4.0, however, this expansion can be disabled, and the tracking algorithms will walk the CSG trees verbatim, at some tracking efficiency cost. Therefore, most output generated from the GDML to FLUKA conversion described here can only be used with FLUKA 4.0.

6.3. Materials

Pyg4ometry converts GDML/Geant4 materials to FLUKA `MATERIAL` and `COMPOUND` cards. Geant4 has a class `G4Material` to assign material state (density, physical state, temperature and pressure) to a logical volume. `G4Material` has two main constructors, the first where an atomic number is supplied and the second is when `G4Element` instances and relative atomic or mass abundances are provided. The *simple* `G4Material` is converted to a `MATERIAL` card, whilst the *element* `G4Material` is converted to a `COMPOUND` card. There are similar issues when converting `G4Element` to FLUKA, as `G4Element` can either be simple, i.e. defined only by atomic number and mass, or composite and defined by an admixture of relative abundances of `G4Isotope`. A similar mapping is performed so that if a `G4Element` is simple it is directly converted to a FLUKA `MATERIAL` card, and the *isotope* `G4Element` is converted to a `COMPOUND` card. Geant4 also defines a set of standard materials [28] or compounds from the US National Institute of Standards and Technology (NIST), so a user can for example, simply specify the name `G4_STAINLESS-STEEL`. Pyg4ometry contains a matching database and creates the appropriate FLUKA cards from these names during the conversion. This database is updated by running a small Geant4 program to output the appropriate material data.

6.4. Discussion

Overall the conversion to FLUKA input format from GDML is quite advanced and stable. Relatively large experimental simulations have been converted from GDML to FLUKA and have been used to produce simulation results. The conversion described still requires a user to understand how geometry is specified in both Geant4 and FLUKA. For example the subtraction of many non-primitive body daughter volumes from a mother will create a very complex non-DNF region which is inefficient when viewed in FLUKA's graphical user interface (GUI), flair, and used for simulation in FLUKA. A geometry designer can avoid this by restricting daughter volumes solids in Geant4 that can be expressed in DNF simply, for example a cuboid. If Pyg4ometry is used during the creation of geometry then multiple codes can be targeted without additional user effort. There are still a few outstanding technical issues with the conversion, which are discussed in this section.

Currently replica, division and parametrised placements are not implemented and will be added in a future release. In Geant4 it is possible to create scaled solids or placements with reflections, referred to as *scale* in Geant4. FLUKA rototranslations do not support reflections and implementing reflections require transformation of the body definitions. This is not yet implemented in the current version of Pyg4ometry.

In general, recursive application of Equations 6, 8, 10 and 11 can result in very complex regions when converting from Geant4/GDML to FLUKA. The complexity of the final region expression can be compounded if transformed to DNF. The final region Boolean expression can be simplified by evaluating using the meshes for the surfaces, only retaining terms which do not evaluation to a null mesh.

It is possible given the GDML to FLUKA conversion algorithm described in this paper that coplanar overlaps exist in the FLUKA geometry. In Geant4 there is no connection between surfaces used to specify on logical volume solid and another logical volume solid. So for example if a logical volume solid shared a face with one its daughter volume solids the body would be duplicated in the final FLUKA file. It is possible to remove obvious degeneracies but this is complicated by placements of bodies. It should be possible to cast FLUKA bodies *without* rototranslations into a *normal form* which can be used to test for approximate equality. Approximate equality is required as multiple application rototranslations will accrue numerical rounding errors. A simple example of this is the XYP and PLA, it simple to transform an XYP into a PLA with an appropriate rotation. This would allow for the removal of degenerate surfaces removing potential coplanar overlaps and also reduce the final converted file size.

The twisted primitives need to be decomposed into a union of convex solids. This decomposition does not always succeed or produces a far from optimal number of convex solids. An alternative to implement these solids is to approximate each layer of the twisted solid as a union of tetrahedra. A similar problem exists for general tessellated solids, which in general are non-convex and need decomposition into convex hulls. A potential way to avoid a computationally expensive decomposition step is to create a region formed from the unions of tetrahedra. There are numerous algorithms for tetrahedralisation of surface meshes in both CGAL and TetGen [29] and will be implemented in a future release. Even if a stable and general method for converting tessellated solids exists it not efficient to define tessellated objects in this way and memory, body or zone limits might be reached in FLUKA and so limit the size of CAD or STL models which might be loaded.

7. STL and CAD to GDML conversion

STL and CAD conversion are closely related. In both cases the solid (in the case of STL) and solids (in the case of CAD) are converted to tessellated solid(s). STL is a relatively simple file format that can be loaded using pure Python. As STL files typically only contain a single solid the `stl.Reader` provides a single solid and not a logical volume as with the other file readers.

STEP and IGES files can be loaded into Pyg4ometry, via an interface based on FreeCAD [7]. FreeCAD is an open source CAD/CAM program, which in turn is based on OpenCASCADE. FreeCAD allows for scripting in Python and acts as a simple-to-use interface to OpenCASCADE. A STEP CAD file could be considered as a hierarchical tree of parts and *part assemblies*, where a part assembly is a collection of *part features*. A part feature can be used to create a triangular mesh which can be used to instantiate a Pyg4ometry tessellated solid. The placement of the part feature is extracted from the STEP file and used to create an appropriate physical volume. Assignment of materials and visualisation attributes must be performed by the user after conversion to GDML as rarely will CAD/CAM packages included the detailed information required for MCRT codes. Listing 4 shows how Pyg4ometry can load step files.

Listing 4: A simple Pyg4ometry Python script to load a STEP file.

```
reader = pyg4ometry.freecad.Reader("CadFileName.step")
reader.relabelModel()
reader.convertFlat()
logical = reader.getRegistry().getWorldVolume()
```

Compared to other file readers, two additional steps are required `relabelModel` and `convertFlat`. CAD model part names can contain characters which are

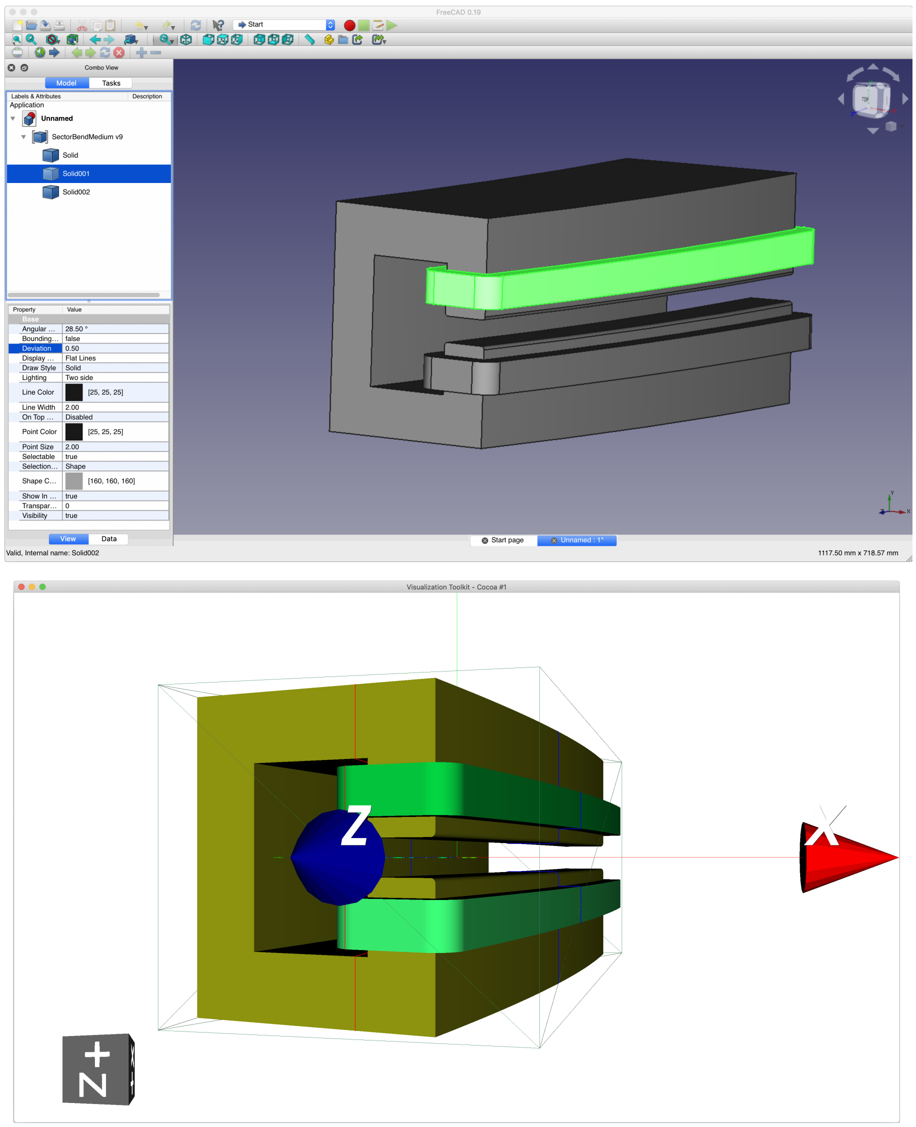

Figure 8: Example conversion of a simple CAD (STEP) geometry (sector bend dipole electromagnet) to GDML, above: the original model in FreeCAD, below: the GDML geometry viewed using PYG4OMETRY.

not allowed in Python dictionaries so need to be replaced by `relabelModel`. CAD models might also have a hierarchy of parts and assemblies, these are converted without this structure by `convertFlat`. In general there is no requirement to avoid geometric overlap of parts in a CAD file. This will result in overlaps of the solids in the converted tessellated solids. This is avoided by shrinking each solid, this is done by computing the a normal $\mathbf{n}$ for each vertex $\mathbf{v}$ and shifting its position by $\epsilon\mathbf{n}$, so the new vertices are $\mathbf{v} - \epsilon\mathbf{n}$. The degree of shrinking is user-controllable.

An example geometry representing a dipole electromagnet, consisting of three parts was created in Autodesk Fusion 360 and saved as a STEP file, the PYG4OMETRY-produced output is shown in Figure 8.

8. Complete simulation example

PYG4OMETRY is designed to be as flexible as possible and offer the user a wide range of usage styles, input files and workflows. A fictitious beam line was created to demonstrate the capabilities of PYG4OMETRY, this creates a composite *scene* which consists of geometry sources from the different formats described in this paper. The beamline consists of a vacuum chamber (modelled in PYG4OMETRY), a vacuum gate valve (STL from the manufacturer), a triplet of quadrupole magnets (exported from BDSIM), a sector bend dipole electromagnet (created in Autodesk Fusion 360) and finally a Faraday cup (FLUKA geometry designed in flair). Each different file is loaded using the appropriate PYG4OMETRY reader class and then placed as a physical volume. The final composite geometry is shown in Figure 9. It must be noted when this complete geometry is written to GDML and loaded into Geant4 it cannot be visualised in anything but the ray tracer because of limitations in the OpenGL visualisation in Geant4. Another important note is that this geometry cannot be converted to FLUKA as it contains tessellated solids (both the STL gate valve and dipole magnet).

Having geometry wrapped in a suitable API allows a wide range of processes to be performed simply and programmatically. The benefits of the API are particularly apparent when needing to process large amounts of geometry efficiently and precisely. Possible transformations include merging registries, removing volumes (de-featuring), editing solid parameters, changing logical volume materials and converting logical volumes to assembly volumes.

The merging of registries and removing volumes is required to create the example shown in Figure 9. Each sub-component is stored in a separate reg-

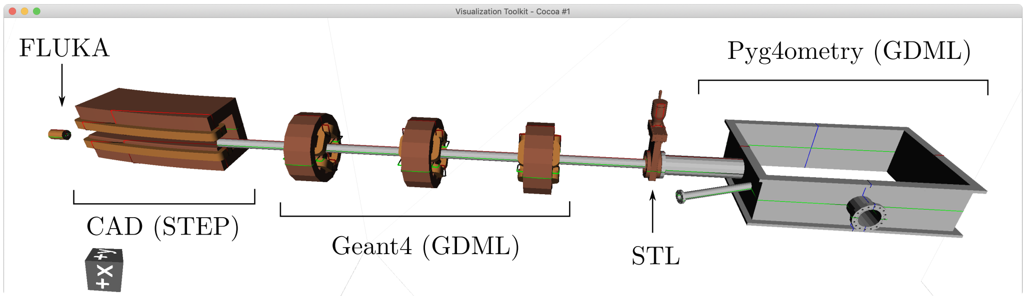

Figure 9: Complete compositing example using PYG4OMETRY.

istry and these have to be combined without any GDML tags clashing. The FLUKA to GDML conversion creates logical volumes which are not generally required in Geant4, so for example the air surrounding the Faraday cup, which is needed to specify a FLUKA geometry is converted to GDML but can be safely removed. Examples of other workflows or geometry manipulation processes can be found in the PYG4OMETRY online manual.

9. Quality assurance

The source and manual code for PYG4OMETRY is stored in a git repository (`https://bitbucket.org/jairhul/pyg4ometry`), where a public issue tracker for users is hosted to report problems or bugs with the code. The manual is created using Sphinx, a Python documentation generator (`http://www.pp.rhul.ac.uk/bdsim/pyg4ometry/`). PYG4OMETRY uses mature packages available for Python as dependencies. PYG4OMETRY has two subpackages that require compilation `pyg4ometry.pycsg` and `pyg4ometry.pycgal` in C++. PYG4OMETRY package dependencies and extensions are installed easily using `setuptools`. All aspects of PYG4OMETRY are routinely checked using software tests, of which 543 are available, resulting in 84% code coverage. The tests also serve as minimal examples to help users understand the code operation.

10. Conclusions and discussion

The authors believe that tools to quickly create geometry, either from scratch or by conversion, for Monte Carlo particle transport programmes will save significant amounts of time and user effort and will ultimately yield more accurate simulations. PYG4OMETRY is a relatively complete implementation of a geometry creation tool, whilst heavily internally based on Geant4 and GDML, it can have utility for users of all MCRT codes. PYG4OMETRY can clearly be extended to other formats or applications. Presently it provides a coherent and uniform interface to existing tools and utilities, also by using the Python programming language PYG4OMETRY allows the programmatic control of geometry creation or modification. This approach allows the integration of other available tools [30] into a unified workflow.

Users should be aware of issues with PYG4OMETRY. The conversions between CAD/STL, GDML, and FLUKA cannot be considered bidirectional. For example Geant4 tessellated solids cannot be converted easily to FLUKA which does not have a convenient way of representing this geometry. In general a user would be unwise to attempt to convert a very large geometry from one format to another, but concentrate on smaller conversions of constituent parts. Workflows should focus on conversion from a primary format and then create conversions to another format as the need arises. This paper outlines the creation of geometry using PYG4OMETRY, and whilst that geometry is subsequently loaded into Geant4, flair and FLUKA, detailed studies of the MCRT simulation performance is beyond the scope of this publication and will be addressed in the future.

There are many output format extensions that can be considered for PYG4OMETRY. Geant4 geometry is primarily created by writing C++ programmes, so an output writer that converts the PYG4OMETRY in-memory representation to C++ will allow rapid geometry modelling but inclusion of the geometry into an existing Geant4 application. This is not implemented in the current version but could be relatively quickly implemented for users that require this functionality. At present PYG4OMETRY supports reading and writing Geant4 (GDML) and FLUKA files but could be extended without significant effort to other MCRT codes like MCNP.

There are more complex extensions that can be considered for inclusion into PYG4OMETRY. The meshes created by PYG4OMETRY are generally of very high quality and can be used for a wide range of applications. An idea already being developed is the export of the geometry mesh data to data formats used in augmented or virtual reality software to create interactive visualisations of MCRT simulations. Triangular meshes also have applications for GPU-accelerated photon tracking in liquid noble dark matter detectors. Paraview/VTK are becoming standard software for complex 3D visualisation and the ability to write geometry to formats readily loaded and manipulated by these programs will significantly aid the presentation of geometry along with the results of the MCRT simulations.

PYG4OMETRY is principally a toolkit but various visualisation and user interface extensions would significantly aid geometry creation workflows. A graphical user interface would enable a user without any programming experience to create geometry for MCRT simulations and expand the number of potential users. PYG4OMETRY has been designed to interface with a GUI in a relatively straightforward manner. The VTK visualiser currently limits the display of very large models as geometry instances are replicated as opposed to reused in visualisation. However, this could be drastically improved in a future release.

The conversion which would most dramatically enhance the creation of RTMC geometry is CAD to Geant4 or FLUKA without use use of a triangular or tetrahedral mesh. There are existing approaches to decompose BREP solids to Geant4 and FLUKA-like CSG geometry [18, 31]. The FreeCAD/OpenCASCADE interface combined with the Geant4 and FLUKA Python API in PYG4OMETRY will allow for the creation of CAD BREP decomposition algorithms. There are python based CAD modelling tools like cadquery [32] which allow the creation of models using pure Python which should allow the conversion of GDML to STEP.

There is a strong relationship between PYG4OMETRY and Geant4 and to a lesser extent between PYG4OMETRY and FLUKA. PYG4OMETRY can be used as a testing ground for ideas prior to implementation in Geant4 or FLUKA. An example of this is the VTK visualisation system implemented in PYG4OMETRY, which could be used in Geant4 to render Boolean solids which frequently fail in the Geant4 OpenGL viewer, despite being otherwise perfectly valid constructs. This would involve using CGAL meshing in the `G4Polyhedron` class.

PYG4OMETRY is already proving to be a useful tool for geometry conversion, creation and manipulation. There are numerous international researcher and research groups already using the code for their particular applications. The users are focused in accelerator physics, but PYG4OMETRY could find application in any scientific area where MCRT simulations are needed, for example particle physics, space environment and medical physics. The authors welcome contributions, extensions and bug fixes as well as suggestions for larger collaborations.

11. Acknowledgements

The development of Pyg4ometry has received funding from Science and Technology Research council grant "The John Adams Institute for Accelerator Science" ST/P00203X/1 through the John Adams Institute at Royal Holloway, and Royal Holloway Impact Acceleration Account. Gian Luigi D'Alessandro (CERN/JAI) provided geometry and tested multiple conversions of beamlines. Multiple undergraduate and masters students have contributed to the code for their degree project work: Simon Williams, Benjamin Shellswell and Joshua Albrecht. STB has benefited from discussions with and insight of Vasilis Vlachoudis (CERN) on FLUKA geometry and Flair. Cédric Hernalsteens (ULB/CERN) and Alistair Butcher (RHUL) also made useful contributions to the code.

References

[1] C. J. Werner *et al.*, MCNP6.2 Release Notes (2018).

[2] A. Ferrari, P. R. Sala, A. Fassò, J. Ranft, FLUKA: A multi-particle transport code (program version 2005), CERN Yellow Reports: Monographs, CERN, Geneva, 2005. `doi:10.5170/CERN-2005-010`.

[3] T. Böhlen, F. Cerutti, M. Chin, A. Fassò, A. Ferrari, P. Ortega, A. Mairani, P. Sala, G. Smirnov, V. Vlachoudis, The FLUKA code: Developments and Challenges for High Energy and Medical Applications, Nuclear Data Sheets 120 (2014) 211 – 214. `doi:10.1016/j.nds.2014.07.049`.

[4] R. Brun, F. Bruyant, M. Maire, A. McPherson, P. Zanarini, GEANT3 (Sep. 1987).

[5] S. Agostinelli *et al.*, Geant4—a simulation toolkit, Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment 506 (3) (2003) 250–303. `doi:10.1016/S0168-9002(03)01368-8`.

[6] R. Chytracek, J. McCormick, W. Pokorski, G. Santin, Geometry Description Markup Language for physics simulation and analysis applications, IEEE Transactions on Nuclear Science 53 (5) (2006) 2892–2896. `doi:10.1109/TNS.2006.881062`.

[7] The FreeCAD Team, FreeCAD version 0.19 (2020) [cited 16 September 2020].
URL `https://www.freecadweb.org`

[8] Open Cascade S.A.S, OpenCASCADE (2020) [cited 16 September 2020].
URL `https://www.opencascade.com`

[9] W. Schroeder, K. Martin, B. Lorensen, The Visualization Toolkit–An Object-Oriented Approach To 3D Graphics, 4th Edition, Kitware, Inc., 2006.

[10] L. Nevay, S. Boogert, J. Snuverink, A. Abramov, L. Deacon, H. Garcia-Morales, H. Lefebvre, S. Gibson, R. Kwee-Hinzmann, W. Shields, S. Walker, BDSIM: An accelerator tracking code with particle–matter interactions, Computer Physics Communications 252 (2020) 107200. `doi:10.1016/j.cpc.2020.107200`.

[11] R. Brun, F. Rademakers, ROOT: An object oriented data analysis framework, Nucl. Instrum. Meth. A 389 (1997) 81–86. `doi:10.1016/S0168-9002(97)00048-X`.

[12] M. Pinto, P. Gonçalves, GUIMesh: A tool to import STEP geometries into Geant4 via GDML, Computer Physics Communications 239 (2019) 150 – 156. `doi:10.1016/j.cpc.2019.01.024`.

[13] A. Miller, M. Zaake, F. Großmann-ruh, K. Bunte, M. Millinger, G. Drolshagen, Recent extensions of the ESABASE2 / debris impact risk assessment tool, in: Proceedings of the 7th European Conference on Space Debris, 2017.

[14] P. Pourrouquet, J. Thomas, P. Peyrard, R. Ecoffet, G. Rolland, FASTRAD 3.2: Radiation shielding tool with a new Monte Carlo module, in: 2011 IEEE Radiation Effects Data Workshop, 2011, pp. 1–5. `doi:10.1109/redw.2010.6062530`.

[15] P. P. Wilson, T. J. Tautges, J. A. Kraftcheck, B. M. Smith, D. L. Henderson, Acceleration techniques for the direct use of CAD-based geometry in fusion neutronics analysis, Fusion Engineering and Design 85 (10) (2010) 1759 – 1765, proceedings of the Ninth International Symposium on Fusion Nuclear Technology. `doi:10.1016/j.fusengdes.2010.05.030`.

[16] L. Lu, U. Fischer, P. Pereslavtsev, Improved algorithms and advanced features of the CAD to MC conversion tool McCad, Fusion Engineering and Design 89 (9) (2014) 1885 – 1888, proceedings of the 11th International Symposium on Fusion Nuclear Technology-11 (ISFNT-11) Barcelona, Spain, 15-20 September, 2013. `doi:10.1016/j.fusengdes.2014.05.015`.

[17] C. M. Poole, I. Cornelius, J. V. Trapp, C. M. Langton, A CAD Interface for GEANT4, Australasian Physical & Engineering Science in Medicine (Sep. 2012). `doi:10.1007/s13246-012-0159-8`.

[18] X. Wang, J. Li, Z. Wu *et al.*, CMGC: a CAD to Monte Carlo geometry conversion code, Nucl Sci Tech 31 (2020) 82. `doi:10.1007/s41365-020-00793-8`.

[19] F. Rademakers, P. Canal, A. Naumann, O. Couet, L. Moneta, V. Vassilev, S. Linev, D. Piparo, G. GANIS, B. Bellenot, E. Guiraud, G. Amadio, wverkerke, P. Mato, TimurP, M. Tadel, wlav, E. Tejedor, J. Blomer, A. Gheata, S. Hageboeck, S. Roiser, marsupial, S. Wunsch, O. Shadura, A. Bose, CristinaCristescu, X. Valls, R. Isemann, root-project/root: v6.18/02 (Aug. 2019). `doi:10.5281/zenodo.3895860`.

[20] T. Parr, The Definitive ANTLR 4 Reference, 2nd Edition, Pragmatic Bookshelf, 2013.

[21] T. Knip, pycsg (2016) [cited 16 September 2020].
URL `"https://github.com/timknip/pycsg"`

[22] M. Botsch, D. Sieger, P. Moeller, A. Fabri, Surface mesh, in: CGAL User and Reference Manual, 5.0.3 Edition, CGAL Editorial Board, 2020.
URL `https://doc.cgal.org/5.0.3/Manual/packages.html#PkgSurfaceMesh`

[23] S. Loriot, M. Rouxel-Labbé, J. Tournois, I. O. Yaz, Polygon mesh processing, in: CGAL User and Reference Manual, 5.0.3 Edition, CGAL Editorial Board, 2020.
URL `https://doc.cgal.org/5.0.3/Manual/packages.html#PkgPolygonMeshProcessing`

[24] P. Hachenberger, L. Kettner, 3D boolean operations on nef polyhedra, in: CGAL User and Reference Manual, 5.1 Edition, CGAL Editorial Board, 2020.
URL `https://doc.cgal.org/5.1/Manual/packages.html#PkgNef3`

[25] P. Hachenberger, Convex decomposition of polyhedra, in: CGAL User and Reference Manual, 5.1 Edition, CGAL Editorial Board, 2020.
URL `https://doc.cgal.org/5.1/Manual/packages.html#PkgConvexDecomposition3`

[26] A. Meurer, C. P. Smith, M. Paprocki, O. Čertík, S. B. Kirpichev, M. Rocklin, A. Kumar, S. Ivanov, J. K. Moore, S. Singh, T. Rathnayake, S. Vig, B. E. Granger, R. P. Muller, F. Bonazzi, H. Gupta, S. Vats, F. Johansson, F. Pedregosa, M. J. Curry, A. R. Terrel, v. Roučka, A. Saboo, I. Fernando, S. Kulal, R. Cimrman, A. Scopatz, SymPy: symbolic computing in Python, PeerJ Computer Science 3 (2017) e103. `doi:10.7717/peerj-cs.103`.

[27] J. Goldfeather, S. Monar, G. Turk, H. Fuchs, Near real-time CSG rendering using tree normalization and geometric pruning, IEEE Computer Graphics and Applications 9 (3) (1989) 20–28. `doi:10.21236/ada201085`.

[28] The Geant4 Collaboration, The Geant4 Material Database (2020).
URL `http://geant4-userdoc.web.cern.ch/geant4-userdoc/UsersGuides/ForApplicationDeveloper/html/Appendix/materialNames.html#nist-compounds`

[29] H. Si, Tetgen, a delaunay-based quality tetrahedral mesh generator, ACM Trans. Math. Softw. 41 (2) (Feb. 2015). `doi:10.1145/2629697`.

[30] A. Davis, J. Barzilla, A. Ferrari, K. T. Lee, V. Vlachoudis, P. P. Wilson, FluDAG: A CAD based tool for high energy physics, Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment 915 (2019) 65 – 74. `doi:10.1016/j.nima.2018.10.117`.

[31] L. Lu, Y. Qiu, U. Fischer, Improved solid decomposition algorithms for the CAD-to-MC conversion tool McCad, Fusion Engineering and Design 124 (2017) 1269 – 1272. `doi:10.1016/j.fusengdes.2017.02.040`.

[32] CadQuery 2.0 Documentation) [cited 16 September 2020]. URL `https://cadquery.readthedocs.io/en/latest/`