ISBN: 978-3-95450-248-6 ISSN: 2673-5490 doi: 10.18429/JACoW-IPAC2025-WEPS038

PYTHON FLUKA BEAMLINE (PYFLUBL), A PYTHON LIBRARY TO CREATE FLUKA SIMULATIONS OF ACCELERATORS

S. T. Boogert *, Cockcroft Institute, Daresbury Laboratory, Daresbury, UK
L. Nevay, CERN, Geneva, Switzerland

Abstract

FLUKA simulations of beam lines are important for understanding different aspects of accelerators, including beam losses, backgrounds, activation, and shielding. Creating a beam line simulation using FLUKA is a time-consuming and error-prone process. This paper describes a set of Python tools called `pyflubl` (Python FLUKA beam-line) which can create a FLUKA simulation using input from MAD-X, MAD8, Transport or BDSIM. `pyflubl` is based on multiple stable and advanced Python packages created to make BDSIM (Geant4) beamline simulations as simple as possible; these are pymadx (an interface to MAD-X), `pymad8` (an interface to MAD8), `pybdsim` (interface to BDSIM) and most importantly `pyg4ometry` (a geometry engine for Monte Carlo geometry creation). This paper describes `pyflubl` design and implementation and example results for an idealised electron beamline.

INTRODUCTION

There are multiple frameworks or programs to simulate radiation transport (RT) including FLUKA [1, 2], Geant4, MCNP, PHITs etc. Users can of course develop a simulation based on directly creating the input to the RT code. There are existing examples of conversion from an accelerator-like description to a FLUKA model, for example, FLUKA line builder [3] and MadFluka [4]. The authors know of no generally applicable, flexible and openly available FLUKA beam line builder software. Unlike MadFluka and line builder, `pyflubl` can create generic accelerator component geometry to quickly get initial simulations without a complete geometry and material description of a component.

`pyflubl` is inspired by BDSIM [5] which is Geant4 based simulation which is gaining traction within the accelerator community. `pyflubl` is a Python based set of scripts, functions and classes to make the creation of a FLUKA simulation significantly easier. In essence, `pyflubl` is a thin Python interface layer to the input cards required to create a complete FLUKA simulation. `pyflubl` uses multiple established and mature Python packages developed for BDSIM. These include `pyg4ometry` [6], `pymadx` [7], `pymad8` [8], and `pytransport` [9]. `pyflubl` is inspired by and follows a very similar interface to `pybdsim` which is a Python package designed to create the BDSIM input from other input formats. The most important element of `pyflubl` is its use of `pyg4ometry`, which allows the creation and manipulation of FLUKA geometry from within Python.

* stewart.boogert@cockcroft.ac.uk

CODE DESIGN AND IMPLEMENTATION

The core class for creating a simulation is `pyflubl.Builder.Machine`. Once instantiated, elements of type `pyflubl.Builder.Elements` or classes derived from it can be added to the machine where they are appended to the sequence. The calculation of beamline coordinates is done automatically as each component is added. There are also specific convenience methods in `pyflubl.Builder.Machine` to add common elements, so for example `pyflubl.Builder.Machine.AddDrift`. The only required parameters for an element are its name (str) and length (*float*). The keyword arguments for all the `AddXXX` methods in general follow those of BDSIM [5], which were designed to be user-friendly and human-readable without an 8-character limit. Many generic components have already been implemented.

FLUKA uses a global Cartesian coordinate system (X, Y, Z). Placing geometry (bodies, zones and regions), user scoring meshes, beams etc., all with respect to this global coordinate system. However, accelerators are typically best described by a curvilinear coordinate system as shown in Fig. 1 with variables (x, y, s). Placing geometry, scoring meshes, beam etc. in a way which is familiar to accelerator physicists greatly accelerates the development and debugging of FLUKA models as well as simplify the input required.

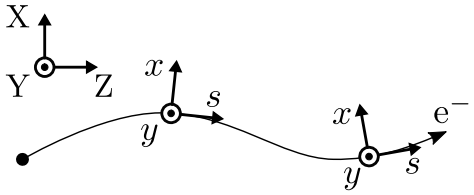

Figure 1: Global and curvilinear coordinate systems used in `pyflubl`.

Typically each accelerator element is defined by a length L and a rotation defined by three angles (θ, ϕ, ψ) with respect to the coordinate system of the previous element. `pyflubl` computes a rotation matrix $\mathbf{M}_N$ from global world coordinates to the N-th component by compounding

$$\mathbf{M}_N = \prod_{i=N}^{1} \mathbf{M}(\theta_i, \phi_i, \psi_i). \quad (1)$$

Similarly the middle position of each component $\mathbf{v}_i$ is the sum of

$$\mathbf{v}_N = \sum_{i=1}^{N-1} L_i \mathbf{M}_i \hat{z} + \frac{L_N}{2} \mathbf{M}_N \hat{z}, \quad (2)$$

ISBN: 978-3-95450-248-6 ISSN: 2673-5490 doi: 10.18429/JACoW-IPAC2025-WEPS038

where L_i is the length of the i-th element and $\hat{z}$ is the unit vector in the z direction[1]. The translation and rotation for each element is then used to place geometry, fields, and scorers.

FLUKA cards not used to define the geometry (e.g. DEFAULTS, TITLE etc.) are wrapped in a thin Python interface. This allows the user or other `pyflubl` functions to include control cards where appropriate. Currently not all FLUKA control cards are implemented but futures releases will be complete.

Objects in `pyflubl` do not have to match exactly to a card in FLUKA. A good example of this is a quadrupole, where calling `Machine.AddQuadrupole(...)` will create the geometry for a quadrupole, but will also add the appropriate magnetic field cards. There is a one-to-one mapping between `Machine.AddScoringMesh(...)` and the FLUKA `USRBDX` but here, the scorer is placed according to the local curvilinear transform. A "sampler" is inspired by BDSIM, where `Machine.AddSampler` adds a thin cubic region and either attaches `USRBIN` or monitors the region in custom Fortran code in the FLUKA simulation.

As a beam line model is built up in `pyflubl` using `AddXXX` methods various "bookkeeping" information is collected. For each component the global to local transformation $\mathbf{M}_N$, and translation $\mathbf{v}_N$ is stored along with the FLUKA REGIONS, FLUKA scoring and field cards. This bookkeeping information is saved as a JSON file for later use. This allows the custom FLUKA user code or subsequent analysis to quickly convert between FLUKA regions and the associated accelerator component names.

GEOMETRY

The manipulation of geometry is the most significant part of the problem of writing a beam-line builder for FLUKA. `pyflubl` depends heavily on `pyg4ometry`. `pyg4ometry` has Python classes which represent Geant4 or FLUKA geometric primitives (e.g. BODIES, ROT-DEFI etc) and the ability to convert between Geant4 and FLUKA descriptions. It is this programmatic interface to the geometry that allows the rapid creation of FLUKA geometry using `pyflubl`. Users can create FLUKA or GDML geometry in `pyg4ometry` and use it `pyflubl` or import external files.

The conversion from Geant4/GDML cannot be completely flexible, i.e. operate on arbitrary geometry. This is because each REGION in FLUKA must be in disjunctive normal form (DNF i.e. the union of convex zones) to be practically usable. Any Boolean expression can be converted to DNF but in many cases the resulting Boolean expression is far too large to be used with FLUKA. To avoid this problem there is a preference for certain FLUKA bodies when performing subtractions, these are RPP/BOX (cuboid), RCC (cylinder), TRC (cone), REC (elliptical cylinder), ARB (arbitrary up to 8 faced solid) and WED (triangular prism). All of these solids can be described by other simpler bodies but would if described in this way would require parentheses which would require expansion and a profusion of terms.

Rectangular (rbend) and sector bends (sbend) rotate the coordinate system. Initially, these are geometrically described as cuboids. `pyg4ometry` is used to detect overlap between different binding volumes for elements. Given an overlap, the end faces can be rotated appropriately.

USAGE

Typically `pyflubl` machines are written by creating a short Python script, for example, that shown in the listing 1. The accelerator is a simple dog-leg and a custom FLUKA file, consisting of a cylindrical target and shield.

Listing 1: A simplified Python script to create a FLUKA beamline. A complete version can be found the project GitHub repository.

```
import pyflubl as _pfbl
m = _pfbl.Builder.Machine(bakeTransforms=
    True)
m.AddDrift(name="d1")
m.AddSBendSplit(name="b1", length=1, angle=
    pi/8)
m.AddDrift(name="d2", length=1)
m.AddSBendSplit(name="b2", length=1, angle=-
    pi/8)
m.AddSamplerPlane(name="s1", length=10e-6)
m.AddCustomFlukaFile(name="c1", length=1,
    geometryFile="./target.inp")
m.AddSamplerPlane(name="s2", length=10e-6)
m.AddDrift(name="d3", length=0.5)
m.Write("IPAC_2025")
```

The output of this code is two files, firstly the FLUKA input file (`IPAC_2025.inp`) and secondly the bookkeeping information (`IPAC_2025.json`). Figure 2 shows the output of the example in FLAIR, where the bends and the custom geometry are clearly visible.

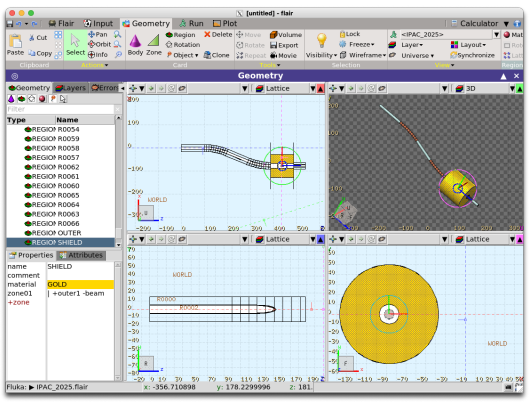

Figure 2: Output of listing 1 displayed in FLAIR.

The conversion from MAD-X uses `pymadx` to load a TWISS file. Creating a FLUKA representation is as simple

[1] This needs to be upgraded to deal with rotations around s and the displacement to the actual trajectory opposed to the chord.

ISBN: 978-3-95450-248-6 ISSN: 2673-5490 doi: 10.18429/JACoW-IPAC2025-WEPS038

as looping through beamline elements in the TWISS file and calling the appropriate `pyflubl.Machine.AddXXX` methods, in a very similar way to the FLUKA linebuilder and MadFLUKA. Given that the creation of a machine is separated from the beam optics files, almost any accelerator code can be very easily interfaced to `pyflubl`. `pyflubl` provides classes (in the module `pyflubl.Analysis`) to run FLUKA and load and process FLUKA output.

SAMPLERS, ENERGY LOSS AND SCORING GRIDS

`pyflubl` provides classes which wrap the scoring commands (USRTRACK, USRCOLL, USRBDX, USRBIN, USRYIELD, SCORE, USERDUMP). Figure 3 shows a plan of the example simulation with a FLUKA USRBIN overlaid. Custom routines has been created to write ROOT output

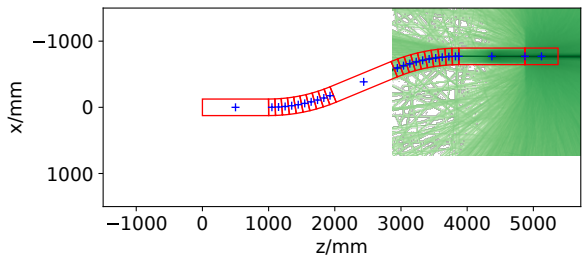

Figure 3: Plan of the example beam line with a FLUKA USRBIN scoring mesh overlaid. The USRBIN is an ALLPART rectangular grid of size 600 × 600 × 600 cm, centred on the target.

from a pyflubl run of FLUKA. Specifically custom routines have been written to initialise (`usrini.f`), store important event data (`mgdraw.f`), end of event (`usreou.f`) and end of run (`usrout.f`). These standard FLUKA user functions immediately call C++ functions, so for example `usrini.f` calls `usrini_c.cxx`. The initialisation routine reads the JSON bookkeeping information and opens a ROOT file. `mgdraw.f` stores boundary crossings and energy deposits, whist end of event performs ROOT IO for a complete event. `usrout.f` then closes the ROOT file and performs a tidying of the run. Positions and directions in global FLUKA are transformed to local element coordinates using the bookkeeping information stored for the simulation. The example in Listing 1 has two samplers labelled `s1` and `s2` and example output is shown in Fig. 4. In a similar fashion to samplers, all energy deposits (including position) can be recorded by FLUKA. The global position of boundary crossings or energy loss (X, Y, Z) can be transformed to S or local element coordinates (x,y,z) using the element bookkeeping information. Figure 5 shows such a loss-map for the example in this paper.

Samplers are currently implemented as a transversely wide and longitudinally thin cubical bodies. Clearly samplers need to be thin, but for very thin samplers ($\mathcal{O}(1)$ μm) a

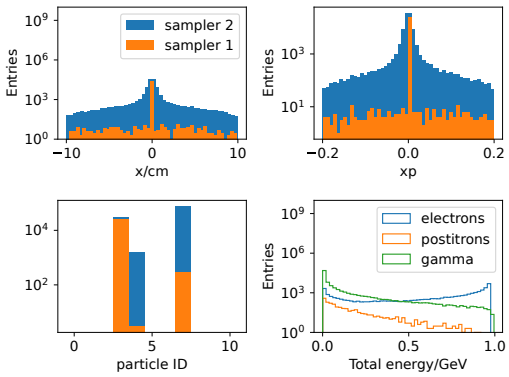

Figure 4: Sample sampler output from before and after the custom FLUKA geometry. Top left : local horizontal position (x). Top right : local angle x'. Bottom left : FLUKA particle ID. Bottom right : energy spectra.

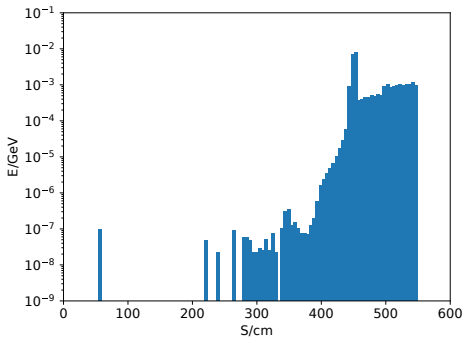

Figure 5: Plot of the energy loss as a function S along the simulation. The target can clearly be seen at ≈ 450 cm.

boundary crossing can be missed by FLUKA. Explicit sampler bodies are not needed and boundary crossings can be stored between any two sequential accelerator REGIONS.

FUTURE WORK AND CONCLUSIONS

`pyflubl` is already a very capable and extendable Python package to rapidly create FLUKA beam line simulations. Future developments will include the ability to split or merge beam lines to better simulate beam injection and extraction, and also development of generic FLUKA geometry for common beamline elements. A complete "accelerator" is presented in this paper, whereas a common use case will be to couple `pyflubl` to external beam tracking codes (e.g. X-suite, elegant etc). This will be added to `pyflubl` via the ability to create disconnected components and then inject or extract particles from pyflubl/FLUKA to exchange with the beam tracking code. The `LATTICE` command also needs to be fully implemented to allow larger machines to be simulated. In addition, a custom source user routine (`source.f`) will simplify the creation of beam particles in accelerator coordinates and coupling to field user routines (`magfld.f` and `elefld.f`) will allow direct comparison with BDSIM. The code is available on GitHub [10] and new users and contributors are always welcome.

ISBN: 978-3-95450-248-6 ISSN: 2673-5490 doi: 10.18429/JACoW-IPAC2025-WEPS038

REFERENCES

[1] C. Ahdida, *et al*, "New Capabilities of the FLUKA Multi-Purpose Code". *Front. Phys.*, vol. 9, Jan. 2021. doi:10.3389/fphy.2021.788253

[2] G. Battistoni, *et al*, "Overview of the FLUKA code." *Annals Of Nuclear Energy*, vol. 82, pp. 10–18, Aug. 2015. doi.org/10.1016/j.anucene.2014.11.007

[3] A. Mereghetti, V. Boccone, F. Cerutti, R. Versaci, and V. Vlachoudis, "The FLUKA LineBuilder and Element DataBase: Tools for Building Complex Models of Accelerator Beam Lines", in *Proc. IPAC'12*, New Orleans, LA, USA, May 2012, paper WEPPD071, pp. 2687–2689.

[4] M. Santana-Leitner, Y. Nosochkov, and T. O. Raubenheimer, "MadFLUKA Beam Line 3D Builder. Simulation of Beam Loss Propagation in Accelerators", in *Proc. IPAC'14*, Dresden, Germany, Jun. 2014, pp. 463–465. doi:10.18429/JACoW-IPAC2014-MOPME040

[5] L.J. Nevay *et al.*, "BDSIM: An accelerator tracking code with particle-matter interactions", *Computer Physics Communications*, vol. 252, pp. 107200, 2020. doi:10.1016/j.cpc.2020.107200

[6] S.D. Walker, A. Abramov, L.J. Nevay, W. Shields, & S.T. Boogert, Pyg4ometry: A Python library for the creation of Monte Carlo radiation transport physical geometries. *Computer Physics Communications*. vol. 272, pp. 108228, Mar. 2022. doi:10.1016/j.cpc.2021.108228

[7] Pymadx code repository, https://bitbucket.org/jairhul/pymadx

[8] Pymad8 code repository, https://bitbucket.org/jairhul/pymad8

[9] Pytransport code repository, https://bitbucket.org/jairhul/pytransport

[10] Pyflubl code repository, https://github.com/bdsim-collaboration/pyflubl