

ParticleGenerator

An Athena Monte-Carlo event generator for single particles

William Seligman <seligman@nevis.columbia.edu>

24-Jun-2003

Like SingleParticleGun (see <<http://www-theory.lbl.gov/~ianh/monte/Generators/SingleParticleGun/>>), the ParticleGenerator package runs within Athena and puts single particles into the HepMC transient data store. The difference between SingleParticleGun and ParticleGenerator is that the latter offers greater control over a greater number of variables. The intent of the package is to give the user a means of generating single primary particles normally available within Geant3 or Geant4, but using an Athena jobOptions file as the control mechanism.

Quickstart

For the purposes of this section, it's assumed that you are using the TestRelease package to run an Athena job.

If you only wish to include the packages with which you're working in the TestRelease `cmt/requirements` file, then include these lines:

```
use ParticleGenerator ParticleGenerator-* Generators
use McEventSelector McEventSelector-* Generators
```

In the jobOptions file, load the ParticleGenerator library and algorithm; you'll probably want ParticleGenerator to be the first algorithm invoked (unless you have a task that needs to be performed before event generation):

```
ApplicationMgr.DLLs += { "ParticleGenerator" };
ApplicationMgr.TopAlg = { "ParticleGenerator" };
```

ParticleGenerator has just one parameter: an array of strings. Each string contains an order for a different kinematic variable. The following example demonstrates the range of orders available:

```
ParticleGenerator.orders = {
  "pdgcode: constant 11",
  "energy: constant 629",
  "vertZ: gauss 0.0 50",
  "vertX: loop -10 11 1",
  "eta: sequence 0.1 0.3 0.6 1.2 2.8 3.0 5",
  "phi: flat 0 6.2832"
};
```

The remaining sections of this documentation describe the kinematic variables that can be used in ParticleGenerator, and the different commands that can be assigned to a kinematic variable.

Variables

Table 1: The following kinematic variables are available in ParticleGenerator:

variable	abbrev.	name	description	default command (1)
<i>PDG code</i>	id	pdgcode	particle ID code from PDG	fixed 22 (2)
V_x	vx	vertx	vertex x-position	fixed 0
V_y	vy	verty	vertex y-position	fixed 0
V_z	vz	vertz	vertex z-position	fixed 0
E	e	energy	total energy	fixed 10
E_T	et	transverse	energy transverse to z-axis	fixed 0
p_x (3)	px	momx	momentum direction along the x-axis	fixed 0
p_y	py	momy	momentum direction along the y-axis	fixed 0
p_z	pz	momz	momentum direction along the z-axis	fixed 1
η	eta	eta	eta	fixed 0
θ	theta	theta	theta angle from positive z-axis	fixed 0
ϕ	phi	phi	phi angle from positive x-axis	fixed 0
T_x (4)	tx	targetx	target x-position	fixed 0
T_y	ty	targety	target y-position	fixed 0
T_z	tz	targetz	target z-position	fixed 0
t	t	time	time of particle at vertex	fixed 0
\mathcal{P}_x (5)	polx	polx	polarization along x-axis	fixed 0
\mathcal{P}_y	poly	poly	polarization along y-axis	fixed 0
\mathcal{P}_z	polz	polz	polarization along z-axis	fixed 0

Notes:

1. "default command" is the command used to generate the value of the variable if the user does not supply an order for that variable. If some of the commands in the table seem contradictory, note that the default mode to determine the angle is by θ and ϕ , and the default mode to determine the energy is by using E .
2. In the Particle Data Group's (PDG) scheme of numbering particles in a Monte Carlo, the value 22 corresponds to a γ . For a list of codes, see <http://pdg.lbl.gov/mc_particle_id_contents.html>.
3. Following the convention of Geant4, p_x , p_y , and p_z specify the momentum *direction*, not the projection of the particle momentum onto a given axis. The absolute value of these variables are irrelevant; only their values relative to each other are used.

4. T_x , T_y , and T_z are an alternate method of specifying the particle angle; the particle's θ and ϕ are taken from a straight line drawn from the vertex (x,y,z) to the target (x,y,z) .
5. Following the convention of Geant4, the particle's polarization is given by projections in Cartesian co-ordinates. If you require the polarization given in terms of \mathcal{P}_θ and \mathcal{P}_ϕ , contact the package author at seligman@nevis.columbia.edu and let him know. (In fact, if you use ParticleGenerator to control the polarization at all, please let him know; he strongly suspects that no one will find the feature to be useful.)

The units follow the HepMC convention: all energy and momenta are in *GeV*; distances and time ($c\tau$) are in *mm*.

Modes (or the lack thereof)

The SingleParticleGun package has a parameter for the user to specify a "mode" for how to compute the particle's kinematics: by (E_T, η, ϕ) , by (E, θ, ϕ) , etc. ParticleGenerator does not require a mode to be explicitly specified. Instead, the package examines the user's orders to determine how to calculate the particle's kinematics.

The package examines the variables included in the user's orders. The package will pick one of the following modes to determine θ and ϕ :

- by (θ, ϕ) ;
- by (η, ϕ) ;
- by (p_x, p_y, p_z) (see note (2) in Table 1);
- by (V_x, V_y, V_z) and (T_x, T_y, T_z) (see note (3) in Table 1);
- by the ratio E_T/E , ϕ , and p_z (the last variable is only examined for its sign; if $p_z < 0$ then the particle is sent in the negative z -direction).

If the particle's angle is *not* being determined by the E_T/E ratio, then the particle's energy will be specified by the variable that the user specifies in an order: either E or E_T .

What happens if the user's orders are consistent with more than one of the above modes? For example, what if the user tries the following orders:

```
ParticleGenerator.orders = {
  "id: constant 211",
  "eta: fixed 1.5",
  "theta: loop 0 3.14159 0.157",
  "e: constant 629",
  "transverse: gauss 62.9 5"
};
```

It's not clear whether the user wants to specify the angle with respect to the z -axis by η , by θ , or by the E_T/E ratio. (In any case ϕ will be zero, the default value from Table 1.) In the case of any ambiguity in determining how to generate a particle's kinematics, ParticleGenerator will display a detailed error message, then throw a `GaudiException`.

The moral here is: Look over your orders carefully, and make sure they're consistent.

Commands

The syntax of a ParticleGenerator "order" is:

```
"<variable>: command"
```

...where <variable> is an abbreviation or name from Table 1, and the command is one of the following. Note that orders are not case-sensitive.

"Constant" or "Fixed"

The keyword `constant` or `fixed` (the two words are equivalent) must be followed by a numeric value. This command will assign that value to the variable in the order for every event. For example:

```
ParticleGenerator.orders = {  
  "ID: constant -11",  
  "Vx: fixed 100",  
  "Vy: fixed 30.0",  
  "Vz: fixed -50",  
  "Theta: constant .707",  
  "Phi: fixed 3.14159",  
  "Energy: constant 600."  
};
```

"Uniform" or "Flat"

The keyword `uniform` or `flat` (the two words are equivalent) must be followed by two numeric values. This command will generate a random value uniformly between the two values. The lower value should be specified before the higher one, but if they're reversed they'll be put into the correct order.

An example (note that the default vertex is at (0,0,0)):

```
ParticleGenerator.orders = {  
  "pdgcode: fixed 11",  
  "theta: uniform -2.82 2.82",  
  "phi: flat 0 6.2832",  
  "Et: constant 600."  
};
```

"Gaussian" or "Normal"

The keyword `gaussian` or `normal` (the two words are equivalent) must be followed by two numeric values. This command will generate a random value according to a Gaussian distribution, with a mean of the first value and a width of the second.

For example:

```
ParticleGenerator.orders = {  
  "PDGcode: constant 111",  
  "vertZ: gaussian 0 10",  
  "targetX: fixed 30",  
  "targetY: fixed -4.5",  
  "targetZ: fixed 2.71828",  
  "transverse: normal 6.5 1.0"  
};
```

In this example, the z-vertex will be generated by a gaussian distribution with a mean of 0.0 and

a width of 10.0; E_T will be generated by a gaussian with a mean of 6.5 and a width of 1.0.

"Loop"

The keyword `loop` must be followed by three numeric values: 'start', 'finish', and 'step'. The resulting command will generate a repeating cycle of numbers: the first value will be 'start'; the subsequent values will be the previous value plus 'step'; if the value is not less than 'finish', then the 'start' value will be returned.

This vaguely resembles the behavior of a FORTRAN `DO` loop, but there are some important differences:

- The third value, 'step', is *not* optional. The default 'step' is *not* 1. If you don't supply a third value, a `GaudiException` will be thrown and the job will abort.
- The value returned by the loop will never equal 'finish'. Note the above wording: the loop resets when the value is not less than 'finish'; this isn't the same thing as saying that the loop resets when its value is greater than finish.

For example, if you want a value to loop from 0 to 10 by 1, an appropriate command is:

```
"loop 0 11 1" or even "loop 0 10.1 1".
```

If this behavior seems puzzling, consider:

- It's consistent with how STL containers behave within C++; they are referenced by the half-open interval `[begin,end)`, that is, the iterator `end()` is not included in the container. The `loop` command operates on the interval `[start,finish)`, that is, the value 'finish' is not included in the cycle.
- Even if you supply integer values, the `loop` command works with floating-point numbers. Testing floating-point values for equality can be risky. Depending on the machine and the compiler, adding 0.628318530718 ($\sim\pi/5$) to itself ten times may not give exactly the value 6.28318530718 ($\sim 2\pi$).

Example: If you wanted to vary ϕ from 0 to 360 degrees in 30-degree steps, an appropriate order is `"phi: loop 0 6.28 .5235988"`.

By the way, the `loop` command will work properly with negative steps; for example, `"loop 10 0 -1"` (which will go from 10 down to 1, then go back to 10).

"Sequence"

The keyword `sequence` must be followed by at least one numeric value. The resulting command will generate each value supplied in the command in the order they're given, then go back to the beginning.

For example, given the order `"eta: sequence 0.02 0.1 0.5 1.2 2.8 3.6"`, the first event would have $\eta = 0.02$, the second event would have $\eta = 0.1$, the third event would have $\eta = 0.5$... the sixth event would have $\eta = 3.6$, the seventh event would have $\eta = 0.02$, the eighth would have $\eta = 0.1$, and so on.

For nested loops and sequences: "After"

Both the `loop` and `sequence` commands generate a "cycle" of values; that is, they generate a repeating set of values. (Strictly speaking, this is true of the `uniform` and `gaussian` generators as well, but hopefully the cycle in these commands contains more than 10^{10} values.)

In `ParticleGenerator`, a "cyclic" command has the property that it can nest inside another command. An example may make this clearer: Suppose we have the following set of orders:

```
ParticleGenerator.orders = {  
  "id: constant 22",  
  "VertX: loop -100 101 10",  
  "VertY: loop -100 101 10",  
  "VertZ: loop -100 101 10"  
};
```

Each `loop` command will be cycled for each event, with the following result:

Event #	vertex (x, y, z)
1	(-100, -100, -100)
2	(-90, -90, -90)
3	(-80, -80, -80)
...	...
20	(+90, +90, +90)
21	(+100, +100, +100)
22	(-100, -100, -100)

This may be what the user wants, but probably the desired result is to have the equivalent of a nested FORTRAN `DO` loop. To get this, `ParticleGenerator` offers the keyword `after`, which must be followed by the name or abbreviation of a kinematic variable.

If the above example is modified to implement nested loops, it looks like:

```
ParticleGenerator.orders = {  
  "id: constant 22",  
  "VertX: loop -100 101 10",  
  "VertY: loop -100 101 10 after VertX",  
  "VertZ: loop -100 101 10 after VertY"  
};
```

The result will be:

Event #	vertex (x, y, z)
1	(-100, -100, -100)
2	(- 90, -100, -100)
3	(-80, -100, -100)
...	...
20	(+90, -100, -100)
21	(+100, -100, -100)
22	(-100, -90, -100)

An english translation of "VertY: loop -100 101 10 after VertX" is: only move to the next value in the VertY cycle after VertX has gone through one entire cycle.

Here's another example of using nested cycles:

```
ParticleGenerator.orders = {  
  "pdgcode: constant -112",  
  "theta: sequence 0.785398 -0.785398",  
  "phi: loop 0 6.28 .5235988 After theta",  
  "Et: sequence 300 500 700 1000 After phi"  
};
```

The above set of orders sets θ to +45 degrees, then to -45 degrees; the ϕ loop is incremented after each value of θ is used; the E_T sequence goes to its next value when the ϕ loop goes back to 0.

Notes and Warnings

- ParticleGenerator is flexible when it comes to interpreting orders: case is ignored; only the first letter of a word is used to identify keywords (except for `fixed` or `flat`, which require two letters to distinguish them from each other); extraneous words between numeric values are ignored; the `after` keyword + kinematic name can occur anywhere after the keywords `loop` or `sequence`; equals signs (=) are converted into spaces; the kinematic variable that begins an order can be followed by a space or other punctuation instead of a colon (:).

The practical upshot of the preceding paragraph is that ParticleGenerator is fairly tolerant of typographic errors, or of comments inserted into orders. For example, all of the following orders are interpreted identically (the z -vertex is given by a gaussian distribution with a mean of 0 and a width of 10):

```
"vz: gaussian 0 10"  
"vertZ: Gauss +0.0 10.0"  
"Vz: NorMal 0 1E+1"  
"VeRTZ=Normal Mean=0 Width=10"  
"veRtz g00fy This Is Ignored 0 By ParticleGenerator 10 Most Of the Time"  
"vz: gauss width=0 mean=10"
```

Note the last line; don't be fooled by extraneous text. It's the order of the values in the command that determines their meaning, not the comments.

- ParticleGenerator "parses" each order and stores it in an internal format. You may not see the exact text of your orders in ParticleGenerator's error messages; it will display the interpreted version of the order.
- The order in which you give your orders is irrelevant.
- If you're trying to determine how ParticleGenerator is interpreting your orders, or you want to see the values that are generated for each event, set `MsgService.OutputLevel` to `DEBUG` or `VERBOSE`.
- It is very easy to "out-think" ParticleGenerator. For example, you may think, "If I supply target (x,y,z) , E_T , V_x , V_y , and ϕ , ParticleGenerator should be able to determine V_z ." No, it can't. ParticleGenerator is limited to the modes described above.
- You can only nest loops or sequences within other loops and sequences. If you put "after vz" in a loop command, and the V_z order is a gaussian, you'll get an error message.

- The value of PDG code will be converted to an integer. Most of the time, you'll probably want a fixed value for the particle ID (e.g., "ID: fixed 11" to generate electrons), or perhaps a sequence ("PDGcode: sequence 112 -112" will alternate between π^+ and π^-). However, ParticleGenerator does *not* check to see if the command for the PDG code will always return an integer value; you can put in the command "id: gaussian 14 1" but there's no guarantee that you'll get a meaningful result.

Potential Improvements

Here are some improvements that may be made to ParticleGenerator, if someone requests it:

- Other commands, if they would be useful (Lorentz distributions? User-supplied functions?).
- The option of adding units to an order; e.g., "Phi: 0 360 30 degrees" or "Et: fixed 300 MeV".
- Add more kinematic variables; e.g., \mathcal{P}_0 and \mathcal{P}_ϕ as suggested above.
- Allow multiple particles to be specified; e.g.,

```
ParticleGenerator.orders = {
  "id[1]: constant 11",
  "id[2]: constant 11",
  "theta[1]: gaussian 0 .100",
  "vertz[1]: fixed 10",
  "theta[2]: fixed .050",
  "vertz[2]: flat -5 5"
};
```

Example files

In the package directory Generators/ParticleGenerator/share are several files that may be useful:

- requirements.TestRelease - This file can be used as a TestRelease/cmt/requirements file in order to test ParticleGenerator.
- jobOptions_ParticleGenerator.txt - This file can be placed in TestRelease/run/. With the previous file copied to TestRelease/cmt/requirements, this jobOptions file will run just the ParticleGenerator algorithm.

Note that, like all Generator algorithms, the 'athena' command must be executed with TestRelease/run as the current working directory; otherwise, the Generators package will not be able to find the PDGTABLE file and the job will crash with a segmentation fault.

Acknowledgements

Mikhail Leltchouk made the initial request for the functionality of ParticleGenerator to be implemented as a command-line extension to Geant4; he did so in 1999, so it only took four years to find time to implement it (and it's part of Athena, not Geant4). The original idea behind ParticleGenerator was inspired by the GKINE command-line functionality available in Geant3. The structure of ParticleGenerator is based on that of SingleParticleGun, as prepared by Marjorie Shapiro and Ian Hinchliffe.

Giorgos Stavropoulos provided a tremendous amount of assistance during ParticleGenerator development. The advice of Paolo Calafiura was invaluable as well.

While under development, this package was named ParticleCannon, ParticleRifle, ParticleFire, and ParticleCommand. Fortunately, for the final name, common sense prevailed.