The GlastRandomSvc

Michael Kuss
INFN Pisa
Michael.Kuss@pi.infn.it

v1.0.0 (June 14, 2012)

1 Introduction

This manual was born from the need for a general description of the way random number generators are being used in Fermi Gamma-ray Space Telescope[1] (Fermi-GRST, formerly GLAST) Large Area Telescope (LAT) software packages, as well as for practical advices for those who run mass-simulation tasks using the GlastRelease package. It is not a description of Monte-Carlo techniques in general, the Geant4[2] toolkit for the simulation of the passage of particles through matter, or the entire GlastRelease[3] package. Who doesn't like to waste time to read everything can immediately skip to the section 3 called "Recommedations", on p. 12. However, for details the reader may have to consult the other sections too.
GlastRelease contains 7 packages which need to generate random numbers during a simulation run. These 7 engines are:

- AcdDigiRandom
- CalXtalRespRandom
- FluxSvcRandom
- G4GeneratorRandom
- OverlayRandom
- RootIoRandom
- TkrDigiRandom

Each of them implements it's own random number engine using a Gaudi[4] service. Most of these services interact with Geant4 based packages for the particle and detector response

[1] http://fermi.gsfc.nasa.gov/
[2] http://geant4.cern.ch/ and references therein.
[3] See e.g. GLAST LAT full simulation, L. Baldini et *al.*, Nucl. Phys. Proc. Suppl. 150:62, 2006.
[4] http://proj-gaudi.web.cern.ch/proj-gaudi/

generation. Since Geant4 is very much interweaved with another CERN project, CLHEP[5], it was a natural choice to use CLHEP's random module (actually, the CLHEP random module is based on Geant4's HEP random module).
Our default random number engine is the CLHEP::TripleRand[6] engine. It has a huge sequence length of order of 2^{159} (10^{48}), and all it's 2^{32} potential sequences are different. To generate one single event about 20k random number calls are needed[7]. Thus, for a run with 50k events about 1G (10^9) random numbers have to be generated.
TripleRand's SetSeed() method accepts two parameters. The first is the seed, of type long, the second is called luxury and of type int. However, luxury is not being used, it's only there because of inheritance from the CLHEP::HepRandom class. On 32 bit systems, a long is identical to an int. On 64 bit systems, however, a long has 8 bytes, allowing 2^{64} (10^{19}) different values for the seed. But, within TripleRand the seed is implicitly type casted to an unsigned int[8] (which is 4 byte on any platform, 32 bit and 64 bit). Hence, it is prudent to use integer numbers in the range $[0 \dots 2^{31}[$ (i.e. $[0 \dots 2\text{Gi}[$) only.
Other pitfalls with random number generators are that even seeds may cause short sequence lengths[9], as well as negative seeds[10]. From the beginning, the code of GlastRandomSvc didn't generate even seeds and was trying to avoid negative ones.

1.1 Reproducibility

Restricting ourselves to only odd numbers in the range $[0 \dots 2^{31}[$, we are left with 1Gi possible seeds for the random number generator. Let's neglect for the moment that with each of these seeds we can generate a sequence of 10^{48} random numbers. When the first version of the GlastRandomSvc was designed in 2002, tasks to simulate more than 1Gi events were not foreseen. Thus, having 1Gi seeds available it was natural to assign a distinct seed to each event to be simulated.
There was also a good reason for doing this. Let's assume a run with 20k events (the first version of the GlastRandomSvc allowed up to 100k events). In 2002, the simulation of these 20k events took many hours. Finished, examining the result with the event display we note that event 12345 is a γ conversion into an e^+e^- pair with well developed tracks, but somehow our reconstruction code doesn't find the tracks. We have an idea how to help the algorithm, and implement it. Before validating the new code on all events we would like to check it on this particular event. Thus, we tell Gaudi to start with event 12345. However, if we would set the seed only once at the beginning of the run, we wouldn't get the same event. Depending on how we set the seed the new event 12345

[5] http://proj-clhep.web.cern.ch/

[6] http://proj-clhep.web.cern.ch/proj-clhep/manual/RefGuide/Random/TripleRand.html

[7] This was determined using GlastRelease v12r16 with a modified CLHEP::TripleRand class, for the muon_pencil source class.

[8] The gcc standard requires that a signed to unsigned conversion preserves as many as possible low-order bits of the source that will fit into the destination. Thus, for 32 bit builds negative seeds will be mapped to positive seeds in the range $[2^{31} \dots 2^{32}[$, while for 64 bit builds numbers in the range $[0 \dots 2^{32}[$ will preserve their value and for all other values the high-order bits will be stripped.

[9] Sorry, can't find the reference anymore!

[10] Check e.g. the file Random/doc/JamesRandomSeeding.txt in the CLHEP source distribution. Though, HepJamesRandom and TripleRand are very different.

would be identical to the old event 0 (using only the run number to set the seed), or completely different (if we would use the run number and the event id of the first event of the run). Rerunning all events from 0 up to 12345 is also not an option. We would like to determine the optimal parameters for the new algorithm, and waiting each time several hours to see the result is annoying. But, if we would set the seed for each event based on it's run number and event id, we could tell Gaudi to precisely simulate this particular event, and tune our algorithm fast.
Does it work? Well, kind of. First, I should say that thoroughly this was last tested 8 years ago[11]. For each build (optimized and non-optimized) two sequences of events were simulated. One sequence started with event id 0, the other one with an offset (usually 1). This was repeated on another computer with the same architecture. The results of 8 years ago were:

- sequences produced with non-optimized code were identical.
- sequences produced with optimized code were identical.
- optimized and non-optimized code produced different sequences. I don't recall if this meant completely different events or similar but non identical ones.
- sequences produced with the same code but on different computers are identical to better than 99.7%.

The above was and is valid for simple sources only. Composite sources won't reproduce at all, due to the way the various components are handled by FluxSvcRandom. Too, I can't imagine that reproducibility will work for Overlay runs.
And obviously, seeding the random number generator only once at the beginning of the run will generate completely different sequences of events for different starting event ids.

1.1.1 CMT vs. Scons Builds

Initially, the order how the random engines are created was different for scons and CMT builds. However, it seems that this issue is solved. Currently, the system tests are modified to allow for the comparison of scons and CMT builds.

1.2 Linking

A feature which creates additional issues is the difference in linking to the CLHEP and Geant4 libraries under Linux and Windows. On Windows, we link statically. Static linking creates at each linkage a separate engine object. Hence, depending on which services are being used for the simulation, the final executable may involve up to 7 engines. Instead, on Linux we link shared. The separate engines will share a pointer to the same base object. Effectively, under Linux we have only one engine instantiation.

[11] See https://jira.slac.stanford.edu/browse/GRINF-5, https://jira.slac.stanford.edu/browse/GRINF-10, and http://www.pi.infn.it/ kuss/glast/randoms/RandomsReport040826.html. Actually, involuntarily a test was performed in March 2012, with "good" results, see Sec. 2.4.1.

2 GlastSvc

GlastSvc is a package which bundles several services used for data analysis and simulation. Of interest here is the GlastRandomSvc only. It has undergone various modifications in the past. I will only describe the last two incarnations, because all older ones are not being used anymore since long.
The GlastRandomSvc doesn't have it's own tags, being part of GlastSvc. Unfortunately, the major version number of the GlastSvc doesn't exactly relate to the code advancement of the GlastRandomSvc. Thus, I will coin a label Xn, with n more or less inline with the GlastSvc version. I will start with the most recent code, and proceed back in time.

2.1 GlastRandomSvc X10

This variant is the current code. It is used by the following tags:

- GlastSvc-10-00-01 and newer tags (on the MAIN branch). They are used by GlastRelease-18-08-06 and newer, all v19, and all v20.
- GlastSvc-09-28-04-gr01 and newer tags (on the GlastReleasev17r35p24 branch). They are only used by all GlastRelease-17-35-24-ccnn branch tags, except for gr01 - gr13, which use X9 (Sec. 2.2). Seems gr14 and newer, and all ghost (gh), L1 processing (lp), reprocessing (rp), and truncation (tr) patches use X10.

2.1.1 Job Option Parameters

Job option parameters can be used to change the program flow without changing the actual code. Following I will list the job option parameters available for the GlastRandomSvc. The values in brackets are the default values.

RandomEngine("TripleRand") is the CLHEP random engine (see Sec. 1) we use. Our default random engine is the CLHEP::TripleRand engine. To my knowledge we have never used a different one. It is not recommended to change to another engine for production runs. And, if you decide to do so, you're on your own!

RunNumber(-1)

RunNumberString("$(runName)") : there are four methods to specify the run number, in order of precedence:

1. setting GlastSvc.RunNumber to any positive signed int number.
2. setting the environment variable runName to any positive signed int number.
3. setting GlastSvc.RunNumberString to any positive signed int number.
4. if the run number wasn't set with any preceding method, or was set to the value -1, it's being set to 10.

InitialSequenceNumber(0) is the event id of the first event (not necessarily the first that triggers!) of the run. Since it is stored as a Gaudi IntegerProperty, it's range is limited to the range of a signed int.

autoSeed(false) was used for a different purpose once, thus the misleading name. A better one would be "eventSeeded".
If autoSeed is set to true (event seeding), the random engine is seeded for each event, based on run number and current event id. I.e., for each event the random number engine is set to a well defined state. Sloppy speaking, each event has it's own distinct seed.
If autoSeed is set to false (run seeding), the random engine is seeded only once per run, using the run number and initial sequence number. For subsequent events the initial state of the engine is not well defined but as left behind by the respective preceding event.

NumberOfRuns(autoSeed?20000:20000000) specifies the number of runs of a task (a mass simulation). In older code, either a constant value of 20000 (for event seeding, see Sec. 2.4) or the constant bigEnough, preset to 1000000 (for run seeding) was used. It defaults to 20000 (20k) for autoSeed=true and to 20000000 (20M) for autoSeed=false. Changing NumberOfRuns changes the algorithm to calculate the seeds, so it has to be set to the same value for all runs of a task, to avoid possible event duplication!

Seed(0) can be used for shared linked builds only (Linux), and there only for run seeding. If the value of Seed is different from 0, the run is seeded with this value rather than calculating the seed. This parameter can be used as a last resort if a task is messed up. I.e. imagine a series of runs which were simulated using one setting of NumberOfRuns, supplemented by another series of runs with another setting. It's very probable that some runs were performed using the same seed. Eliminating the duplicate runs and reconstructing the seeds which were used in the good runs, for the remaining runs the unused seeds could be used, avoiding a restart from scratch.

pedantic(true) : as described in Sec. 2.1.2, GlastRandomSvc checks for conditions which could cause event duplication, and logs a warning. pedantic=true (the default) causes GlastRandomSvc to terminate with StatusCode::FAILURE if any of these conditions are given. If the user is willing to take the risk this behavior can be avoided setting pedantic to false. However, then the user is on it's own!

2.1.2 Sanity Checks

X10 checks for conditions which surely will (or potentially may) result in the use of negative seeds or duplicate events, or are insane anyway. These conditions are:

- The run number should not be negative.
- The initial sequence number should not be negative.
- The run number should be smaller than the total number of runs.

- The number of random number engines should be either 1 (shared linking) or equal to the size of engineMap. Everything else is hinting for a problem.
- The maximum event id used for seeding (InitialSequenceNumber+ApplicationMgr-.EvtMax-1 for event seeded runs, InitialSequenceNumber for run seeded runs) should not exceed m_maxEventId (see Sec. 2.1.3).

If one of these conditions is encountered, a warning will be logged. If pedantic is set to true (the default), GlastRandomSvc will terminate with StatusCode::FAILURE. If it is set to false, the code will proceed trusting in the wisdom of the user.
If a seed is user specified through the Seed job option parameter, several conditions are checked. Setting the seed to the value of Seed is invalid for:

- event seeded runs.
- for runs with multiple engine instantiations.

I consider it insane to set the same seed for all events of a run, or all engines created. The code simply will ignore the directive, point.
Setting the seed to the value of Seed is dangerous if Seed is:

- negative.
- even.

For all these conditions above, only a warning is logged, GlastRandomSvc will not terminate even in the case of pedantic set to true. I don't remember my reasoning behind this, and may change it in future.

2.1.3 The Code

The GlastRandomSvc::applySeeds() method is called once for each event in case of event seeding, and only once for the entire run for run seeding. The following is an excerpt of the code:

```
void GlastRandomSvc::applySeeds(int runNo, int seqNo) {
    int engineNum = 1;
    const int luxury = 0; // 2nd argument to setSeed
    EngineMap::const_iterator dllEngine;
    EngineMap m_engineMap = m_randObs->getEngineMap();
    int m_engineNum = m_randObs->getEngineNum();
    if (m_engineNum == 0)
       m_engineNum = 1;
    for ( dllEngine=m_engineMap.begin(); dllEngine!=m_engineMap.end(); ++dllEngine ) {
        long theSeed;
        if ( m_seed )
            theSeed = m_seed;
        else
            theSeed = 2 * (
                        (runNo%m_numberOfRuns)*m_engineNum*(m_maxEventId+1)
                        + (engineNum-1)*(m_maxEventId+1) + seqNo
```

```
                        )
                + 1;
        dllEngine->second->setSeed(theSeed, luxury);
        // use engineNum only if there are multiple engine instantiations (Windows)
        if ( m_engineNum > 1 )
            ++engineNum;
    }
}
```

Let's try to examine the code. In case that m_seed was initialized from GlastRandomSvc.Seed, the seed is simply set to m_seed. Otherwise, it is calculated, based on:

- runNo, the run number
- seqNo, the event id
- engineNum, the engine position in m_engineMap
- m_engineNum, the number of engine instantiations, one for Linux, multiple for Windows
- m_numberOfRuns, the number of distinct run numbers
- m_maxEventId, the maximum safe event id for given engineNum and m_numberOfRuns

Because I don't assume we will ever make mass simulations on Windows, I will consider the case of Linux only. Thus, engineNum and m_engineNum are unity, and the code for the seed simplifies to:

$$\text{theSeed} = 2 * ((\text{runNo}\%\text{m\_numberOfRuns}) * (\text{m\_maxEventId} + 1) + \text{seqNo}) + 1; \quad (1)$$

Remember that we use odd seeds only. Furthermore, we should avoid negative seeds. Thus, good seeds are all odd numbers in the range $[1 \ldots 2^{31}[$, 1Gi seeds in total.
The term in the outermost brackets of Eq.(1) has to fulfill the following condition:

$$(\text{runNo}\%\text{m\_numberOfRuns}) * (\text{m\_maxEventId} + 1) + \text{seqNo} \leq 2^{30} - 1$$

runNo%m_numberOfRuns cannot exceed m_numberOfRuns-1, and seqNo not m_maxEventId, respectively. Thus:

$$(\text{m\_numberOfRuns} - 1) * (\text{m\_maxEventId} + 1) + \text{m\_maxEventId} \leq 2^{30} - 1$$

leads to

$$\text{m\_maxEventId} \leq 2^{30}/\text{m\_numberOfRuns} - 1. \quad (2)$$

Indeed, in GlastRandomSvc m_maxEventId is set to the value to the right of Eq.(2). It's not a job option parameter, i.e. not under control of the user.
Let's summarize this section with two examples:

1. autoSeed=true (event seeding): the default are 20000 runs, but for the example I choose 2^{14} (16384) runs[12]. First, we choose run numbers in the range $[0 \ldots 16383]$. Allowing for 16384 runs, the maximum event id permitted in each run is 65535 (i.e. 65536 event ids in the range $[0 \ldots 65535]$). Hence, ApplicationMgr.EvtMax would have to be set to 65536 or less. What would happen if we would choose to have more events in a run?
   Do the math, event i in run n+1 would have the same seed as event i+65536 in run n!

2. autoSeed=false (run seeding): the default are 20M runs. Again, choose run numbers in the range $[0 \ldots 20000000[$. Having 20M runs, the maximum event id that could be used for seeding the random number generator is 52.6870912, i.e. 52. However, remember we set the seed only once at the beginning of the run, using the run number and the initial sequence number (the event id of the first event). Hence, with run seeding we are limited in our choice of the initial sequence number. Only if for all runs of a task the initial sequence numbers (they could be different) are within the allowed range, it is guaranteed that

   - no two runs use the same seed
   - no run uses a negative seed

2.2 GlastRandomSvc X9

Versions prior to X9 allowed only event seeding. The implementation of run seeding started with GlastSvc-09-23-01, but several patches were needed to make it work. Attention, GlastRandomSvc X9 is also in GlastSvc branch tags 09-28-03-gr01 and older. Some older GlastRelease versions still in use today contain one of these GlastSvc versions.

- GlastSvc 09-26-00 - 09-28-04 (MAIN branch), i.e. all GlastRelease v17 (except for a few versions which contain X10, see Sec. 2.1), and GlastRelease-18-08-06 and older.
- GlastSvc-09-26-01-gr1 (GlastReleasev15r47p12 branch), in GR-15-47-12-gr08 - gr21.

2.3 Job Option Parameters

The job option parameters of X9 are (default values are in brackets):

- RandomEngine("TripleRand")
- RunNumber(-1)
- RunNumberString("$(runName)")
- InitialSequenceNumber(0)

[12] I admit I made it unnecessary complicated by not having chosen the number of runs as an exponent of 2.

- autoSeed(): attention, the default was changed a few times. To avoid unpleasant surprises, set autoSeed to the desired value (true for event seeding, false for run seeding) ALWAYS!

These parameters are identical to those of X10. For details check their description in Sec. 2.1.1.

2.4 The Code

In the following code snippet log and other non relevant statements have been removed.

```
void GlastRandomSvc::applySeeds(int runNo, int seqNo) {
  int multiplier = 1;
  int dummy = 0; // for 2nd argument to setSeed
  EngineMap::const_iterator dllEngine;
  EngineMap m_engineMap = m_randObs->getEngineMap();
  for ( dllEngine=m_engineMap.begin(); dllEngine!=m_engineMap.end(); ++dllEngine ) {
    long theSeed;
    if (m_autoSeed) {
      theSeed = multiplier * 100000 * ((runNo+1) % 20000) + 2*seqNo+1;
    }
    else {
      long bigEnough = 1000000;
      theSeed = 2 * ( (runNo+1)%bigEnough + multiplier * bigEnough * (seqNo+1) ) + 1;
    }
    dllEngine->second->setSeed(theSeed,dummy);
    ++multiplier;
  }
}
```

The big difference to X10 is that X9 used the multiplier indifferently also for Linux. As was explained in Sec. 1.2, for Windows builds we link statically against the CLHEP libraries, resulting in a separate random number engine instantiation for each random service. If we would seed all these engines with the same value, the random number sequences generated for each engine would be identical. E.g., the 5^{th} number of G4GeneratorRandom could represent the particle energy (it does not, this is an example), while the identical 5^{th} number of TkrDigiRandom could be the number of noise hits to be generated. To avoid these subtle correlations, each engine needs separate seeds, which is realized by using a counter (multiplier).
For Linux, a multiplier wouldn't be necessary, but the code of X9 (actually, all versions prior to X10) is common for Windows and Linux, unfortunately. In any case m_engineMap contains all engines, 6 or 7, only for Linux all share a common pointer to the base engine. Hence, we have to loop over m_engineMap, and set a different seed each time.
What determines the seed in Linux code which is effectively used? Naively one could assume that the last engine in m_engineMap determines the seed, thus the relevant multiplier would be m_engineMap.size(). However, it is the last engine being declared!
This is an excerpt from the log of an Overlay simulation with a CMT built GlastRelease:

```
GlastRandomSvc       INFO Setting run number from environment variable "runName"
GlastRandomSvc       INFO Run number set to: 0
GlastRandomSvc       INFO ================================================
GlastRandomSvc       INFO Setting CLHEP Engine TripleRand for TkrDigiRandom at 0xa38fbd0
GlastRandomSvc       INFO RandGauss setFlag at 1
GlastRandomSvc       INFO Setting CLHEP Engine TripleRand for AcdDigiRandom at 0xa38fe10
GlastRandomSvc    WARNING  Previous engine (0xa38fbd0) for AcdDigiRandom was also set for TkrDigiRandom
GlastRandomSvc       INFO RandGauss setFlag at 1
GlastRandomSvc       INFO Setting CLHEP Engine TripleRand for RootIoRandom at 0xa390108
GlastRandomSvc    WARNING  Previous engine (0xa38fe10) for RootIoRandom was also set for AcdDigiRandom
GlastRandomSvc       INFO RandGauss setFlag at 1
GlastRandomSvc       INFO Setting CLHEP Engine TripleRand for G4GeneratorRandom at 0xa390180
GlastRandomSvc    WARNING  Previous engine (0xa390108) for G4GeneratorRandom was also set for RootIoRandom
GlastRandomSvc       INFO RandGauss setFlag at 1
GlastRandomSvc       INFO Setting CLHEP Engine TripleRand for FluxSvcRandom at 0xa38fff8
GlastRandomSvc    WARNING  Previous engine (0xa390180) for FluxSvcRandom was also set for G4GeneratorRandom
GlastRandomSvc       INFO RandGauss setFlag at 1
GlastRandomSvc       INFO Setting CLHEP Engine TripleRand for CalXtalRespRandom at 0xa3903d0
GlastRandomSvc    WARNING  Previous engine (0xa38fff8) for CalXtalRespRandom was also set for FluxSvcRandom
GlastRandomSvc       INFO RandGauss setFlag at 1
GlastRandomSvc       INFO Setting CLHEP Engine TripleRand for OverlayRandom at 0xa390500
GlastRandomSvc    WARNING  Previous engine (0xa3903d0) for OverlayRandom was also set for CalXtalRespRandom
GlastRandomSvc       INFO RandGauss setFlag at 1
```

Note that OverlayRandom is the last engine instantiated. Only setting it's seed sets also the seed of the base engine. All other SetSeed() operations have no effect! Because the engines in m_engineMap are ordered alphabetically, OverlayRandom is the 5^{th} and thus multiplier becomes 5. For a simulation without OverlayRandom, CalXtalRespRandom is the last to be initialized. It's the second in m_engineMap, thus here the multiplier is 2! For scons builds, this could be different (but see Sec. 1.1.1)!

Bad news: it really is a mess!

Good news: scons builds of GR aren't used for production yet[13]. So, we have to consider only CMT RM builds with multipliers of 2 and 5 only.

Let's go back to the code and study it in more detail!

2.4.1 Event Seeding

```
long theSeed = multiplier * 100000 * ((runNo+1) % 20000) + 2*seqNo+1;
```

Again, runNo can run in the range $[0 \ldots 20000[$.
For a multiplier of 2 (non-Overlay simulations), only run numbers up to 10737 lead to non-negative seeds. Fortunately, even run number 19999 will not cause an overflow of the 4 byte internal seed. For each run, we could use initial sequence numbers up to 99999, resulting in 1Gi (2^{30}) events with odd positive seeds, or even 2G events including also those with odd negative seeds. Note that there is nothing that protects from using negative seeds, as imposed by X10 (see Sec. 2.1.2).
For a multiplier of 5 (Overlay simulations), only run numbers up to 4294 lead to non-negative seeds. In run number 8588 event ids exceeding 233647 will result in an overflow of the 4-byte value of the seed, causing potential involuntary reuse of seeds and thus

[13] And, once they are, we may not use a GlastSvc with X9 anymore.

duplicated events. Exactly this happened for the All-Gamma runs AG-GR-v17r35p14-IRFS76BK-allE and AG-GR-v17r35p14-IRFS76BK-lowE[14]! For each run, we could use initial sequence numbers up to 249999, leading to 1Gi (2^{30}) events with odd positive seeds, or 2Gi (2^{31}) events including also those with odd negative seeds. Careful, there are also more than 2.85G events whose seeds overflow, i.e. potentially duplicate events. Again, there is nothing that protects from involuntarily using negative seeds as well as from overflowing the 4 byte seed register.
Note that the number of good events (1Gi with odd positive seeds, 2Gi with also odd negative seeds) doesn't depend on the value of the multiplier. It's simply the product of runNo and ApplicationMgr.EvtMax. But, for each multiplier runNo and EvtMax have to be chosen carefully following the two examples above.

2.4.2 Run seeding

$$\text{theSeed} = 2 * ((\text{runNo} + 1)\%\text{bigEnough} + \text{multiplier} * \text{bigEnough} * (\text{seqNo} + 1)) + 1; \quad (3)$$

First I have to admit that I introduced in X9 another name monster here, bigEnough. BigEnough was a const int set to 1M, and was used in run seeding only. Thus, runNo can run in the range $[0 \ldots 1000000[$. Curious, eventually bigEnough was not big enough, and disappeared again in X10.
Note that from a technical point of view this code has a different structure than all other codes encountered so far. Usually, runNo gives the leading contribution to the seed. I.e., a runNo of 1 gives a base summand of 200000 (for multiplier 2), runNo 2 one of 400000, etc.. The initial sequence numbers of 0, 1, 2, etc. give summands of 1, 3, 5, respectively, thus filling the gap between e.g. 200000 and 400000. Here, initial sequence numbers of 0, 1, 2, etc. give a base summand (multiplier 2) of 2M, 4M, and 6M, respectively[15], and the run numbers fill the gap.
For initial sequence number of 0 (the default), the 1M runs would have odd seeds in the range $[1 \ldots 2000000[$, independent of bigEnough and multiplier. What is the valid range for the initial sequence number? A gross estimate gives that the valid range is about $[0 \ldots 1000/\text{multiplier}]$, so there are more than enough initial sequence numbers to choose from.

2.5 GlastRandomSvc X8

Version X8 allowed event seeding only. It was used in GlastSvc tags older than v9r23. Since this old code is not in use anymore, X8 is not of interest anymore. Anyway, the description in Sec. 2.4 is valid, the code for event seeding wasn't changed going from X8 to X9.

[14] See e.g. https://confluence.slac.stanford.edu/display/SCIGRPS/2012/02/14/More+on+duplicated-+events+in+back-entering+allGamma+runs.

[15] Because of seqNo+1, this code doesn't use seeds lower than 4M. I don't think I had a good reason for doing this. I would call it a bug. Also runNo+1 is awkward, but doesn't do harm.

3 Recommendations for X10

This section should suffice for those who won't like to read all the gory details of this document.
Whenever possible, use a GlastRelease using GlastRandomSvc X10 (see Sec. 2.1) or newer. If you need to use X9, read Sec. 2.2.

1. Keep in mind that whatever you choose to do, in any case you have always 1Gi odd positive (good) seeds that can be used.

2. Use InitialSequenceNumber=0 (it's the default), unless you have a good reason to do different. A good reason would be that the run with initial sequence number 0 segfaults. You need a run with this particular run number? Then try with InitialSequenceNumber 1! Or 2![16]

3. Decide how many events you need for your task.

4. autoSeed: if you need more than 1Gi events, you need to use run seeding (autoSeed=false) in any case. If you need event by event reproducibility, you need event seeding (autoSeed=true), and are limited to 1Gi events.

5. Decide how many events you would like to have per run (ApplicationMgr.EvtMax). In general, this boils down to how long you can let Gleam run, either because of CPU time limits or because of memory consumption due to memory leaks.

Last, you need to choose the number of runs. Here, we need to distinguish between run and event seeding.

3.1 Run Seeding

Calculate how many runs you need. Keep in mind that once you have decided on a number of runs, you can't change your mind anymore. E.g., if you decide for 1Mi runs, you have 1024 choices for the initial sequence number. Do you need so many initial sequence numbers? Why not setting NumberOfRuns to 16Mi runs anyway, as a safety margin? You still would have 32 initial sequence numbers to choose from. Is that not enough?
Example: you have simulated 50G events, in 1Mi (2^{20}) runs with 50k events each. Later you realize that 200G events would be better.
Scenario 1: you used NumberOfRuns=2^{20} and InitialSequenceNumber 0. Can you simulate more runs? Yes you can, but it's really tedious! You can't simply increase NumberOfRuns, as can be deduced from Eqs.(1) and (2) on p. 7, and illustrated in Tab. 1 on p. 13:
Changing NumberOfRuns changes the value of m_maxEventId[17] which changes the seeds for the various run numbers.

[16] Recipe of Tom Glanzman used in one Monte-Carlo task.

[17] In a future version Eq.(1) should be modified to a structure similar of Eq.(3). If all runs would be done with the initial sequence number set to 0, a later increase of the number of runs could be possible.

NumberOfRuns		1Mi	2Mi	4Mi	8Mi
m_maxEventId + 1		1024	512	256	128
run number	0	1	1	1	1
...	1	2049	1025	513	257
...	2	4097	2049	1025	513
...	3	6145	3073	1537	769
...	4	8193	4097	2049	1025
...	5	10241	5121	2561	1281
...	6	12289	6145	3073	1537
...	7	14337	7169	3585	1793
...	8	16385	8193	4097	2049

Table 1: The seed as function of run number, for four different pairs of NumberOfRuns/m_maxEventId+1. InitialSequenceNumber is 0 in all cases.

The only resort would be to get a list of seeds already used, and seed each additional run manually using the Seed job option parameter (see Sec. 2.1.1).
Scenario 2: you used NumberOfRuns=2^{24} (16Mi) and InitialSequenceNumber 0. Remember, you simulated 1Mi runs. Can you simulate more runs? Sure you can, just choose run numbers you haven't used before! The safety margin pays off!

3.2 Event Seeding

Calculate how many runs you need. Keep in mind that once you have decided on a number of runs, you can't change your mind anymore. The product of NumberOfRuns and ApplicationMgr.EvtMax has to be less or equal 2^{30} (1Gi).
Scenario: you used NumberOfRuns=20000 (the default), and simulated 10000 runs with InitialSequenceNumber=0 and EvtMax=20000, for a total of 200M events. Now, you realize you need 500M events. What can you do?
Solution 1: obviously, you can simulate 10000 more runs, because you have used only 10000 run numbers so far, with EvtMax=30000.
Solution 2: you don't create new runs, but increase the number of events for each run number. How? Set EvtMax to 50000, and InitialSequenceNumber to 20000, and reuse the same 10000 run numbers. This will create 30000 events with event ids in the range [20000 ... 50000[. Remember, EvtMax should never exceed 2^{30}/NumberOfRuns!