nEXO FLUKA Simulations Manual

Regan Ross

October 16, 2024

Abstract

This document is intented to provide insight into the functioning of the nEXO FLUKA simulations and justifications for choices made in their design. Further, this work should provide a near comprehensive baseline for anyone willing to replicate similar cosmogenic studies for the nEXO collaboration, or even another. FLUKA can be a daunting software to work with given it is written in FORTRAN77, has sparse documentation, and whose source code, without a specific licence, is veiled to the user. However, this proprietary simulation program is fast and rich with built-in features that have been used extensively for decades [1], [2]. Moreover, it is particularly useful for the case of studying cosmogenic muons at several hundred GeV energies and their secondaries [3] [4]. This document will provide an overview and explanation of the FLUKA features enabled for the specific nEXO Outer Detector (OD) case and an overview of the additional user routines, scripts, and containerization procedures that are all ancilliary to the base simulations (but required to run the simulations on the S3DF cluster and perform the more in-depth analyses required by our questions). All references hereafter to the FLUKA manual refer to [5].

List of Figures

Contents

1 Prerequisites

Hello, new user. My goal is to make your job of running this (hopefully not old) code as easy as possible for you to run. We'll get through this together. First, though, there are a few things you'll need. There are a few quirks that will arise. If you're a Mac or Windows user (a regular person), there is an extra step in store for making the right container to hold the dependencies. Follow along closely.

1.1 Dependencies and Preparation

1.1.1 Dependencies

The first order of business is to install the necessary software. Then we'll go over how to use it. Your shopping cart should include:

- **FLUKA License** — In order to use FLUKA, you'll need at least an individual license. FLUKA is not open-source, but if you apply from an academic institution for academic purposes, you shouldn't have an issue. You'll then have access to the binaries, but not the source code.
- **Docker** — The Docker container aleviates a lot of headache. It will be assembled containing much of what you need— including *Flair* and Singularity. It contains the tools necessary to build the Singularity container and edit/make the input files for the simulation. Ideally you don't have to worry about the dependencies for Flair.
- **X11** — You need a program that will permit graphics forwarding from the Docker container. On Mac, I use XQuartz. I'm told that on Windows, you could use Xming. Though I really have no clue whether this would work. Certainly there is a way, but if you're a Windows user, I apologize. This is probably the only problem you'll have to solve on your own here (kidding, there will be more but not related to OS).
- **A cluster** — If you want to run large simulations like those done here, you'll want access to a cluster where you can submit jobs. For this work, the code was deployed on S3DF. You could probably also use something like DRAC (or what was once called "Compute Canada" and makes more sense in my opinion).
- **Python** — While you don't technically need to install Python to run the code on the cluster (the dependencies will be in the container), you should have it installed anyway locally to use some of the analysis tools. You'll also want numpy, scipy, matplotlib, pyROOT, and pandas. You may also want plotly.

1.1.2 Recommendations

You shouldn't strictly *have* to do the following, but these are suggestions for you. If you're going to commit to a project using this work, this advice may help.

- **FLUKA Forum** — Create a FLUKA User Forum account. You will likely end up searching for advice here should you have to deviate from *ordinary* functionality of FLUKA.
- **Version Control** — Be mindful of version control. If you're going to make regular changes to the simulation inputs, you need to be well sure of which outputs correspond to particular inputs. It's probably a good idea to use git/github.
- **Bookmarks** — Bookmark the following sites for future convenience: FLUKA Manual, Older FLUKA Manual, FLUKA 2023 Course.

1.2 Building the Tools

In this section, it's our endeavour to get the Docker container running and make sure graphics are being properly forwarded, then we can edit FLUKA input files, and we can build the Singularity container. This process is definitely technical and uninteresting, but we'll get through it. Before we begin though, make sure that you have installed the dependencies outlined in section 1.1.

1.2.1 Building the Docker Container

Navigate to the `Docker_tools/` directory where you will find the `Dockerfile`. This is a text file that contains within it the pieces necessary for building the Docker container. You don't have to do much work here. Simply run:

```
docker build -t <name> .
```

where `<name>` is something you replace with a judicious choice like "flair_singularity_image" as that's exactly what you'd expect to find in that Docker container. The container will be created. Before you can use *Flair*, you'll need to make sure that graphics forwarding is set up. This might be a bit tricky for you if you're not a Mac user. But if you *are* one, there is a script also in `Docker_tools` that contains what you need. It is called `boot_flair_and_singularity_container.sh`. Have a look inside of it as you may need to modify some things to have it work for you.

If the script works for you, and graphics forwarding works, you should be able to use *Flair* to modify or create an input file. In our case, there is a copy of the input file in the `Docker_tools` directory. Try running:

```
flair nEXO_2024.inp
```

1.2.2 Building the Singularity Container

Yeah yeah yeah there are a lot of "containers" flying around. Let's get this straight: you cannot run FLUKA on your server without putting it first into its own mini isolated environment, so you put it in a *Singularity* container. Now, you unfortunately cannot make such a container outside of a Linux environment (which you don't have), so you make a fake one inside a Docker container that you can use on your own computer. Conveniently, you can also launch *Flair* from this local container. So first you need to make sure you have the necessary FLUKA files, then you need to compile Singularity within the Docker container.

Moving on, you will now create the Singularity container (image) that will be copied onto your remote working directory. How do you do this you ask? Good question. First, make sure you have access to the FLUKA binaries. Once you've been granted access by CERN, you should be able to navigate here to download them. If you can't do this yet, you should wait until you can. You need an account. Follow the instructions on the site. OKAY, if you can access the binaries (namely, if you can download them legally[1]), then you should go ahead and download them. You probably want the most up-to-date version. You definitely want one that will be compiled using a compiler in your Docker image. For instance, the download at the time of writing is: `fluka_4-4.0.x86-Linux-gfor9_amd64.deb`. That is, you want the 64-bit Debian version with gfortran9 compiler. Make sure that the version you download as a `.deb` file is the same as referenced in the singularity definition file. Here it's called `FLUKA.def`.

Okay great. Now you have the right FLUKA binary files. You'll also want to download the cross-section libraries. What are these? These are data about neutron propagation and interactions compiled into a handy little package you can download and deploy in your simulation. The one to use is created by JEFF (not a dude). But the download for FLUKA that you want is here. Download those and put 'em with your FLUKA distribution.

Now we can build the Singularity container. This is an easy process. First, compile the Singularity source within the Docker container. Hint: use the script I've included called `compile_singularity.sh`. Run this script from within your Docker container. Great. Singularity is compiled within the Docker container. Now it's an easy process to build the Singularity container:

```
singularity build FLUKA.sif FLUKA.def
```

[1]There may be many dubious parts in this project, but this doesn't have to be

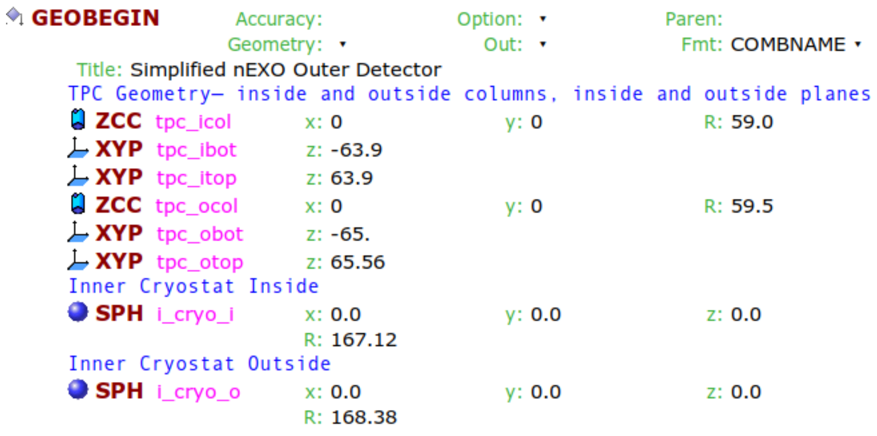
2 The nEXO OD Input File

The input file (".inp") is exactly as it is named— it is the De facto interface between the user and the FLUKA binaries. It contains, among other things, the entire physical configuration of the media through which the particles are transported, the defining characteristics of the impinging beam (for simple sources), options for enabling particular physical processes, definitions of materials, and cards to deploy default FLUKA scoring methods. It is a human-readable ascii file with a very specific format. Each input card in the input file must not contain more than 8 fields each of which has a character limit. This is due (presumably) to constraints in the early development of FLUKA. Programmed in FORTRAN77 which imposes constrains on the length of statements which, in the early days, were written into computers (with very little memory) using paper punch cards. Now, this formatting is an annoying anachronism but it probably does still keep the program slim and fast. Generally though, a user need not worry about writing the input file directly, as there is a great GUI interface to FLUKA called *Flair* which, by the way, is open source and contains all the possible input options [6]. The next section will overview the specific components of the nEXO OD input file and the functions they serve.

It is expected that the user interfaces with the input files via *Flair* as this is probably the easiest way. Errors in the configuration will be unambiguously shown, and it is hard to mess up the format of the input file this way.

2.1 Geometry

Arguably the most important parts of the input file are the cards defining the configuration space— namely the detector and media with respect to the cartesian coordinate system. In FLUKA this is known as the *geometry*. The geometry section of the input card is demarcated by **GEOBEGIN** and **GEOEND** cards. Between these two cards are first cards for various *bodies* which are basic 2D and 3D geometric surfaces, and second, cards for *regions* whose bounds are defined by combinations of geometric *bodies*. For instance, a triad of bodies could be a cylinder whose axis lies on the z-axis, and two planes lying parallel the $x-y$ plane at different zs. A region defined by these bodies could be the 3D cylindrical volume made by the cylinder body capped off on either end by the two planes. This is exactly the type of region representing the nEXO OD and the nEXO TPC.



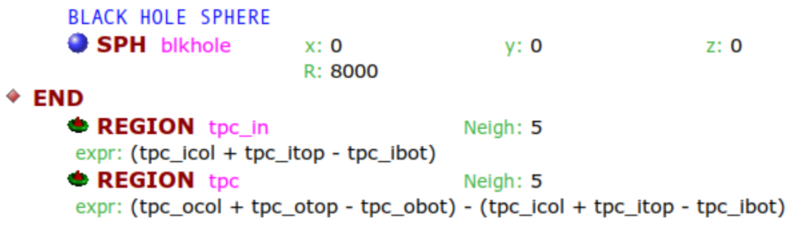
```
GEOBEGIN        Accuracy:              Option: ▾          Paren:
               Geometry: ▾             Out: ▾            Fmt: COMBNAME ▾
  Title: Simplified nEXO Outer Detector
 TPC Geometry- inside and outside columns, inside and outside planes
 ZCC  tpc_icol      x: 0                y: 0               R: 59.0
 XYP  tpc_ibot      z: -63.9
 XYP  tpc_itop      z: 63.9
 ZCC  tpc_ocol      x: 0                y: 0               R: 59.5
 XYP  tpc_obot      z: -65.
 XYP  tpc_otop      z: 65.56
 Inner Cryostat Inside
 SPH  i_cryo_i      x: 0.0              y: 0.0             z: 0.0
                    R: 167.12
 Inner Cryostat Outside
 SPH  i_cryo_o      x: 0.0              y: 0.0             z: 0.0
                    R: 168.38
```

Figure 1: An example of the geometry body declarations in the input file

Figure 1 shows the first part of the geometry declarations in the nEXO input file. We see the first card is the **GEOBEGIN** with the argument *fmt* set to COMBNAME. This tells the FLUKA binaries how to read in the following geometry cards. This is not particularly important. This seems to be the default mode. Text in blue indicates a comment (equivalent to a FORTRAN77 comment in the input file) and body variable names are in pink and are limited to lengths of 8 characters. The names or types of cards are fully capitalized and in maroon. The argument names for each card are written in green and the arguments follow with an 8 character length limit. These colour conventions hold for every other type of input card. Each card can have many arguments, but generally, not all are necessary. In the geometry body cards however, each argument is provided as these arguments are all necessary to uniquely define their respective geometric body. For instance, there is a body called i_cryo_i defined with a SPH card. The name is intented to be shorthand for "inner cryostat inside" and, being a sphere, requires a radius and three coordinates for its location in space. There are many choices for bodies in FLUKA, each of which fairly simple to define.

Regions, as previously discussed, are created with logical combinations of bodies. Imagine the bodies defining surfaces in 3D, and the regions being the volumes they surround. Following the set of cards defining the bodies, there is an **END** card, then the region cards as shown in figure 2, finally there is the **GEOEND** card. Each region must be assigned **one** material— we'll get to material assignments later. For example, the region inside the nEXO OD that is full of water would be defined by the OD inside body minus the outer cryostat outside body leaving the configuration of a cylinder with a spherical hole in its volume— the entirety of this region is to be water.

Looking at figure 2, we see the final body declaration followed by the first few region declarations in the nEXO input file. The final body here is important— each FLUKA configuration defines its physical boundaries by assigning material *blackhole* to its outermost surface- hence the name of the sphere. The first



```
BLACK HOLE SPHERE
SPH blkhole        x: 0              y: 0              z: 0
                   R: 8000
END
REGION tpc_in                     Neigh: 5
expr: (tpc_icol + tpc_itop - tpc_ibot)
REGION tpc                        Neigh: 5
expr: (tpc_ocol + tpc_otop - tpc_obot) - (tpc_icol + tpc_itop - tpc_ibot)
```

Figure 2: An example of the geometry region declarations in the input file

region we have defined is tpc_in will be the inside of the TPC cylinder— the part filled with liquid xenon. While no number is assigned by the user to the region, FLUKA assigns it number 1 as it is the first defined- this will be important later. The construction of the region is in the expr argument given by "(tpc_icol + tpc_itop − tpc_ibot)". We first have *tpc_icol* which is the inside cylinder (along z) of the TPC which alone is unbound and spans the entire z-axis. To the cylinder we add *tpc_itop* (xy-plane) which sets the upper bound, and subtract *tpc_ibot* (xy-plane) which sets the lower bound. This process is similar for all other region declarations. Unfortunately, one may not reference regions by name in region expressions, only bodies; hence the following region declaration of the tpc which is intended to be the material composing the TPC. The Neigh argument is an integer which defines "neighbourhoods" relating regions to each other. This can perhaps be deployed for larger, more complex geometries but given the simplicity of the nEXO OD geometry and the lack of thorough documentation of this feature, it has been set to the default value of 5 for each region. Once more, the **GEOEND** card (not shown) brings us to the end of the geometry card section.

2.1.1 The Chosen Configuration

The simulations performed here deployed a simplified geometry of nEXO. There were no PMTs included, no complicated internal structure and no contoured cavernous room around the detector. It is simply a large stainless steel cylinder surrounded by norite rock [7] and full of water containing the inner and outer cryostats and the copper TPC filled with liquid Xenon-136. That's it. The masses for the TPC and the cupric TPC shell were chosen to correspond with parallel GEANT4 simulations. This is an important consideration for scoring activation of particular isotopes. The muons are propagated through at least 15m of rock before reaching the detector, and the rock surrounding the OD laterally is 20m, and below it, 5m. Where possible, the measurements for the configuration were taken from the "nEXO Preliminary Design Report" except for the size of the OD which has been adjusted to meet the more recent specification.

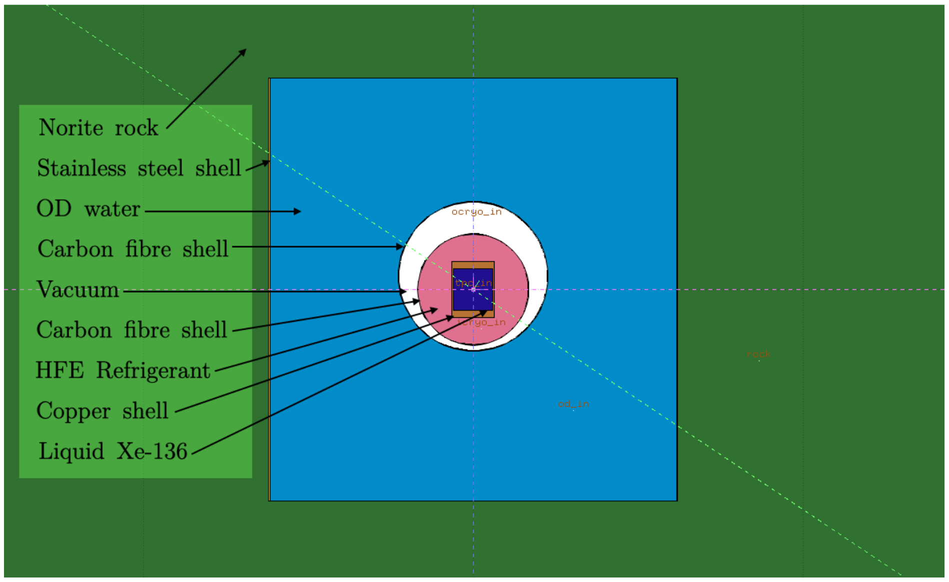
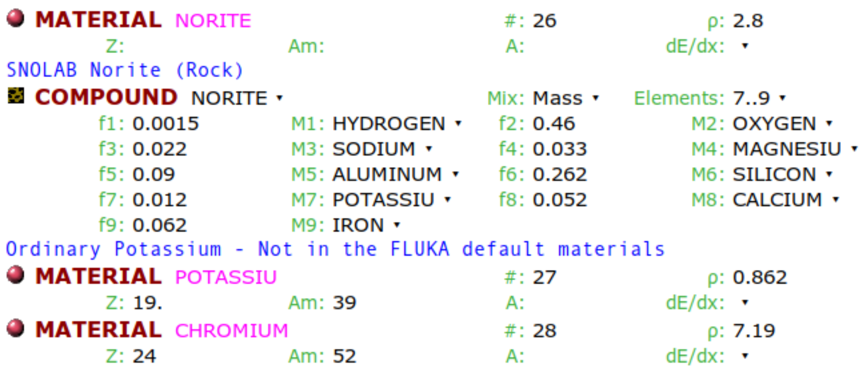


Figure 3: A 2D projection of the nEXO geometry used in the FLUKA simulations

2.2 Media

After *geometry* the most natural section to discuss in the input card is the *Media* section. In this section, materials are defined if they aren't in the FLUKA default material database and they're assigned to regions. There are four main types of cards in this section: MATERIAL, COMPOUND, ASSIGNMA, and LOW-PWXS.



```
● MATERIAL NORITE                              #: 26              ρ: 2.8
        Z:              Am:                    A:             dE/dx: ▾
SNOLAB Norite (Rock)
▣ COMPOUND NORITE ▾                         Mix: Mass ▾    Elements: 7..9 ▾
        f1: 0.0015      M1: HYDROGEN ▾        f2: 0.46            M2: OXYGEN ▾
        f3: 0.022       M3: SODIUM ▾          f4: 0.033           M4: MAGNESIU ▾
        f5: 0.09        M5: ALUMINUM ▾        f6: 0.262           M6: SILICON ▾
        f7: 0.012       M7: POTASSIU ▾        f8: 0.052           M8: CALCIUM ▾
        f9: 0.062       M9: IRON ▾
Ordinary Potassium - Not in the FLUKA default materials
● MATERIAL POTASSIU                            #: 27              ρ: 0.862
        Z: 19.          Am: 39                 A:             dE/dx: ▾
● MATERIAL CHROMIUM                            #: 28              ρ: 7.19
        Z: 24           Am: 52                 A:             dE/dx: ▾
```

Figure 4: A subset of the media section in the input file

First, referring to figure 4 and describe the first card in the media section. It is a MATERIAL card with variable name NORITE. This card declares a new material called NORITE which is given a density, and a user-chosen material number for later reference [2]. Norite rock is an abundant type of igneous rock around the Sudbury basin [8]. It is a mixture of various elements and must therefore be defined with the subsequent COMPOUND card. The arguments for this card are the component materials of the compound and their respective fractions by mass, volume, or atom abundance. These arguments should be very clear, the odd one out is Elements which simply allows for the resizing of the compound card to accomodate more or fewer elements. The following cards, POTASSIU and CHROMIUM are necessary to define as they are not in the default FLUKA materials list. Given that these are elemental, we must provide the arguments of atomic number (Z), the atomic mass (Am) in g/mol, and the atomic mass number (A) which is assumed to be the most abundant for the given (Z) if left unspecified. Of course ρ is the density, and dE/dx allows to select another material to use for the case of ionization— we do not use this feature.

Once all the materials are defined they are then assigned to regions with the ASSIGNMA cards. This card a list of arguments including the material to be assigned and the regions it will be assigned to. One of these cards can assign a material to several regions, however, the regions had to have been declared in some regular order for this to work for multiple regions. That is, a user might wish to have every region from the first to last of N regions to be full of water. In this case, the card would have arguments Reg: $=$ 1 and to Reg: $= N$. Alternatively, a user can assign a material to every k^{th} region in the range $[1, N]$ by setting Step: $= 3$. This seems like a strange and uncomfortable way to do things because any modification to the region declarations can totally offset the material assignments. Nevertheless, it is how it works. In the "nEXO_OD.inp" input file, each region is assigned a material separately and by name. This issue ought not occur.

2.2.1 Specific Materials

Now that the media input options have been discussed, we will overview some of the materials used in the "nEXO_OD.inp" input file as these may not be an exact representation of the material structure of nEXO. First, the rock surrounding the geometry is of a particular variety that is very common around SNOLAB:

[2]The numbers for user-defined materials can't start at 1, they proceed from the last number of the native FLUKA materials list (25)

norite [9]. Its composition is shown in the table below. Then, another compound used in the configuration is HFE, the refrigerant in the inner cryostat whose composition is shown in the table below.

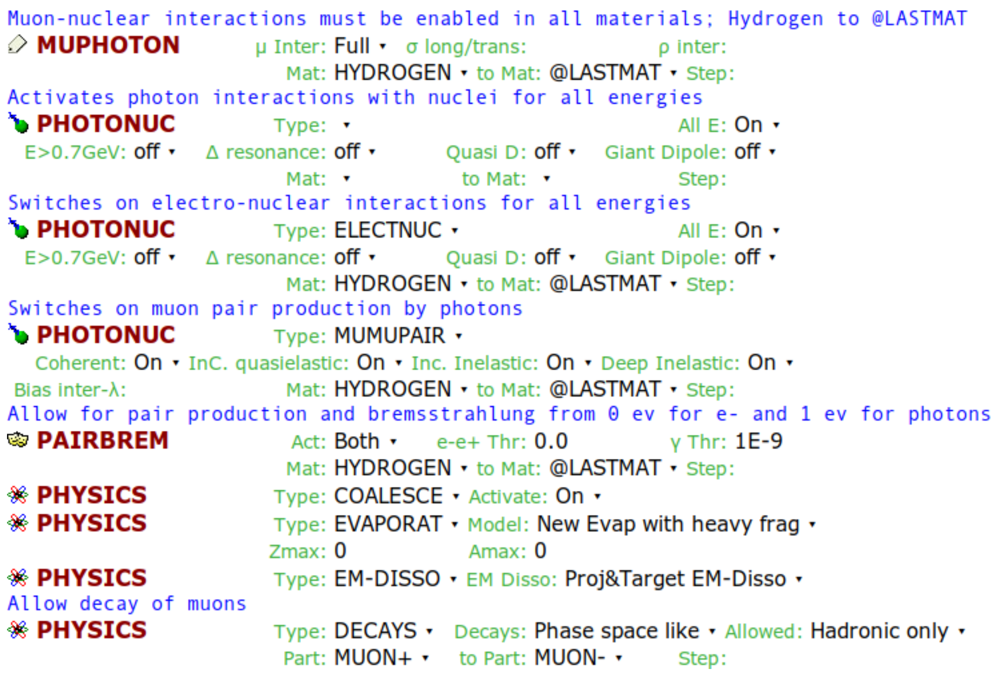
Norite $\rho = 2.894$ g/cm^3	
Element	Atomic Composition (%)
O	46.0
Si	26.2
Al	9.0
Fe	6.2
Ca	5.2
Mg	3.3
Na	2.2
K	1.2
Ti	0.5
H	0.15
Mn	0.1
C	0.04

HFE $\rho \approx 1.5$ g/ml	
Element	Atomic Composition (%)
F	46.6
C	26.7
H	20.0
O	6.7

LOW-PWXS cards could be in either the "Media" section or the "Physics" section. These indicate how neutron transport ought to be treated in a given medium. If the user desires point-wise low energy neutron transport for the entire simulation, a LOW-PWXS card without any arguments is the solution. Otherwise a user may specify particular media within which it will be activated. Basically the LOW-PWXS card will accept a material as an argument and activate this neutron treatment for regions containing that material. More on this will be presented in the "Physics" section.

2.3 Physics and Particle Transport

By default, FLUKA transports neutrons down to 1×10^{-5} eV or “thermal energies”. Some other physics processes have to be enabled explicitly using various FLUKA cards. For instance, photo-nuclear events between muons and nuclei are not enabled by default. Figure 5 below shows the set of physics cards currently used in the nEXO input file. In all cards with the arguments “Mat:” and “to Mat:”, the respective arguments are “HYDROGEN” and “@LASTMAT”. This simply means that these processes are enabled in all materials from the first to last FLUKA material in the file. Namely, hydrogen has FLUKA material number 1, and the materials used in section 2.2 extend to some arbitrary highest integer— @LASTMAT.



```
Muon-nuclear interactions must be enabled in all materials; Hydrogen to @LASTMAT
MUPHOTON          μ Inter: Full ▾  σ long/trans:             ρ inter:
                      Mat: HYDROGEN ▾ to Mat: @LASTMAT ▾ Step:
Activates photon interactions with nuclei for all energies
PHOTONUC             Type: ▾                              All E: On ▾
 E>0.7GeV: off ▾  Δ resonance: off ▾    Quasi D: off ▾  Giant Dipole: off ▾
                      Mat: ▾          to Mat: ▾           Step:
Switches on electro-nuclear interactions for all energies
PHOTONUC             Type: ELECTNUC ▾                     All E: On ▾
 E>0.7GeV: off ▾  Δ resonance: off ▾    Quasi D: off ▾  Giant Dipole: off ▾
                      Mat: HYDROGEN ▾ to Mat: @LASTMAT ▾ Step:
Switches on muon pair production by photons
PHOTONUC             Type: MUMUPAIR ▾
  Coherent: On ▾ InC. quasielastic: On ▾ Inc. Inelastic: On ▾ Deep Inelastic: On ▾
 Bias inter-λ:        Mat: HYDROGEN ▾ to Mat: @LASTMAT ▾ Step:
Allow for pair production and bremsstrahlung from 0 ev for e- and 1 ev for photons
PAIRBREM              Act: Both ▾   e-e+ Thr: 0.0          γ Thr: 1E-9
                      Mat: HYDROGEN ▾ to Mat: @LASTMAT ▾ Step:
PHYSICS              Type: COALESCE ▾ Activate: On ▾
PHYSICS              Type: EVAPORAT ▾ Model: New Evap with heavy frag ▾
                    Zmax: 0            Amax: 0
PHYSICS              Type: EM-DISSO ▾ EM Disso: Proj&Target EM-Disso ▾
Allow decay of muons
PHYSICS              Type: DECAYS ▾  Decays: Phase space like ▾ Allowed: Hadronic only ▾
                     Part: MUON+ ▾   to Part: MUON- ▾       Step:
```

Figure 5: The set of physics cards in the nEXO input file

MUPHOTON This is used to enable muon photonuclear interactions and to allow for the production of all secondary hadrons by muons. In this card, the arguments not provided are reserved for code development and not in use.

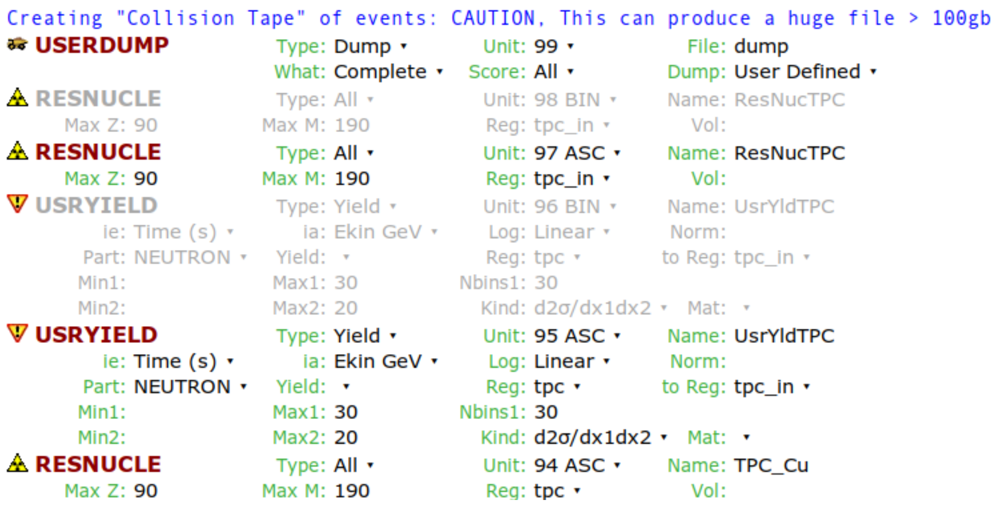
PHOTONUC These cards are used to activate gamma interactions with nuclei. The first in the series requires only the “All E” argument to be ON to enable photonuclear interactions over all energies in FLUKA. The second two PHOTONUC cards are used to activate electronuclear interactions and muon-muon pair production respectively.

PAIRBREM This card controls simulation of pair production and bremsstrahlung by high-energy muons, charged hadrons and light ions (up to alpha’s). Here, both bremsstrahlung and pair production are activated with the electron energy threshold set to 0 which corresponds to the lowest FLUKA limits, and the γ threshold set to 1 eV.

PHYSICS There are four physics cards here. The first of which enables coalescence; high energy light fragments can be produced by a mechanism joining together nucleons that are near in the phase space. This card ensures this feature of the simulation is enabled. It is important for when residual nuclei are being scored like the activation of Xe-136 for instance. The second PHYSICS card enables the emission of heavy nuclear fragments (also important for residucl nuclei scoring). Next, there is the EM-DISSO card. When a photon in the range of a nuclear resonance impinges upon a nucleus, it can create an excitation known as a giant dipole resonance. This decays with the emission of a nucleon— this process is activated by EM-DISSO. Finally, the decay of hadrons is permitted with the DECAY card.

2.4 Scoring

Also within the input file are some scoring options. In FLUKA, the *measurables* are *scored* or synonymously, *estimated*. Nevertheless, the scoring section of the input file contains a few cards activating the builtin scoring features of FLUKA, and another single card that requests the custom scoring from within the “mgdraw.f” file.



```
Creating "Collision Tape" of events: CAUTION, This can produce a huge file > 100gb
USERDUMP             Type: Dump ▾          Unit: 99 ▾          File: dump
                     What: Complete ▾      Score: All ▾        Dump: User Defined ▾
RESNUCLE             Type: All ▾           Unit: 98 BIN ▾      Name: ResNucTPC
          Max Z: 90  Max M: 190            Reg: tpc_in ▾       Vol:
RESNUCLE             Type: All ▾           Unit: 97 ASC ▾      Name: ResNucTPC
          Max Z: 90  Max M: 190            Reg: tpc_in ▾       Vol:
USRYIELD             Type: Yield ▾         Unit: 96 BIN ▾      Name: UsrYldTPC
      ie: Time (s) ▾ ia: Ekin GeV ▾        Log: Linear ▾       Norm:
   Part: NEUTRON ▾   Yield: ▾              Reg: tpc ▾          to Reg: tpc_in ▾
   Min1:             Max1: 30              Nbins1: 30
   Min2:             Max2: 20              Kind: d2σ/dx1dx2 ▾  Mat: ▾
USRYIELD             Type: Yield ▾         Unit: 95 ASC ▾      Name: UsrYldTPC
      ie: Time (s) ▾ ia: Ekin GeV ▾        Log: Linear ▾       Norm:
   Part: NEUTRON ▾   Yield: ▾              Reg: tpc ▾          to Reg: tpc_in ▾
   Min1:             Max1: 30              Nbins1: 30
   Min2:             Max2: 20              Kind: d2σ/dx1dx2 ▾  Mat: ▾
RESNUCLE             Type: All ▾           Unit: 94 ASC ▾      Name: TPC_Cu
          Max Z: 90  Max M: 190            Reg: tpc ▾          Vol:
```

Figure 6: The set of scoring cards in the nEXO input file

In figure 6 there is the entirety of the scoring section in the input card. The USERDUMP card specifies that the “mgdraw.f” routine gets called to enable the customized scoring therein. However, one has to be careful as the default “mgdraw.f” file can save **much** more data than is necessary resulting in files exceeding tens of gigabytes in size— hence the comment above the card.

The RESNUCLE cards tell FLUKA to print out ascii (human-readable) files which contain the counts of residual nuclei in a given region. Seen here we have cards for the TPC liquid xenon and also the copper TPC shell.

Data from the other scoring cards are not currently used but they can be used to score differential fluence (with respect to energy, time, etc...). In essence all one needs for the nEXO scoring are the residual nuclei scoring cards for activation, and the USERDUMP which sends the data to the custom routine that takes care of the rest of the neutron data.

3 Muon Source

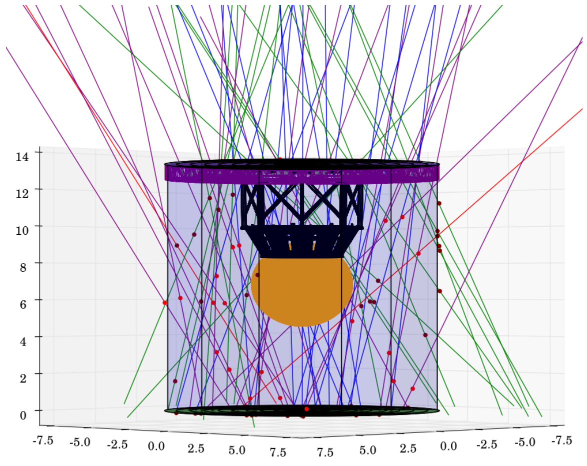


Figure 7: A visualization of muons passing through a more sophisticated nEXO geometry

In FLUKA, one may define a beam-like source using the built-in BEAM cards. However, if one wishes to simulate a more novel or complex series of source particles, a different user routine can do the trick. In the FLUKA literature and on the forums, this more complex source file is called “source_newgen.f”. In the simulation files here, it is called “muon_from_file.f” as this makes more sense. This user routine comes with a set of built-in functions allowing a sophisticated user to sample energies from various functions (Maxwell-Boltzmann for instance) and sample various initial positions. In our case, however, we only make use of a pair of features— the *real* source generation is done with a more easily readable and modifiable set of python functions. To learn about how the muons are simulated, please see [10].

In “muon_from_file.f” particles are read from a phase-space file (we’ll get to that) and passed to FLUKA where they are propogated through the configuration and scored as requested. This is done using the built in routine “read_phase_space_file”. In this user routine, a file argument is provided pointing to the list of muons (called a phase space file). This is kept in the “sim_files” directory and created with each run using the “run_simulation.py” script (for now). The files are created externally using the muon_functions.py module in the “sim_tools” directory. Basically, the file contains a list of muons with attributes listed as comma-separated values in the following order:

$$\text{Particle No.} \quad \text{Energy [GeV]} \quad X_0 \text{ [m]} \quad Y_0 \text{ [m]} \quad Z_0 \text{ [m]} \quad \cos(x) \quad \cos(y) \quad -\cos(z) \quad \text{Weight}$$

or equivalently in FLUKA language:

JTRACK ETRACK XSCO YSCO ZSCO CXTRCK CYTRCK CZTRCK WTRACK

The particle number (JTRACK) is either 10 (positive muon) or 11 (negative muon). These variables will be further discussed in the section on customized scoring here: 4.1.

4 Custom Outputs

4.1 mgdraw.f

While for many general purposes FLUKA comes equipped with adequate features, more particular circumstances may call for probing the particle stack directly. This is done through external user routines that are only deployed when explicitly linked at compile time. For instance, to determine the distances of closest approach for muons that produce neutrons in the TPC region, access to these data is necessary. For the most part, all the features of FLUKA can be accessed with customized code through the "mgdraw.f" user routine. For the nEXO simulations, only a small part of the "mgdraw.f" capabilities were deployed with a two primary goals in the readout:

1. Retrieve neutron data for neutrons in the configuration (within the TPC or internal to the OD)
2. For each logged neutron, save its the parent muon data

Specifically the data saved were the following:

Particle Stack Variable	**Definition**	**Data Type**
ICODE	event type being logged (for which the current particle is a parent)	integer
NCASE	number of the current primary $\in [1, N]$	integer
JTRACK	FLUKA particle ID	integer
MREG	region number where the particle was scored	integer
LTRACK	generation of the particle (primaries have LTRACK = 1)	integer
ETRACK	total energy of the particle (rest + kinetic)	float
(X or Y or Z)SCO	geometric coordinate where the particle was scored	float
C(X or Y or Z)TRCK	direction cosines for the respective coordinate axis	float

In this instance, saving the data implied telling FLUKA to write the data to a particular output channel in the form of a human-readable ascii file. Later on however, these data were transcribed into HDF5 files for smaller storage sizes and for faster parsing. An overview of this process will be presented in the section on analysis.

4.2 Activation

The activation data can also be scored in a custom fashion, however, when possible it is easier to use the builtin FLUKA features. The RESNUCLE output is an array consisting of 10 columns and N rows (as most FLUKA ascii output files are) that must be re-arranged to be in Z-A format.

5 Some Results

The primary motivation for these FLUKA simulations was to validate the GEANT4 simulations that had previously been performed to estimate activation due to the cosmogenic muons. Other connections were investigated including the distances of closest approach (impact parameters) of the muons producing neutrons in the vicinity of the nEXO OD. There had been a connection established between the impact parameters and the propensity of a muon to produce neutrons within or entering the OD or the TPC volume; muons with large impact parameters were unlikely to contribute to activation.

For the simulations in this work, the impact parameters were calculated upon the random generation of the muons in the external python module. These data, along with all the other muon data, were saved and added to the ".hdf5" files after the simulations were performed. The activation data were scored with FLUKA's RESNUCLEi function that counts "residual nuclei" in a chosen region. The analyses were entirely performed externally using various python libraries.

5.1 Activation

The activation was scored for the TPC internal liquid xenon volume and its copper shell. Production rates were scored for a range of isotopes including xenon-137 and copper isotopes in the TPC shell. Over 150 simulated years, the mean yearly count of Xe-137 was 22 atoms. Figure 8 displays tables of isotopes for both the TPC xenon region and the copper shell.

5.2 Impact Parameters

As mentioned previously, another parameter of interest with respect to the muons were the distances of closest approach; we refer to these as the impact parameters of the muons. The findings were particularly interesting. For the same 150-year batch of muons, all the muons responsible for neutrons in the TPC passed through the OD volume and would therefore be detectable. These results are shown in figure 9. Bear in mind the maximum possible impact parameter for a muon in the OD is just shy of 9 meters, and the smallest impact parameter a muon can have without passing through the OD is the OD radius- 6.172 m. The maximum possible impact parameter from the simulations was nearly 12 meters; there are no muons beyond 11 meter impact parameters producing neutrons in the OD- let alone the TPC.

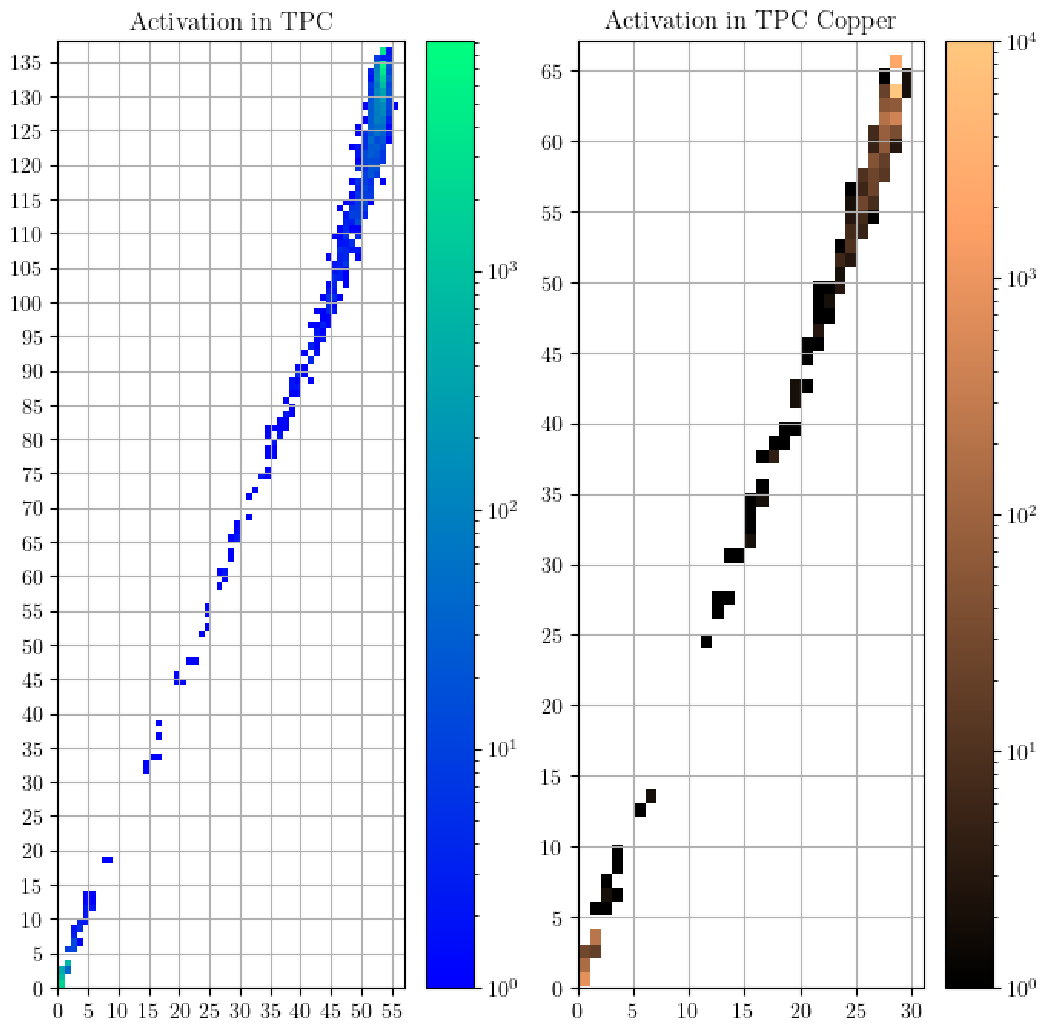


Figure 8: A plot of the residual nuclei counted in their respective volumes. 150 years of data presented.

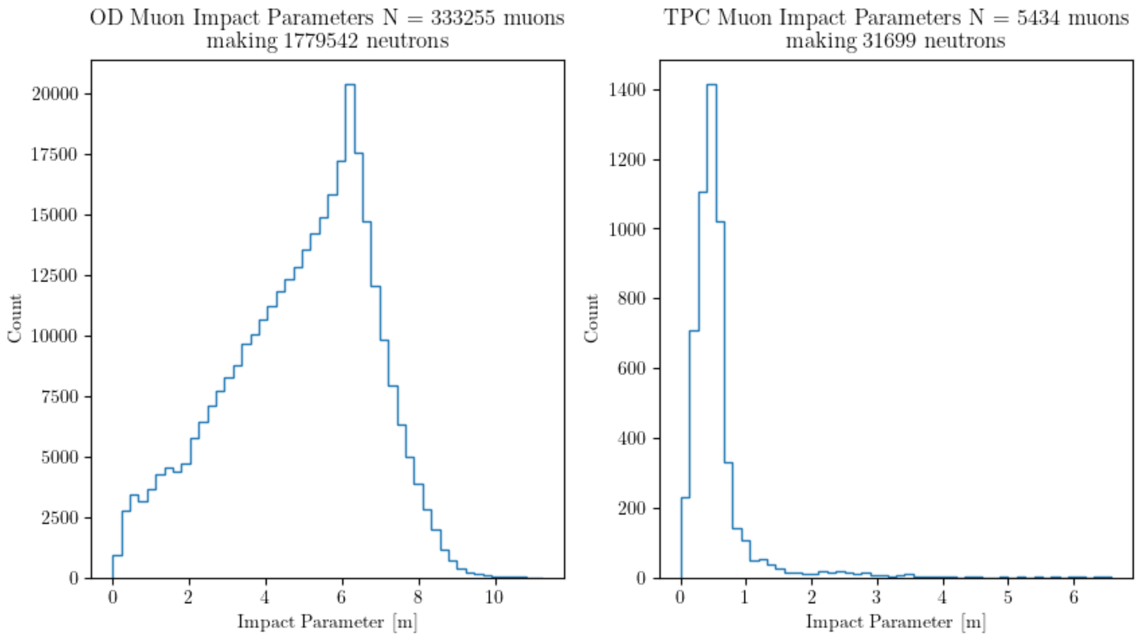


Figure 9: A plot of the the impact parameters of muons producing neutrons in either the TPC (left) or the OD (right)

Appendix A Containerization with Singularity

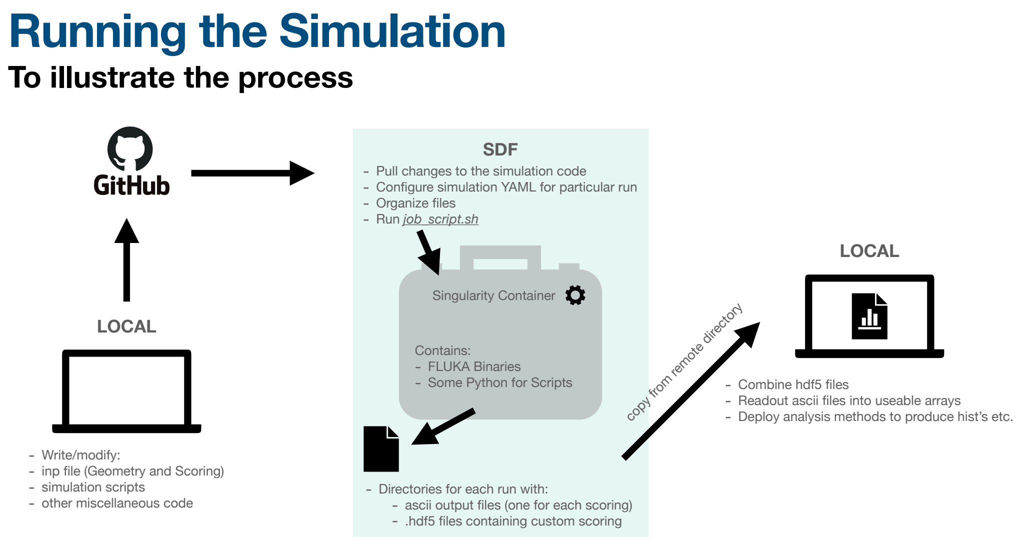

Running the Simulation
To illustrate the process
GitHub
LOCAL
- Write/modify:
- inp file (Geometry and Scoring)
- simulation scripts
- other miscellaneous code
SDF
- Pull changes to the simulation code
- Configure simulation YAML for particular run
- Organize files
- Run job_script.sh
Singularity Container
Contains:
- FLUKA Binaries
- Some Python for Scripts
- Directories for each run with:
- ascii output files (one for each scoring)
- .hdf5 files containing custom scoring
copy from remote directory
LOCAL
- Combine hdf5 files
- Readout ascii files into useable arrays
- Deploy analysis methods to produce hist's etc.

Figure 10: A visualization of the remote simulation and development process

In order to deploy FLUKA on a remote cluster, SDF for instance, it must be packaged up in a *container*. SDF recommends using Singularity which is essentially designed for containerizing software with its dependencies to maximize portability. Naturally, there are a few caveats to containerizing the simulation. The first, rather inconvenient constraint is that Singularity containers must (at the time of writing) be built from within a Linux OS. Therefore, to build a Singularity image using a Mac, one must use a linux VM. This requires, among other pieces, Vagrant which is used to run the Linux virtual machine. There's no use in giving a comprehensive guide to installing Singularity, Vagrant and creating the entire Singularity image here; these softwares are actively being developed and improved. Readers are referred to the Sylabs Singularity documentation: `https://docs.sylabs.io/guides/3.11/admin-guide/installation.html#mac`.

The definition file "`.def`" for the Singularity container is in the Github repository. Should the reader wish to reproduce or modify the container, they'll need at minimum a CERN FLUKA licence in order to download the FLUKA binaries and they'll need to be placed in the directory where the container will be created along with any other files and software that cannot be installed separately from within the "`.def`" file.

Put simply, the Singularity container holds some Python packages, the FLUKA binaries and an Evaluated Nuclear Data File (ENDF) database holding cross sections read by FLUKA when transporting neutrons.

Appendix B Submitting Jobs with SLURM

When running FLUKA simulations on a computing cluster such as SDF, one has to submit *jobs* to be performed. These are the tasks that you are requesting the computer perform; for instance, you may submit a job of simulating 5000 muons in your geometry. The simulations performed here were generally submitted in batches— many jobs at once. Each job would require a node (a CPU) and these would each churn through two-months worth of muons through the OD simultaneously. The SDF cluster uses SLURM to allocate jobs and thus we will briefly discuss SLURM here as it pertains to running the FLUKA simulations. Following this section is an example of a job submission script used to submit jobs to the SDF cluster.

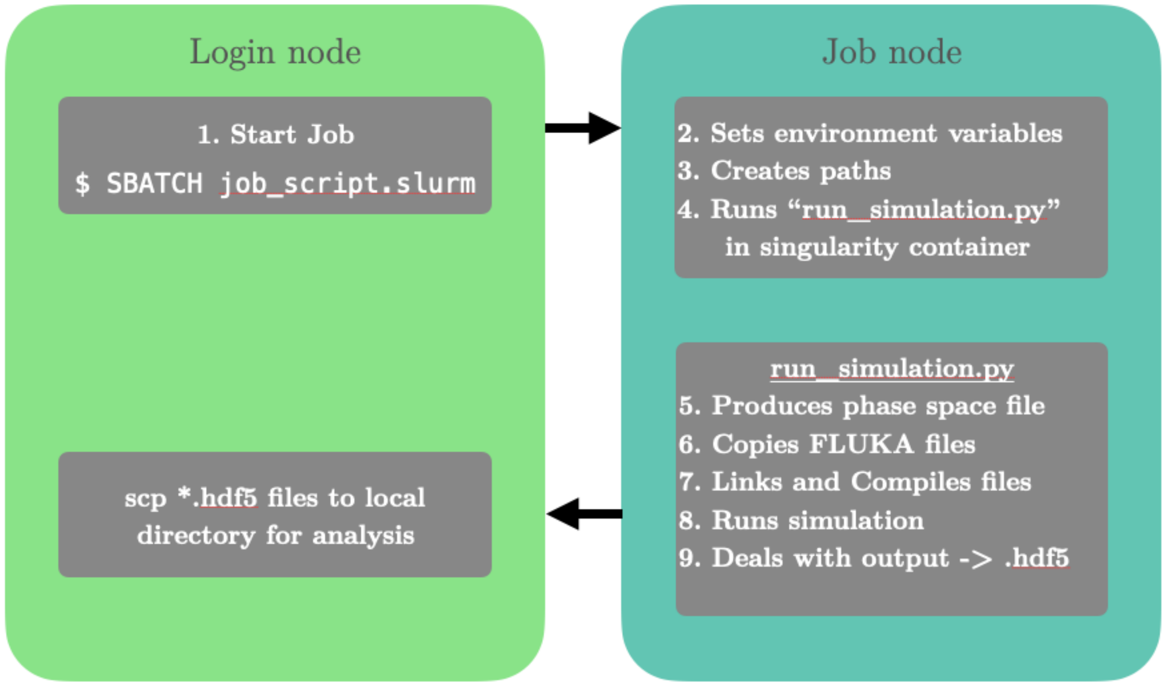
In the following script, the first line tells the interpreter to use *bash* to interpret the uncommented lines. Bear in mind, this script is not run in the conventional way by entering its path into the terminal, but by typing *SBATCH* followed by the path to the script. Now, the next commented (#) lines beginning with SBATCH are commands read in exclusively by SLURM. These request the specific resources for each node in the array (we're requesting an array of jobs here— see line 14). Each node will have 3 hours and 30 minutes to complete the task before the job will be ended by SLURM (line 13). Then for each one of these nodes, the following bash commands are executed (lines 19 onward).

```
1  #!/bin/bash
2
3  #################################################
4  #                   SLURM
5  #################################################
6
7  #SBATCH --partition=shared
8  #SBATCH --job-name=fluka_sims               # a short name for your job
9  #SBATCH --output=simrun-%A-%a.out           # stdout file
10 #SBATCH --error=simrun-%A-%a.err            # stderr file
11 #SBATCH --cpus-per-task=1                   # cpu-cores per task (>1 if multi-threaded tasks)
12 #SBATCH --mem-per-cpu=10G                   # memory per cpu-core (4G is default)
13 #SBATCH --time=03:30:00                     # total run time limit (HH:MM:SS)
14 #SBATCH --array=0-50                        # job array with index values 0, 1, 2, 3, 4
15
16 #################################################
17 #                  Variables
18 #################################################
19
20 output_dir="run"$SLURM_ARRAY_JOB_ID
21
22 rand=$RANDOM
23 path_to_sim='/gpfs/slac/staas/fs1/g/exo/exo_data8/exo_data/users/rross/flukaSims/'
24 image=$path_to_sim"fluka_nEXO.sif"
25 job_num=$SLURM_ARRAY_JOB_ID         # An integer number representing this array job submission
26 task_num=$SLURM_ARRAY_TASK_ID       # An integer within the array range above representing which part o
27 fluka_path='/usr/local/fluka/bin/'
28
29 muon_source=$path_to_sim'muon_from_file.f'
30 mgdraw=$path_to_sim'mgdraw_neutron_count.f'
31 input=$path_to_sim'nEXO_OD.inp'
32
33 new_muon_source=$path_to_sim'musource'$rand'.f'
34 new_muon_source_o=$path_to_sim'musource'$rand'.o'
35 new_mgdraw=$path_to_sim'mgdrw'$rand'.f'
36 new_mgdraw_o=$path_to_sim'mgdrw'$rand'.o'
```

```
new_input=$path_to_sim'input'$rand'.inp'
exe_name=path_to_sim'exe'$rand'.exe'

#################################################
#    REMEMBER: These do not run sequentially.
################################################

mkdir $output_dir
mkdir $output_dir/subrun$SLURM_ARRAY_TASK_ID

#################################################
#         Printing meta data to stdout
################################################

cd $path_to_sim
date
echo "Executing on the machine:" $(hostname)
echo "System random number used to seed the sim:" $rand
echo "Running Simulation!"
echo ; echo ; echo ; echo "COMPILING JARGON:"
echo

singularity exec -B /gpfs $image python $path_to_sim/run_simulation.py -s $rand

mv *$rand* $output_dir/subrun$SLURM_ARRAY_TASK_ID
mv *.hdf5 $output_dir/
```

Appendix C Running the simulation

Other users of this code may choose to make adjustments to better suit their needs, but for now, this is how the code is generally run. Not every command will be reviewed here, but a general description of the procedure will be provided.



Login node
1. Start Job
$ SBATCH job_script.slurm
scp *.hdf5 files to local directory for analysis
Job node
2. Sets environment variables
3. Creates paths
4. Runs "run_simulation.py" in singularity container
run_simulation.py
5. Produces phase space file
6. Copies FLUKA files
7. Links and Compiles files
8. Runs simulation
9. Deals with output -> .hdf5

Figure 11: A simple display of what is happening when the code is run

Preparing Inputs First, the user must be sure of how many muons ought to be simulated and what size the targetting region should be. The yaml file should be configured to match these specifications. Apt file names should be assigned for outputs. These can be changed from within the SLURM file.

Running The only remaining step in running the simulation is to execute the SLURM file; this requests the necessary resources to run a batch of simulations simultaneously and moves the output to the specified paths:

```
$ sbatch job_script.slurm
```

Analyzing the HDF5 files Now the user is free to analyze the resultant hdf5 files which contain the necessary FLUKA outputs.

Appendix D HDF5 File Structure

It's in the name; HDF5 (hierarchical data format) files have a hierarchy. Within the file there are groups, within the groups there are datasets. Within the datasets there are data. Figure 12 shows the current data format for the h5 output files.

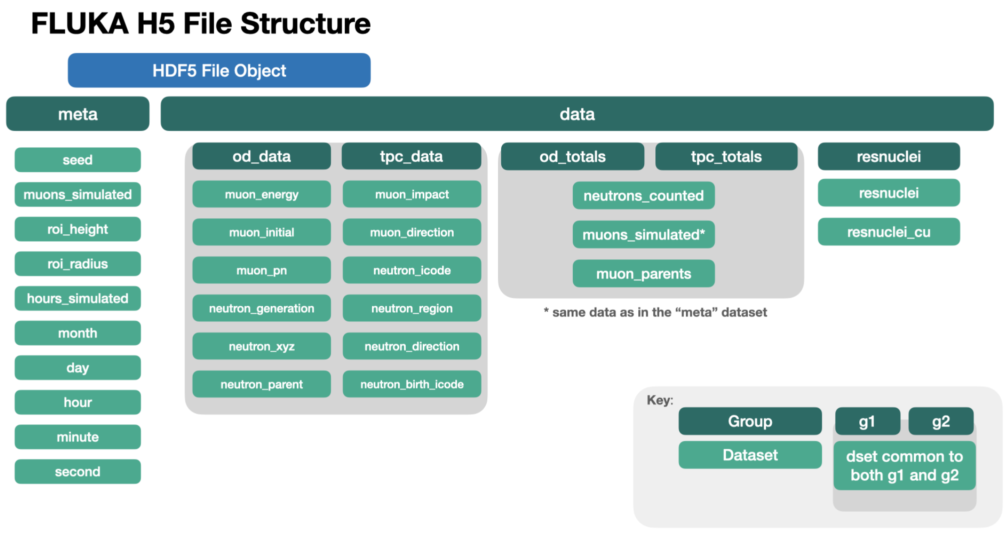


Figure 12: The structure of the output hdf5 files

References

[1] C. Ahdida et al., Frontiers in Physics **9** (2022).

[2] G. Battistoni et al., Annals of Nuclear Energy **82**, 10 (2015).

[3] Y.-F. Wang et al., Phys. Rev. D **64**, 013012 (2001).

[4] Zen Collaboration et al., (2023).

[5] FLUKA Collaboration, *FLUKA Manual*, CERN, 2022.

[6] V. Vlachoudis, Flair: A powerful but user friendly graphical interface for FLUKA, 2009.

[7] O. Scallon, *Background reduction techniques and simulations for the PICASSO and PICO dark matter search experiments.*, PhD thesis, Laurentian University, 2019-11-18.

[8] P. Giblin, The Geology and Ore Deposits of the Sudbury Structure, Technical report, 1984.

[9] V. V. Golovko et al., Sensors **23** (2023).

[10] R. Ross, nEXO Report on Muon Generation and Simulations, Technical report, Laurentian University, 2022, https://nexowiki.llnl.gov/img_auth.php/2/2d/Rross_muon_generation_report.pdf.