TRIUMF Beam Physics Note
TRI-BN-16-16
December 2016

Notes on Using G4Beamline

F.W. Jones

TRIUMF

Abstract: These notes are intended to provide hints and guidance on using G4Beamline, the Geant4-based simulation code developed by Tom Roberts. Since G4Beamline comes with its own comprehensive documentation, the emphasis here is on describing the practical aspects of putting together a simulation (including the actual layout of a beam line), how to configure and run the program, and how to obtain and interpret the results. Most importantly, cautionary tales about the peculiarities of the program are given, together with tips and guidelines for effective use.

4004 Wesbrook Mall, Vancouver, B.C. Canada V6T 2A3 Tel: 604 222-1047 Fax: 604 222-1074
www.triumf.ca

b

Contents

1 Motivation

- The Geant4 simulation toolkit has thousands of users and applications, hundreds of C++ classes, and over a million lines of code.
- In order to create a Geant4 application, the user is required to write C++ code, sometimes a lot of it, to describe the geometry, the particle beam, and the steering of the simulation.
- For some user groups, e.g. in medical, space, and muon physics, "turnkey" applications have been developed which allow simulations to be run without having to do any C++ programming at all. Perhaps less well known is that there is also a turnkey application, G4Beamline, dedicated to accelerator and beam line simulations.
- These turnkey applications are invaluable but tend to have cumbersome interfaces, such as the non-object-oriented and cryptic Geant4 "command" facility which emulates an archaic CERN library package.
- G4Beamline is remarkable because it uses a shell-like input language which is trivial to learn and easy to write and read. It has some object-oriented features such as prototypes, composites, and inheritance, and these things turn out to be very important. Actually, it is very similar to the MAD input language devised by accelerator physicists!
- In addition to its powerful input language G4Beamline is packed with features and facilities that go beyond what is available at the Geant4 toolkit level. In general, the motto "when in doubt, give the user control" has been followed and quite a bit of flexibility and programmability is built in. The best way to get an impression of this is to do some random dipping into the User's Guide (hereafter denoted by UG).
- G4Beamline is motivated above all by beam line simulations (originally for muon cooling), and does not provide facilities for, e.g., particle detectors or other complex structures requiring specialized data management. Nevertheless it is sufficiently general to allow many different uses, with or without particle interactions in matter. For example it is an efficient and accurate ray-tracer in a fully 3D setting with fields specified by expressions or field maps, and allowing overlapping fields.

2 What G4Beamline Won't Do

- Any kind of transport maps or symplectic integrators. Particle transport is only by Runge-Kutta integration through fields.
- Space charge, except for a beam travelling along a straight path (no bends allowed).
- Scoring of particle fluence, dose, activation, and other things related to shielding studies.

3 A Mini-Project

Suppose you wish to construct and run a simulation to find the gamma production and energy spectrum for a hypothetical ARIEL 50 MeV electron photo-converter? The converter is made of a block of Tantalum that is 6 mm thick. In G4Beamline this problem can be defined in 7 lines:

```
* Ariel 50MeV e- converter target TRIUMF
physics QGSP_BIC
beam gaussian particle=e- meanMomentum=50.508415 nEvents=10000
box CVTARGET width=100 height=100 length=6 material=Ta
place CVTARGET front=1 z=0
zntuple format=ascii z=7
```

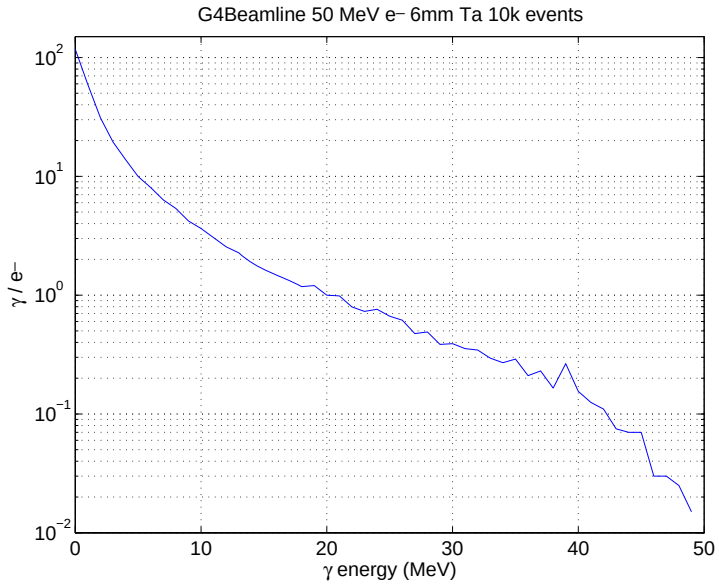
Calling this file `econv.in`, we run it as follows:

```
g4beamline econv.in |& tee econv.log
```

In around 20 seconds the ntuple file `Z7.txt` will be produced. This file records all particles that are moving forward and pass the end of the converter.

The following MATLAB procedure will read in this file, identify the gammas, and plot a histogram of their energies, normalized to show the yield per electron:

```
[x y z px py pz t PDGid EventID TrackID ParentID Weight] = ...
  textread('Z7.txt', '%n %n %n %n %n %n %n %n %n %n %n %n', ...
    'commentstyle','shell');
p = sqrt(px.*px + py.*py + pz.*pz);
pgamma = p(PDGid==22);
nbins = 50;
nevents = max(EventID);
n = histc(pgamma, 0:1:nbins);
n = n*nbins/nevents;
semilogy(bins, n, 'b');
axis([0 50 1e-2 1.5e2])
grid on
xlabel('\gamma energy (MeV)')
ylabel('\gamma / e^\_')
title('G4Beamline 50 MeV e^\_ 6mm Ta 10k events')
```



A few seconds more, and we get a nice plot. Job done!

4 Events, Tracks, and Trajectories

Before going further it is useful to explain some basic terms which otherwise may cause confusion. Some of these things have more general definitions in Geant4, but I have restricted the descriptions here to what is relevant to G4Beamline.

Event An event consists of the launching of a primary particle (either created by the `beam` command or read in from a file), tracking it through the system, and tracking all secondary particles that are created by (1) interactions or decays of primary particles and (2) interactions or decays of secondary particles. Thus, an event may consist of only a primary particle and its track, or a whole shower of particles and their tracks. Each event is assigned a number, the **Event ID**.

Track Within an event, each created particle has a track, which is also assigned a number, the **Track ID**. The term "track" is a bit misleading, because the track structure does not record the history of the particle; it is only a container that holds the Event ID of the event the track belongs to, the particle type, its current position, momentum vector, time, path length, and other book-keeping items. The coordinate values are updated at each step during the actual tracking of the particle in Geant4. Each particle track also has a **Parent ID** which is zero for primary tracks and for secondary tracks indicates the track number of the "parent" particle from whose decay or interaction the secondary was created. At the end of the event, all tracks are deleted and the track numbering starts at 1 again. Thus the primary particle of an event will always have TrackID 1.

Trajectory This data structure *does* record some history for the particle, namely the particle location at the end of each step. At the end of the event this data can be passed to the

visualization system to draw the trajectory. NOTE: at the start of the next event the previous trajectory will be deleted, unless the application requests trajectories to be stored. By default G4Beamline stores up to 100 trajectories, and this can be increased if you wish (see Section 15).

5 Known problems with G4Beamline

1. There seems to be no "halt on error" option. If you have errors (syntax or logical) in your input file, G4Beamline will go through the whole file, emitting all messages and warnings, before it stops and tells you it isn't going to do anything because you have errors. By this time, the error message has likely scrolled off the screen and maybe even out of the terminal's recall buffer. Work-around: always use a log file, even if you are running interactively for visualization:

   ```
   g4beamline myfile.in [options...] [viewer=best] |& tee myfile.log
   ```

   You can then grep the log file to find the error.

2. Sampling of magnetic fringe fields. The tracking appears to jump over the external part of the fringe field in a very non-smooth way. This may be just the way the Geant4 tracking engine is working, because it may sample the field several times before taking a step. But it is difficult to be sure, because setting `maxStep` for the magnet does not work outside the body of the magnet! Work-around: set `maxStep` globally on the command line when running the program and compare with the default. The `trace` command (Section 14.24 reveals the field values seen by a particle at each step.

3. Block input of fields. The block input format is described in detail in the `BLFieldMap` section of the UG, but the code to read it in is simply not in the program! You will get messages like:

   ```
   Bx is not implemented
   ```

   Workaround: use the pointwise input format. New problem: for 3D maps the files can get ridiculously large and reading them in takes ages. Workaround: none, unless you want to rebuild G4Beamline from source (not trivial). I have written the necessary code which can be added to G4Beamline as a modification.

4. Generally speaking it is not possible to get useful coordinates for particles inside a dipole, because the reference path cannot have circular arcs in it and must use only connected straight line segments. This is elaborated in Section 11 below.

5. The `beam` command has a parameter `maxR` for truncating the distribution at a certain radius. Unfortunately it only works for truncating at 4–5 sigma. If you set it smaller the beam generation will eventually get stuck and loop forever. This usually happens after a few thousand events have been processed.

6. During tuning, if a tune particle hits a volume with `kill=1` specified, the track will be killed, the tuning will fail, and the run will be stopped with a `Failed to converge` error. There is no way to temporarily suspend the `kill` directive during tuning. The `tune` command parameters `z0`, `z1`, and `initial` must be such that the tune particle has a clear path from its starting point at `z0` to its end point `z1`.

7. The `survey` command does not work correctly, as noted in the release notes for the most recent version (3.04).

6 Cautions

Here some are ways to get burned using G4Beamline.

- Tunable parameters are not like other parameters. Do not define them using the `param` command and do not use the `$` prefix when using these variables in element definitions.

- During tuning, stochastic processes are disabled. But if the tune particle hits an element with `kill=1` it will stop! If it goes through a material it will deterministically lose energy! These things usually lead to `failed to converge` messages.

- Spaces in the input are significant and are used as separators. Do not put them anywhere except between commands and their arguments.

- For elements with a hole (or a gap) in them such as dipoles and quadrupoles, the hole may actually be a geometry element made of vacuum. Anything you place in the hole should be placed into the parent element and not into the World, otherwise there will be an overlap and possible invalid tracking.

  Example: placing a vacuum chamber (segment of a toroid) into the gap of a dipole:

```
param LB610=1903.8051
genericbend B610 fieldWidth=1000 fieldHeight=102 fieldLength=$LB610 \
   ironColor=1,0,0 ironWidth=1000 ironHeight=1000 ironLength=$LB610 \
   fringeFactor=0.2
# Torus 2.75 inch inner diameter
param iheightB610=2.75*25.4
param hchordB610=$LB610/2
param radiusB610=$hchordB610/sin(pi/8)
param sagB610=$radiusB610-sqrt($radiusB610*$radiusB610-\
              $hchordB610*$hchordB610)
torus pipe-B610 innerRadius=$iheightB610/2 \
      outerRadius=$iheightB610/2+15 \
      majorRadius=$radiusB610 initialPhi=-22.5 finalPhi=22.5 \
      material=ss  color=.2,.2,.2,.9 kill=1
place pipe-B610 parent=B610 x=-$radiusB610+$sagB610 rotation=x+90
```

- Be careful to check for leaks in the vacuum chambers. By this I mean particles that somehow escape through unintentional gaps in the geometry. If your statistics don't seem to add up, these "leaking" particles could be the cause. Check in particular where different chamber sections are butted together, to make sure the apertures and wall thicknesses are compatible. It's not always easy to identify where the leaks are coming from since the locations recorded in `LostParticles.txt` may be far from where the leak takes place. See Section 17 for a technique to trace the origin of lost particles.

7 Input Language and Element Definitions

The input language used in G4Beamline seems to be unique to the program's author. At first glance it may seem to be rather limited and simplistic, but its main function is to act as a kind of shell to configure and launch the commands, which are where the real power lies. In this sense it is reminiscent of the Unix shell and of the Tcl language, but it does have some built-in programming constructs and a native mathematical expression evaluator which is invaluable for building and maintaining self-contained simulations.

The syntax and usage are very well documented in the UG, under Input File Description, so I will limit myself to a few remarks here. Everybody should carefully read this section of the UG, as well as the rules about geometry placements, even if they skip other things.

The input language has some object-oriented features. All commands follow the same usage pattern

```
command  name  parameter=value parameter=value ...
```

and in most cases you can think of the command as referring to a **class** and the name as referring to a newly-created **object** of that class.

Most of the keywords have default values, so in practice they can often be left out. For objects, there is a powerful **prototype-instance** functionality where some objects can be created as prototypes, for example a type of quadrupole, and the instances are the actual individual quadrupoles that are created and put into the world using the `place` command. The key aspect of this is that each instance will by default inherit all of its characteristics (parameters) from its prototype, but most of the parameters can be changed (overridden) in a given instance, thus allowing variations among the individuals, such as the strength of each quadrupole.

Another powerful feature is the creation of **composite objects**, whereby one or more objects (children) can be placed into a parent object in the parent's local coordinate system, and **groups**, which are arbitrary collections of objects that can be manipulated as one. Once such an object has been assembled, it can be put into the geometry using a single `place` command.

With these tools, constructing families of magnetic elements and elaborate geometries can be greatly simplified and the simulation can be organized into readable and maintainable units.

A final remark: accelerator physicists who have used MAD, or MADX, or DIMAD, will find G4Beamline programming very familiar, because the MAD input language, written in Fortran

and providing a durable and rock-solid foundation for those codes, is in many ways nearly indistinquishable from G4Beamline language!

8 Beam Line Layout

In G4Beamline's object-oriented world, there are not many restrictions on how you organize the input file. With a few exceptions, the order of things is not critical, as long as you don't try to use something before it is defined.

Nevertheless, I prefer to prepare my input files with a certain amount of structure, which is motivated in part by making them easy to compare with MAD and MAD input files. The structure is easily described and involves grouping the commands together into the following sections:

1. **Preliminaries**: the most basic settings and definitions, e.g. `physics`, `reference`, `beam`, `particlecolor`, `material` definitions, and the `worldMaterial` setting. Global controls such as `eventcuts` and `trackcuts` should also be put here. Individual program control parameters such as `maxStep` and `eventTimeLimit` can also be set here although I prefer to set these on the command line that runs the program.

2. **Element definitions** (including field maps): `genericbend`, `genericquad`, `solenoid`, and so on. If elements can be grouped into families, then define prototypes here and then vary the individual parameters later in the `place` commands. Beam pipes (`tubs`) and other vacuum chambers can also be defined here, near to the elements they go into or through. For tunable elements, the relevant `tune` command should be given immediately prior to the element command, so as to define the tunable parameter. If elements have virtualdetectors or other elements inside them, it is good to include the relevant `place` commands here to completely build the composite element, making it ready to place in beam line later.

3. **Beamline layout**: a sequence of `place` commands to place the individual beam line elements in centerline coordinates, with `corner` or `cornerarc` commands to change the direction of the centerline. In the place command, the `z` coordinate refers to the distance along the centerline, and I usually order the elements according to this distance, although it is not strictly required. If `zntuple`s are used they can also be placed within this `z` sequence so their relationships to the elements can be clearly seen.

4. **Further instructions**: global data-gathering or output commands such as `newparticlentuple`, `beamlossntuple`, `trace`, `profile`, `survey`, and `totalenergy` can be placed here. Finally, if `g4ui` is used to transfer control to the Geant4 command line, I put the relevant Geant4 commands at the very end of the file.

Depending on the application, I use one of two different ways of placing elements along the centerline.

8.1 Cumulative-S layout

For small beam lines, or ones where the optics are still being experimented with, it may be better to accumulate element lengths rather than deal with explicit placement locations which may have to be re-computed every time a change is made. This is comparable to using the `LINE=(element,drift,element,drift,...)` layout mechanism in MAD or DIMAD. In G4Beamline the `S` parameter is used to keep track of the distance along the reference trajectory and is incremented by element lengths and drift lengths as we go along:

```
param S=0
zntuple format=ascii z=$S referenceParticle=1
# DM01
param S=$S+400
zntuple format=ascii z=$S referenceParticle=1
# EMBTQ1
param K1=-14.9763
place EMBTQ rename=EMBTQ1 front=1 z=$S gradient=$BRHO10*$K1
param S=$S+100
zntuple format=ascii z=$S referenceParticle=1
# DM02
param S=$S+544.01
zntuple format=ascii z=$S referenceParticle=1
# EMBTQ2
param K1=23.2011
place EMBTQ rename=EMBTQ2 front=1 z=$S gradient=$BRHO10*$K1
#place EMBTQ rename=EMBTQ2 front=1 z=$S gradient=-0.001998096
param S=$S+100
```

In the above the `S` increments are given explicitly, but they could also be defined as parameters. Note that it is perfectly all right to redefine a parameter to a new value in terms of its previous one.

8.2 Direct-S layout

For larger beam lines, or those where drift lengths are no longer being adjusted, it is usually more practical to place elements at an explicit `S`-values which will be directly comparable to `AT=` values in MAD's `SEQUENCE ... ENDSEQUENCE` layout method, and can also be referenced to the engineering layout itself.

```
genericquad VQ apertureRadius=65 ironRadius=914.4/2 \
        ironColor=0,.6,0 fringeFactor=0.1
param LVQ1=411.10
param LVQ2=407.00
param LVQ3=397.20
   ...
param S=483
```

```
zntuple format=ascii z=$S referenceParticle=1
place VQ rename=VQ1 front=1 z=$S fieldLength=$LVQ1 \
       ironLength=$LVQ1 gradient=$qsfv*(-4.79300)
param S=$S+$LVQ1
zntuple format=ascii z=$S referenceParticle=1

param S=1284
zntuple format=ascii z=$S referenceParticle=1
place VQ rename=VQ2 front=1 z=$S fieldLength=$LVQ2 \
      ironLength=$LVQ2 gradient=$qsfv*(7.38300)
param S=$S+$LVQ2
zntuple format=ascii z=$S referenceParticle=1

param S=3498
zntuple format=ascii z=$S referenceParticle=1
place VQ rename=VQ3 front=1 z=$S fieldLength=$LVQ3 \
      ironLength=$LVQ3 gradient=$qsfv*(10.50500)
param S=$S+$LVQ3
zntuple format=ascii z=$S referenceParticle=1
   ...
```

Here, the `S`-values were taken from REVMOC and refer to the entry points rather than the center of the quads, hence the `front=1` usage. The quad lengths are all defined as parameters and thus by incrementing `S` the same command can be used for each quad to place a zntuple at its exit.

9 Parameters

9.1 Run-level Parameters

G4Beamline has a number of global parameters that apply to the run as a whole. Most of them are itemized in the table in Section 4.3.1 of the UG but for the the complete list refer to Section 4.8 which lists Commands by Type.

It is not necessary to set any of these global parameters unless you want to change them from their default values. This can be done on the command line that runs the program:

```
g4beamline myfile.in parameter1=value1 parameter2=value2 ...
```

or in the input file using `param` commands.

Some notes about parameters:

- `viewer` Setting this to `best` or to the name of a specific visualization driver will enable the run to go into interactive visualization mode after its initialization phase or, optionally, after some events have been completed. At this stage, a viewer window will pop up, or for

some drivers a graphics file will be produced for external viewing. By default `best` refers to the Open Inventor (OIX) viewer, which used to be the best interactive viewer. Now, there is a better one: the TRIUMF-developed Open Inventor Extended viewer (OIXE) which has many diagnostic and convenience features motivated by the needs of beam line simulators.

- `steppingVerbose` A similar feature to that in Geant4 itself, which causes information to be printed at each step a particle takes. Indispensable for debugging and for understanding the behavior of Geant4, and completely customizable.

- `eventTimeLimit` If your run seems to "go to sleep" it could be due to a track that never completes, because of a geometry problem or because the particle is looping in a field and never leaves the world volume. Each event (primary particle launched) resets the clock and if the time is exceeded the event is killed and the next one processed. The default time limit is 30 seconds which can be reduced by the impatient.

- `maxStep` Extremely valuable for debugging, especially in cases where tracking in fields is involved. Every step will be limited to this length everywhere in the geometry. Geant4 sometimes takes rather large steps, because it can do so while still preserving accuracy, and this parameter can be used to smooth out line plots of trajectories that otherwise may look non-physical, as well as just to check on what is going on at a finer level of detail.

- Tracking accuracy parameters These control the integration through electric and magnetic fields, which is a sophisticated and highly optimised part of the Geant4 code. Unless you are already an expert, messing with these is very unlikely to provide any material improvement in either performance or accuracy.

- `zTolerance` I haven't used this, but it could be important if you are working with very small scale or large scale problems. It controls the longitudinal "granularity" of the geometry for zNtuples and other processing where the step needs to be limited in order to collect data or check a condition. This parameter is needed in order to preserve consistency and avoid contention between different agents which want to control the step length, such as approaching a boundary, sampling particle coordinates, checking for a condition, or entering a field region.

9.2 User-defined Parameters

The user is free to create and define any number of parameters and use them in any way through the `$parametername` substitution mechanism. Parameters must be created (named) in the input file, but can be set or overridden on the command line provided their `param` definition includes the `-unset` tag. This can be very powerful indeed.

If you are in doubt about the best values for some parameters in your simulation, setting them on the command line enables a series of test runs to be quickly done, possibly from a simple shell script.

For studies of errors and misalignments, the δ-values for fields and element placements can be generated or sampled externally and then plugged into the command line for each run.

Collective beam properties such as emittances and beam sizes can be tuned by using scripts to drive G4Beamline together with an external mimimizer program. In fact, G4BLMINUIT is just such a facility and comes with the G4Beamline distribution.

10 Conventions

In Geant4 one can use virtually any system of units at all, either by choosing from among the many pre-defined ones or by defining your own.

In G4Beamline this is not the case, and a fixed system of units must be adhered to:

- lengths in millimeters
- energy in MeV
- momentum in MeV/c
- angles in degrees
- magnetic fields in Tesla
- electric fields in megaVolt/meter

For accelerators and particle beams I have found these units to be quite practical and a reasonable compromise. Of course, the G4Beamline input language allows you to define your own variables and use whatever kinds of scalings and transformations you want.

From the point of view of geometry and tracking, Geant4 is a true 3D system that does not make any special distinctions between the different coordinate axes or directions. Beams can go anywhere in the world, and objects can have arbitrary positions and rotations. This is also true in G4Beamline, with the exception that there is a user-defined *reference path* called the **centerline** and a centerline coordinate system that can be used to orient the primary beam and to place beam line elements in a convenient way. It is similar to but not the same as the moving "curvilinear" coordinate system used in accelerator theory.

Actually G4Beamline has three coordinate systems available to the user:

1. Global coordinates: a fixed Cartesian coordinate system. By default the beam will start at `z=0` and move along the `z` axis, but these things can be changed.
2. Centerline coordinates: the `z` coordinate is the total distance along the centerline, and the `x` and `y` coordinates are perpendicular distances from the centerline. It is a right-handed system with `y` pointing "up" so if you are looking in the positive direction along the `z` axis then the `x` axis points to the left.
3. Reference coordinates: these are relative to the path of the reference particle defined in the `reference` command and tracked at the beginning of the run. They are like centerline coordinates with the reference particle path substituted for the centerline.

For many problems the centerline coordinates will be all that you need, and the other systems can be avoided. A limitation of G4Beamline is that there can be only one centerline, so if you want a primary beam line to have a branch into a secondary line, you have to choose which branch gets the centerline, and use global coordinates for the other one (probably not a pleasant scenario).

11 Bends and Their Tuning

In the curvilinear coordinate system that is ubiquitous in accelerator and beam line theory and codes, the reference axis through a bend is a circular arc that spans the body of the magnet and is tangent to the straight parts of the path at its entrance and exit.

This is not available in G4Beamline and the reference arc has to be replaced by one or more straight line segments. This need gives rise to the `corner` command which at a given Z location (distance along the centerline) defines a vertex where the next line segment proceeds in a new direction, defined by rotation angles around the X, Y and Z axes. There is a variant `cornerarc` which breaks the rotation into three corners so as to preserve path length.

This state of affairs is unfortunate, because:

1. The centerline coordinates of particles inside dipoles are not very meaningful or useful: instead of a particle's deviation from the ideal orbit of a reference-momentum particle, which is what is wanted, the transverse coordinates grow and shrink in a way that appears to be "nonsense" if you attempt to use these coordinates in plots or diagnostics.

2. The use of abrupt corners leads to an ambiguity in determining the centerline coordinates of particles: a particle may lie in the "perpendicular zone" of more than one line segment, so which segment should be used to compute the X and Y centerline coordinates (perpendicular distances) of the particle? In practice, the convention could be simply taken to use the shortest distance (if the distances are the same then we are okay!) from the centerline, but it is not clear if this is done.

3. A parameter `radiusCut` has been introduced to attempt to fix this ambiguity. See Figure 1 and the **Centerline Coordinates** section of the UG. Note that the radiusCut line passes through the ambiguous regions. Inside the line, the ambiguities remain for all particles in the blue, green and yellow regions. Particles that are outside the radius cut will be killed! This hardly seems like a solution, and it is an effort to resolve ambiguities in coordinates that are not very useful anyways. If you leave out the radiusCut (as I usually do) you will get a series of warning messages, one for each corner or cornerarc, but the program will continue.

For the conventional reference path using line segments and arcs tangent to them, there is no ambiguity as the shortest (perpendicular) distance of a particle to this path is always well defined.

But how to live with what we have? For each bend one has to give one or more corner commands (or a cornerarc command) placed at appropriate Z values so that the centerline will

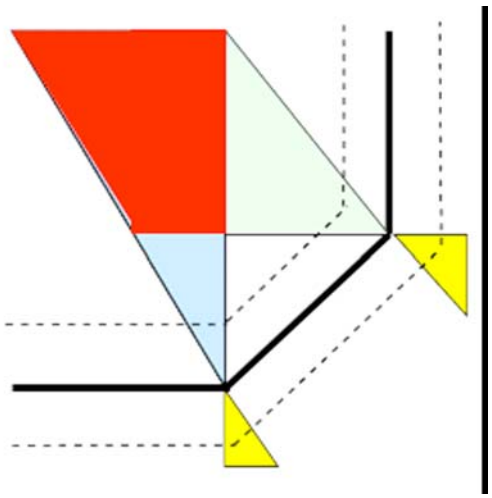


Figure 1: Centerline with radiusCut zone (dashed) and ambiguity regions (coloured).

eventually coincide with the curvilinear reference path followed by an ideal particle through an ideal dipole.

In the UG it is stated that "each bending magnet is normally immediately followed by a *matching* corner or cornerarc command" (my italics). This puzzled me at first: does it mean that if I have a 35-degree bend I should place the bend, and then place the corner immediately after that? This makes no sense, because then the centerline (reference path) will go straight through the magnet and then suddenly turn by 35 degrees at the end of the magnet!

Well, what about placing the matching corner at the crossover point defined by the bend? This makes sense from an engineering point of view, since it is exactly how the beam line will be surveyed. However the path length along the centerline will then be nonsense, not comparable to the path length of a particle actually traversing the bend, and correspondence with any other beam optics or tracking software will be gone.

Then there is the `cornerarc` command. It is a set of three line segments that are rotated through the desired bend angle, and have a total path length equal to the reference arc through the dipole. Unfortunately, the transverse coordinates will still be nonsensical, and will possibly even change sign because the centerline criss-crosses over the particle's path.

A simpler approach is to place corners at the entry and exit point of the dipole, each with a rotation of half the bend angle. The dipole is placed at the center of the line segment connecting the corners, with a suitable transverse displacement to compensate for the sagitta. How long should the line segment be? One possibility is to make it the same length as the chord of the arc that represents the conventional reference path through the dipole, but this will result in incorrect distances for placing downstream objects at the S-values given by distance along the centerline. The other possibility is just to make the segment the same length as the `fieldLength` parameter of the dipole. This results in the correct distance of the downstream elements from the dipole, but distances seen by the tracked particles will be slightly larger, because they follow the arc rather than the line segment (the chord of the arc).

Your mileage may vary, but I have had good results for the latter choice, in the following senses:

1. With slight tuning of the dipole field from its nominal value, the reference particle will bend through the correct angle *and* have the correct displacement at the exit from the dipole, to quite a good precision (in general allowing centering within 1 mm down the line just by tuning the dipoles and without any additional correction).

2. The small error in the particle path length compared to the centerline path length seems to have little impact on the focusing and beam sizes. For models of BL2A and BL1A the quadrupole strength values calculated directly from DIMAD, REVMOC or TRANSOPTR values can just be plugged into G4Beamline and the RMS beam sizes will agree within 1-2 mm. For BL4N, which has five bends and is a bit more touchy, I had to make a 5% adjustment to one quadrupole to get 1-2 mm agreement.

Finally, it should be noted that the layout of the centerline only influences the calculation of centerline coordinates. It in no way affects the actual tracking of particles. Each particle carries its own set of (global) coordinates, including path length and time coordinates that depend only on the materials and fields seen by the particle.

12 How Passive is Passive?

Each track in Geant4 consists of a series of steps, taken sequentially. When a particle is created, it has a certain position (x, y, z) and momentum $\mathbf{p} = (p_x, p_y, p_z)$. In practice, the momentum vector is not stored directly in the track but is derived from the kinetic energy (scalar) and a normalized "momentum direction" vector $(p_x/p, p_y/p, p_z/p)$.

At the end of each step, the particle's position and momentum are updated to new values, together with its time coordinate (elapsed time since particle creation) and its path length (total distance travelled since creation). During the step, the particle may gain or lose energy, due to the presence of fields or due to interactions in matter such as ionization energy loss, bremsstrahlung, and so on. These changes are reflected in the coordinate values at the end of the step, and thus influence the subsequent track of the particle.

In Geant4 step sizes are highly variable; in fact, for a stable particle in vacuum, with no fields present, a huge step may be taken, all the way to the next material volume, or to the edge of the world volume. For unstable particles, the maximum step size is governed (in part) by when the particle decays, based on random sampling of its lifetime. Then there is the geometry – steps must not cross boundaries between volumes: when approaching a boundary, the step is automatically cut back to end at the boundary. In fields and/or materials, step sizes are chosen according to sophisticated criteria to guarantee accuracy of integration and the integrity and consistency of the models applied for each physical process. Models for "continuous" processes such as multiple scattering and energy loss may have to cope with step sizes ranging over orders of magnitude. Great care has been taken to reduce the step-size sensitivity of these models insofar as possible, but it is a non-trivial task.

The big question is, "if I change the step size but change nothing else, will I get different results for the track?" The answer is, "in most cases yes!" This is a fact of life for simulations driven

by random number generators. Even when tracking in a field-free vacuum, different step sizes entail differences in round-off error that can eventually lead to visible differences in coordinates.

The consequence of this is that:

> The apparently "passive" elements such as zntuples, timentuples, and virtualdetectors, may have a visible and even profound effect on a given track, when they are added to a simulation. This is because these elements act as step-limiters in the same way as a physical boundary does.

Thus, if you want to use, for example, a zntuple to debug a certain track behaviour, keep in mind that after placing the zntuple you will be looking at a different track!

13 Independence and Reproducibility of Events

Geant4 uses state-of-the-art random number generators that have been developed over the years by and for high-energy physics researchers who are very fussy about such things. Nevertheless, if the random generator is seeded only once at the beginning of a simulation then events cannot be independent: the random seed at the beginning of an event will be the result of the previous events' use of the generator.

An important aspect of G4Beamline is that it *guarantees the independence of events*. At the beginning of each event the random generator is seeded with a new seed that is in fact the Event ID of the event, which every event must have. Thus each event carries its own random seed with it. Regardless of what other events are run, and in which order, an event will always have identical results, as long as the geometry and physics of the run have not changed. As noted above, "geometry" here must include any step-limiting elements – adding or changing them will break the reproducibility of the track.

The independence of events is very useful for debugging and for doing detailed analysis and visualization. In a large run, particular events can be selected and used in a new run with all other events left out, allowing their tracks to be easily visualized, or examined using the `trace` command. See Section 17 below.

14 Notes About Specific Commands

14.1 `beam`

Note the defective `maxR` parameter (see Section 5).

The number of beam particles generated does not have to be specified in the input file. If you put instead:

```
beam ... nEvents=$nEvents ...
```

then you can specify `nEvents` on the command line when you run the program – much more flexible.

14.2 `detector`

This command is very similar to `virtualdetector`, with the difference that is has a material of its own rather than taking on the material of the parent it is placed into.

Normally I would tend to use `virtualdetector` but the good news is that as of Version 3.04 there is a `solid=` keyword that allows you to make *any preexisting volume into a detector*, something I have been wishing for for years.

14.3 `reference`

This defines the starting coordinates of a reference particle which is independent of the `beam` particles and will be separately tracked before the beam particles are. Its track is used to define `reference` coordinates. It has eventID -1 and can be given its own colour, which is useful for visualization and debugging

14.4 `tune`

Even for an already-optimized beam line, this command may be necessary in order to adjust the dipoles (and/or steerers if you have them) to obtain reasonable beam centering.

The minimization algorithm is both simple and fragile. It uses Newton's method to try to converge on the zero of the objective function and it seems to easily go off the rails. There is some output printed during tuning but often it is insufficient to diagnose problems. In the worst case the process will quit immediately without printing anything at all, except a "failed to converge" message.

A few things to note:

- Only one variable and one objective are allowed. It follows that in general it is not possible to tune a track to have both zero position and zero angle at the same location.
- The tuning variable is special: don't define it with a `param` command and don't use the `$` prefix with it.
- Each tune command launches a series of tune particles, one for each iteration, starting at Z0 and ending at Z1 where the objective function is evaluated. If the tune particle fails to reach Z1 for any reason you will get a `failed to converge` error! In particular, if it enters an element with `kill=1` it will be killed!
- Tuning disables the stochastic processes, but don't over-interpret this. It only disables energy loss *fluctuations* and if the tune particle traverses a material it will still (deterministically) lose energy.

- The tuning algorithm will break very easily if you try to use it blindly. Choose a sensible starting value and a modest tolerance (e.g. 0.1mm if you are centering the beam). Experiment with the step size; setting it too large or too small may result in the by-now familiar `failed to converge`.

In spite of the weakness of the `tune` command, it is still possible to do multi-variable multi-objective optimization with G4Beamline using the external **gminuit** program available free from the G4Beamline web site[2]. See Section 7.25 of the UG.

14.5 `beamlossntuple`

Very useful. Recommended for every application because it tells you where all particles end up. If they don't stop or decay, it tells you where they exit the World Volume that encloses your geometry.

14.6 `demo`

This is similar to the `echo` command in the Unix shell. If you are not sure about the evaluation of an expression this can be used to check it.

14.7 `extrusion`

This versatile command can be used to make objects of polygonal cross section, centered on the Z axis with linear scaling in X and Y. For example, a circle or ellipse can be approximated by a polygon and then stretched out to form a section of a cone or elliptical taper, as in the following implementation of a collimator with entrance and exit taper in a BL2A target module:

```
# Collimator length in mm
param Clength=6*25.4
# Radius of central bore
param RCbore=0.5*0.75*25.4
# These lengths are increased a bit to avoid gaps
# Length of central bore
param LCbore=2.39*25.4+1
# Length of tapers
param LCtaper1=2.38*25.4+1
param LCtaper2=1.23*25.4+1
# Max radius of tapers
param RCtaper=0.5*(1.0*25.4)
box COLL height=$Clength width=$Clength length=$Clength \
         material=Cu color=0,1,0
tubs CBORE innerRadius=0 outerRadius=$RCbore \
           initialPhi=0 finalPhi=360 \
           length=$LCbore material=Vacuum color=0,0,1
```

```
extrusion TAPER1 length=$LCtaper1 \
   vertices=1.000000,-0.000000;0.866025,-0.500000;0.500000,-0.866025;\
   0.000000,-1.000000;-0.500000,-0.866025;-0.866025,-0.500000;\
   -1.000000,-0.000000;-0.866025,0.500000;-0.500000,0.866025;\
   -0.000000,1.000000;0.500000,0.866025;0.866025,0.500000 \
   scale1=$RCtaper scale2=$RCbore material=Vacuum color=0,0,1
extrusion TAPER2 length=$LCtaper2 \
   vertices=1.000000,-0.000000;0.866025,-0.500000;0.500000,-0.866025;\
   0.000000,-1.000000;-0.500000,-0.866025;-0.866025,-0.500000;\
   -1.000000,-0.000000;-0.866025,0.500000;-0.500000,0.866025;\
   -0.000000,1.000000;0.500000,0.866025;0.866025,0.500000 \
   scale1=$RCbore scale2=$RCtaper material=Vacuum color=0,0,1
```

14.8 `fieldexpr`

The key fact here is that the expression is not evaluated at run time (which could be very inefficient). Rather, a field map is generated which is then scaled by the time-dependent expression if there is one. In turn, the time-dependent expression is modeled between `tmin` and `tmax` by fitting a spline which allows for fast run-time evaluation. The `tolerance` parameter governs how accurate these discretizations are.

14.9 `g4ui`

Gives access to the Geant4 command interface at a specified stage of the run. Can be used, for example, to set up a particular visualization environment before the viewer is opened.

14.10 `group`

A group acts a generic “parent volume” into which geometric objects may be placed (relative to a starting point or a center point) and then manipulated as a single entity, including limiting the step size in all group objects. The group volume is a box or cylinder and can have a material, or be “invisible” (vacuum or same material as world volume).

14.11 `material`

This command is not needed for most materials because they are pre-defined (all are listed in the UG). What may be overlooked is that you can define a material that will kill specific particles, or more generally will kill or pass tracks based on a `require` expression.

14.12 `newparticlentuple`

Logs all particles, primary and secondary, as they are created. Optionally, certain tracks can be killed immediately after being created and recorded! A `require` expression can be used to control what goes into this file. As with its counterpart `beamlossntuple` this is an essential command for debugging and analysis.

14.13 `particlefilter`

Another essential command to know. Note that it occupies a rectangular or cylindrical volume, and can be placed into a parent volume (by default it will have the same material as the parent).

14.14 `particlesource`

An interface to the General Particle Source module of Geant4, which is a command-driven facility to generate primary particles. It is much more general than the built-in `beam` generation in G4Beamline.

14.15 `physics`

This command puts together a complete package of physics processes for your simulation. The instantiation and management of these processes is one of the most complex parts of Geant4, but here it is all done for you, together with selective enabling and/or disabling of particular processes if you wish.

One very useful parameter that can be added is `doStocastics=1`, which simply turns off all particle interactions in matter **except** the deterministic ones. Particles will still lose energy in a material, but the energy loss fluctuations will not be there.

14.16 `place`

This command has context-dependent effects, so check the UG entry carefully if you are experiencing unanticipated behavior. The `front=1` option is very convenient for specifying where a beamline element starts rather than where its center is, but note that as of Version 2.16 if this is used then any `rotation` of the element will be done at the front `z` of the element rather than its center.

14.17 `profile`

I have never been able to get sensible results from this command, in spite of trying to carefully formulate the `require` setting to use only the relevant tracks (i.e. those of the primary beam, inside the vacuum chamber, and going forward).

As of Version 3.04 it has been acknowledged in the release notes that “the profile command has numerous problems in the way it computes values; the only workaround is to use a zntuple and compute the values you want”.

14.18 `reweightprocess`

As with any biasing, this should be used with caution and checked carefully against non-biased runs to make sure the final results do not contain further hidden biases or artifacts.

This acts only at the process level (e.g. `ProtonInelastic`). The individual reaction channels cannot be independently biased.

14.19 `showmaterial`

Used for visualization, to assign colours to materials (as opposed to assigning colours to geometry elements), and to optionally make materials invisible.

14.20 `spacecharge`

This is a remarkable achievement. Geant4 tracks particles in distance steps, one particle at a time until its track is finished. Tracking with space charge requires tracking in time steps, with all particles being advanced in concert at each step. Implementing space charge with Geant4 essentially means turning the tracking algorithm inside out! G4Beamline does this by intervening in the transportation and track-stacking processes in Geant4. In some sense this is also a tribute to the design and flexibility of Geant4.

Other than emittance growth in a drift, I have not seen any extensive validation or examples of the use of this package. One thing to be aware of is that it uses a boosted-frame computation and so does not allow a curved trajectory for the beam!

14.21 `survey`

This command prints out coordinates at the start and end of each element placed into the world, sorted by Z. Useful for plotting the machine layout as a graph.

14.22 `timentuple`

A remarkably useful and powerful command, that records the space coordinates of all particles at a specific time, or series of times. Typically the S-based beam optics codes have to do this by forward- or back-tracking each particle to get a common time value. Here it is done easily and precisely using step-limiting, ensuring the correct treatment of any fields that are present.

14.23 `totalenergy`

This tallies the energy deposited at the volume level, so if you wish to get a fine-grained distribution it is best to make a `virtualdetector` of thin layers and place it into the parent volume of interest. For 3D scoring a collection of virtualdetectors is needed. The data is printed by volume name so some external processing of the output will be needed to get it into a form useful for thermal analysis. Room for improvement here.

14.24 `trace`

Unless trajectories have been created for visualization, generally the along-the-track coordinates of particles are not kept around. This command allows you to record this tracking data in files, one for each particle, up to a certain number of particles specified by `ntrace`. The reference and tune particles are also traced. If you don't want to trace secondary tracks, set `primaryOnly=1`.

This data can be essential for debugging, in particular when fields are present, as the field values "seen" by the traced particle at each step are included in the data. Remember that the field values are not given along the axis, but at the actual particle locations!

14.25 `trackcuts`

Another invaluable and sometimes overlooked command. Provides control at the track and step level, regardless of other considerations. Can be used to kill specific particles, do energy and time cuts, and get rid of secondaries completely by `killSecondaries=1`.

14.26 `virtualdetector`

The basic device for gathering data by volume. By default it assumes the material of the parent volume it is placed into.

Note that the `place` command has a `copies=` parameter to stack multiple copies of a virtualdetector sequentially into a parent volume.

In my attempts to use the `kill=1` option, it didn't seem to work and the particles just sailed on through, but this may be related to visualization and is said to be fixed in Version 3.4.

14.27 `zntuple`

Similarly to a virtualdetector, this samples all particle coordinates as they pass the Z location of the zntuple. Each zntuple generates a separate ntuple file, with automatic naming according to the Z value. In laying out a beam line I often place these at the entrance and exit of each beam line element and/or other locations of interest. They can be easily commented out when not needed.

Note that these are snapshots in space rather than time, and most generally each particle in the ntuple will have a different time coordinate. By contrast, snapshots in time can be accomplished with `timentuple`.

15 Visualization

15.1 Open Inventor

Geant4 has many visualization options, but for the day-to-day work of developing and debugging a simulation and its geometry, the Open Inventor Extended (OIXE) viewer is recommended. This viewer was developed at TRIUMF as part of our contribution to the Geant4 collaboration. The initial programming was done by co-op students Rastislav Ondrasek and Pierre-Luc Gagnon, and I am continuing to develop and maintain it.

The OIXE viewer extends the basic Open Inventor viewer developed in the early days of Geant4, adding many new features and functionalities. These have been motivated by my own experiences creating simulations with G4Beamline:

- Save and restore the viewpoint and camera settings using a bookmarks file (surprisingly absent from the other viewers)
- User-definable reference path for navigating the camera along the beam line, specified as a series of 3D points or by selecting a particle trajectory
- Clickable list of elements along the reference path ordered by distance. Clicking on an element moves the camera to that element.
- Interactive navigation and camera rotations on the reference path, using the keyboard (supplementary to the free navigation using the mouse and thumbwheel controls).
- Mouse-over functions for trajectory and volume readout.
- Animated fly-through of the geometry along the reference path (or fly-over or fly-under).
- Menu functions for PostScript and PDF output of the view (with PDF support for transparent objects).

15.2 Transparency

Open Inventor has the ability to set a transparency factor (alpha channel) as part of the colour property of any object in the scene. The syntax to use in G4Beamline commands is:

```
color=r,g,b,a
```

where `r,g,b` are the usual colour values and `a` is the transparency (alpha) value ranging from zero to 1, with 1 being opaque (the default). Note that the PostScript output from the viewer does not preserve transparency, but the PDF output does.

15.3 The `viewer.def` file

Some preliminary set-up is often needed before opening a particular viewer for visualization. To accomplish this and allow for user modifications, the file `viewer.def` is provided in the G4Beamline installation in subdirectory `share/g4beamline/`. For each viewer this gives you control over initial camera settings, how many trajectories to store, how big to make the viewer window, and so on.

16 Field Maps

G4Beamline's support for user-specified electric and magnetic field maps has some good aspects: it is relatively easy to create ascii files describing the maps, and there is a first-order treatment of overlapping fields, such as the fringe fields of magnets that are close to each other. This can be an indispensable feature for some beam lines. On the bad side, only linear interpolation of field values is provided, so fields are not continuous across mesh points and a fine mesh may be needed to obtain sufficient accuracy.

The BLFieldMap format, for inputting the field map structure and values, is described in Section 9.4 of the UG. For 2D maps the easiest variant is to use the pointwise input. For 3D maps the pointwise format can lead to very large files and a very slow program startup, but unfortunately the more compact **block format** is **not implemented** in the G4Beamline code, as you will find out if you try to use it.

I have written the necessary code for the block input and have used it to modify older versions of G4Beamline. At the time of writing, I don't have the current version modified in this way but will be pursuing it.

16.1 Opera Field Maps

Opera[5] has the an option to evaluate a field solution on a cartesian grid and write the values into a file. The following little program, modified for your grid dimensions, can be used to convert the Opera file, connected to logical unit 1, into a BLFieldMap file on logical unit 2:

```
      IMPLICIT NONE

      INTEGER NX,NY,NZ,I1,I2
      REAL X,Y,Z
      REAL*8 BX,BY,BZ

      CHARACTER*80 CBUFF
      CHARACTER*60 R8BUFF
      INTEGER I,LINE
      INTEGER LSIG

 1000 FORMAT(A)
```

```
 2000 FORMAT(3E20.12)

      LINE=1
      READ(1,*,ERR=98,END=99)NX,NZ,NY,I2
      WRITE(6,*)'NX,NZ,NY = ',NX,NZ,NY

      DO I=1,7
         READ(1,1000,ERR=98,END=99)CBUFF
         WRITE(6,*)CBUFF(1:LSIG(CBUFF))
      ENDDO

      WRITE(2,1000)'#BLFieldMap'
      WRITE(2,1000)'param gradient=0 normB=1'
      WRITE(2,1000)'grid X0=-100 Y0=-15 Z0=-300 nX=40 nY=7 nZ=120'//
     &     ' dX=5 dY=5 dZ=5'
      WRITE(2,1000)'data'

      LINE=0

   10 READ(1,*,ERR=98,END=99)X,Z,Y,BX,BZ,BY
      LINE=LINE+1
      WRITE(R8BUFF,2000)BX,BY,BZ
      WRITE(2,*)X,Y,Z,R8BUFF
      IF(Y.NE.0.)THEN
         WRITE(R8BUFF,2000)BX,BY,-BZ
         WRITE(2,*)X,-Y,Z,R8BUFF
      ENDIF
      GO TO 10

   98 WRITE(6,*)'Error reading file at line',LINE
      STOP

   99 WRITE(6,*)'End of file'
      WRITE(6,*)LINE,' data lines processed'
      STOP
      END


      FUNCTION LSIG(STRING)
C=====================================================================C
C  Returns significant length of a character string.
C  All characters are significant except blanks and nulls.
C=====================================================================C
      IMPLICIT NONE
      INTEGER LSIG
      CHARACTER*(*) STRING
      CHARACTER*1 NULL
      INTEGER I
      DO I=LEN(STRING),1,-1
```

```
        LSIG=I
        IF(STRING(I:I).NE.' '.AND.STRING(I:I).NE.NULL)RETURN
      ENDDO
      LSIG=0
      RETURN
      END
```

NOTE: the above little converter is an example of a kind of "on the fly" Fortran programming which I use frequently in conjunction with G4Beamline. Since compiling and linking these small programs is virtually instantaneous, it is simpler to just edit the code to do what you want, rather than designing some additional parameter interface. I/O, numerical, and string programming are incredibly easy in this obsolete language called Fortran. For more sophisticated and automated procedures, use your favourite scripting language.

17 Diagnosis and Back-Tracing with Eventselector

Every event in Geant4 has a unique integer called the EventID associated with it. This is merely a tag and there is no particular order or sequence enforced on events. In G4Beamline events have an additional very nice property: each event is an independent entity because the event ID is used to initialize the random seed at the beginning of the event. This makes events "portable" and any event that is re-run in the same simulation set-up will produce the same track(s) as it did in the original run.

Let's say you have run a million protons, and looking in LostParticles.txt you see that some of their tracks end up in places you didn't expect. Is there something wrong with the geometry or fields? It is impossible to visualize or trace even 0.1% of these protons at a time, so how to find out what is going on?

Here is `eventselector.f`, another Fortran fragment which allows you to pick out the suspicious events and re-run them. Again, the file can be simply edited to impose the desired selection criteria.

```
      IMPLICIT NONE

      REAL X,Y,Z,PX,PY,PZ,T
      INTEGER PDGID,EVNUM,TRKID,PARENT,WEIGHT
      CHARACTER*160 BUF
      INTEGER EVNUMPREV,EVNIN,NSEL
      INTEGER LSIG

 1000 FORMAT(A)

      OPEN(UNIT=82,STATUS='UNKNOWN')
      WRITE(82,1000)
     &  '#BLTrackfile created by EVENTSELECTOR'
      WRITE(82,1000)'#x y z Px Py Pz t PDGid EvNum TrkId Parent weight'
      WRITE(82,1000)'#mm mm mm MeV/c MeV/c MeV/c ns - - - - -'
```

```
C Skip header lines
      READ(1,*)
      READ(1,*)
      READ(1,*)
      READ(81,*)
      READ(81,*)
      READ(81,*)
      EVNIN=0
      NSEL=0
      EVNUMPREV=0

   10 READ(1,*,END=99)X,Y,Z,PX,PY,PZ,T,PDGID,EVNUM,TRKID,PARENT,WEIGHT
      IF(EVNUM.EQ.EVNUMPREV)GO TO 10

C Discard EVENT based on SELECTION CRITERIA
      IF(Z.LT.30000..OR.Z.GT.42000.)GO TO 10

      EVNUMPREV=EVNUM
      WRITE(6,*)'Selecting event ',EVNUM
      DO WHILE(EVNIN.NE.EVNUM)
         READ(81,1000)BUF
         READ(BUF,*)X,Y,Z,PX,PY,PZ,T,PDGID,EVNIN,TRKID,PARENT,WEIGHT
      ENDDO
      WRITE(82,1000)BUF(1:LSIG(BUF))
      NSEL=NSEL+1
      GO TO 10

   99 WRITE(6,*)'End of selection file'
      WRITE(6,*)NSEL,' events were selected'

      END
```

The file LostParticles.txt is connected to Unit 1. The original beam input coordinates file (1 million protons) is connected to Unit 81. Events are read from LostParticles.txt. If an event is selected, its original record in Unit 81 is copied to Unit 82. The file on Unit 82 becomes a new input file for G4Beamline containing only the events of interest. In a new run, these events can be visualized (if there are not too many of them) and/or traced with the `trace` command.

18 Post-processing with Matlab

G4Beamline has extensive built-in support for Root[4] data files and histograms, and an excellent program Historoot for post-processing and graphics. If you are doing complex, large-scale simulations and analyses, this might be the way to go.

For my own projects, I prefer human-readable data files and a free-wheeling scripting language, and have used Matlab scripts, again usually created or modified on-the-fly, for most of my

analysis and plotting of simulation results. G4Beamline excels in providing many ways to get data out of the program, and most of it is in the self-describing **ntuple** format. The most straightforward way to read ntuple files into Matlab is with the `textread` command. The following suffices to read files created by the `zntuple` and `beamlossntuple` commands:

```
[x y z Px Py Pz t PDGid EventID TrackID ParentID Weight] = ...
    textread(filename,...
             '%n %n %n %n %n %n %n %n %n %n %n',...
             'commentstyle','shell');
```

and the following will read *trace* ntuple files:

```
[x y z Px Py Pz t PDGid EventID TrackID ParentID Weight ...
 Bx By Bz Ex Ey Ez] = textread(filename,...
 '%n %n %n %n %n %n %n %n %n %n %n %n %n %n %n %n %n',...
              'commentstyle','shell');
```

The `PDGid` data item identifies the species of particle, according to the unique integer identifier assigned to each particle by the Particle Data Group[7]. The most common ones are listed in Appendix 6 of the UG.

See the other output commands in the UG for additional ntuple formats and extensions. With the above input methods and a reasonable machine to run on, ntuples of a million particles or so should not pose any difficulty, but files with $\sim$10 million may run into trouble, either by Matlab running out of memory or becoming very slow. In this case data must be read in and statistics compiled one record at a time, possibly using a compiled language instead of Matlab.

19 Competitors

There are now quite a few accelerator and beam line tracking codes that incorporate particle interactions in matter, including TRIUMF's REVMOC code which has been a mainstay of our primary and secondary beam line design and configuration. I won't attempt to survey these codes but my general impression is that they tend to be limited, in terms of their capability in 3D geometry and tracking, and/or in terms of their physics coverage: particles, materials, and physical interactions over a wide range of energies. In these respects, G4Beamline is really in a different league, although it may be less well adapted to the needs of working accelerator physicists than some of the older codes.

The more general particle simulation code FLUKA has been used in many accelerator applications and is an industry standard for radiation dose and activation studies. The code and its front-end FLAIR are not very congenial to laying out a beam line, but some additional tools[8] have been developed to ease this process. Unfortunately, FLUKA is subject to very restrictive licensing where individual users must register and agree to a lengthy set of terms and conditions in order to use the software.

A most interesting alternative to G4Beamline is BDSIM[9], also based on Geant4 and developed at Royal Holloway University of London. When I first started with G4Beamline to construct a

model of Beam Line 2A, I also became aware of BDSIM and tried to evaluate the two codes side-by-side for this application. BDSIM looked very promising but I didn't get far with it: it was coming from a high-energy superconducting-magnet background and at the time could not accommodate a rectangular dipole and had no treatment of fringe fields. The developers were friendly and helpful, but there were changes of personnel, and other issues began to crop up so I didn't continue with it.

Now BDSIM seems to have a new life, a new team of developers, and more capabilities such as bends with arbitrary edge angles (not found in G4Beamline). Two interesting aspects of BDSIM are: (1) it uses the MADX syntax, plus extensions, to describe the beam line elements and layout, and to drive the simulation; (2) under favourable conditions (paraxial approximation, in vacuum) it can use efficient transfer maps to track particles through the basic magnetic elements.

20 Conclusion

As with any complex software, working with G4Beamline can be difficult, messy and frustrating. But in spite of its imperfections, I think G4Beamline gives us a glimpse of the future of accelerator physics computing: it will be strongly object-oriented, fully three-dimensional and based around a structured geometry, and will include advanced physics models with greater scope and accuracy than those found in today's accelerator codes.

A willingness to experiment, learn and make mistakes is necessary to undertake any sizable project in G4Beamline. It is very easy to fall into error and produce subtly wrong, or even nonsensical, results. Simulation inputs and outputs must be checked every which way, and re-checked. Comparisons with other codes must be made wherever possible. No anomalous behavior must go undiagnosed.

If it is any consolation, keep in mind that with G4Beamline you are able to do things that would take a team of programmers and experts to do in Geant4 itself.

It seems to me that to really use G4Beamline well and to its full potential (which I have not yet achieved) requires a kind of scientific and engineering discipline for which there are no textbooks: you have to be your own teacher.

References

[1] S. Agostinelli et al., Geant4 – A Simulation Toolkit, *Nucl. Instrum. Meth.* **A 506** (2003). J. Allison et al., Geant4 developments and applications, *IEEE Trans. Nuc. Sci.* **53**:1 (2006).

[2] T.J. Roberts et al., Particle Tracking in Matter-Dominated Beam Lines, IPAC10, Kyoto, 2010. `http://g4beamline.muonsinc.com/`

[3] Tom Roberts, G4Beamline Users Guide 3.02, Muons Inc 2016, `http://g4beamline.muonsinc.com/`

[4] http://root.cern.ch/

[5] Opera-2D and Opera-3D Reference Manuals. http://www.cobham.com/

[6] http://www.coin3d.org/

[7] Particle Data Group,
http://pdg.lbl.gov/2016/reviews/rpp2016-rev-monte-carlo-numbering.pdf

[8] https://twiki.cern.ch/twiki/bin/view/FlukaTeam/FlukaLineBuilder

[9] https://twiki.ph.rhul.ac.uk/twiki/bin/view/PP/JAI/BdSim