JSF Users Guide
(Draft)

Akiya Miyamoto
KEK
E-Mail: akiya.miyamoto@kek.jp

November 7, 2001

Contents

1 Frequently Asked Questions

1.1 Q1: How to turn on/off the decay mode of W and Z in BASES calculation?

In Physsim generator, the range of the decay mode identification number (see sub Sec.1.2). for W and Z decay can be specified in the file, `jsf.conf`. The name of parameters are, `ENWBases.WpModesLo` and `ENWBases.WpModesHi` in the case of ENW generator and `ZZBases.Z1ModesLo`, `ZZBases.Z1ModesHi`, `ZZBases.Z2ModesLo` and `ZZBases.Z2ModesHi` in the case of ZZ generator. These parameter specify the lower limit and upper limit of allowed decay mode of W and Z. The sample files are `$(KFLIBROOT)/wz/ENWStudy/prod/jsf.conf` and `$(KFLIBROOT)/wz/ZZStudy/prod/jsf.conf`. The other generators have also similar name of parameters.

1.2 Q2: Decay mode identification number.

First of all, read the file, `$(KFLIBROOT)/README`. There are valuable information there.

In Physsim generator, each decay mode of W and Z has decay mode ID as shown in the Table 1.

Table 1: *W and Z decay mode and their ID number used in Physsim generator.*

ID number	Z decay	W decay
1	$\nu_\ell \bar{\nu}_\ell$	$\ell \nu_\ell$
2	$\nu_\mu \bar{\nu}_\mu$	$mu \nu_\mu$
3	$\nu_\tau \bar{\nu}_\tau$	$\tau \nu_\tau$
4	$\ell \bar{\ell}$	$d\bar{u}$
5	$\mu \bar{\mu}$	$d\bar{c}$
6	$\tau \bar{\tau}$	$d\bar{t}$ (forbidden)
7	$u\bar{u}$	$s\bar{u}$
8	$c\bar{c}$	$s\bar{c}$
9	$t\bar{t}$ (forbidden)	$s\bar{t}$ (forbidden)
10	$d\bar{d}$	$b\bar{u}$
11	$s\bar{s}$	$b\bar{c}$
12	$b\bar{b}$	$b\bar{t}$ (forbidden)

1.3 Q3: How to generate $e^+e^- \to \gamma/Z \to \mu^+\mu^-$ using Pythia with JSF

The process $e^+e^- \to \gamma/Z$ using Pythia are included in Pythia. To set parameters to run this process, do

```
  $ run jsf gui.C
% $
```

This will open JSF control panel. Then set parameters as follows;

```
[Controls] ->  [Generator Parameters...] ->
       [Pythia ...] -> [Event Type] -> gamma/Z
```

Decay mode of Z can be also restriced only to mu-pair as follows.

```
[Controls] -> [Generator Parameters...] ->
      [Pythia ...] -> [Event Type] -> Z decay mode
```

and specify 13.

1.4 Q4:How to get generator particle information in JSF?

In JSF, generator particle information is saved as JSFGeneratorParticle objects. PythiaGenerator module generate is from Phythia generator information. If Bases/Spring generator is used, it is created by JSFHadronizer module from SpringParton objects information.

PythiaGenerator module creates JSSFGeneratorParticle objects by essentially making a copy of LUJETS common of FORTAN program created by Pythia Generator. They are four momenta of particles, their particle ID, life time, etc at the interaction point.

Particle ID number is defined according to the Particle Data Group convention. It's definition will be found in the PDG book. Some particles decay immediately and create daughter particles. Only few particles such as gamma, pi+/-, p, n, ... have enough lifetime to be observed in the detector. Pythia generator not only generates initial particles at IP but also decays them in to stable particles if their life time are short. So there are two kinds of particle in JSFGeneratorObjects, those decayed or not decayed. If you use GetNDAU method, you will get number of daughter particles. If it is not 0, that means it decayed and has daughter particles. JSFGeneratorParticle class also has pointer to mother particle and first daughter particle so that user can trace decay chain. Pointer to the mother is negative when the particle information is obtained from the documentation line of LUJETS common. These information are used to get initial parton information (quark, lepton and gauge boson information before they are fragmented to hadrons.)

An example to scan all GeneratorParticle information and sum four momentum of stable particles is as follows.

```
// ----------------------
void UserAnalysis()
{
  ......
  // -----------------------------------------------------
  // Loop over GeneratorParticleList
  // -----------------------------------------------------
  JSFGenerator *gen=(JSFGenerator*)jsf->FindModule("JSFGenerator");
  JSFGeneratorBuf *genb=(JSFGeneratorBuf*)gen->EventBuf();
  TClonesArray *gps=genb->GetParticles();
  ANL4DVector psum;
  for(Int_t i=0;i<genb->GetNparticles();i++) {
    JSFGeneratorParticle *gp=(JSFGeneratorParticle*)gps->UncheckedAt(i);
    if( gp->GetNDaughter() > 0 ) continue;  // Select stable particles

    pg=ANL4DVector(gp->GetE(), gp->GetPx(), gp->GetPy(), gp->GetPz());
    psum += pg;
  }
  ...
}
```

Please see `JSFGeneratorParticle.h` for other functions defined by JSFGeneratorParticle class to get other information such as particle ID, etc.

1.5 Q5: Save seeds of random number to restart simulation later.

During course of our analysis, sometimes we want to restart Monte Calro event generation and simulation to study an event which apears only after generating many number of events. It is time consuming if we need to restart event generation from very beginning. In JSF, we can save seeds of random numbers of every events in a file and this file can be used to restart event simulation from middle of run.

The Quick Simulator uses CERNLIB subroutine, `ranmar` and `rnorml` to generate uniform and gausian random number generation. These routine requires to store about 200 words of array data to restore sequence of random number generation. To write these data in a file, a function `JSFLCFULL::WriteRandomSeed(char* filename)` can be used. If `filename` is ommitted, it is obtained from a jsf parameter `JSFLCFULL.RandomSeedWriteFile`. If the jsf parameter is not specified neither, the name `jsf-lcfull-seed.XXXX` is used, where `XXXX` is a Process ID (PID).

The seeds saved by `JSFLCFULL::WriteRandomSeed(char* filename)` are read by `JSFLCFULL::ReadRandomSeed(char* filename)`, which resets the state of the random number generation. Thus the random number generation can be restarted from the status stored in `filename`. The jsf parameter, `JSFLCFULL.RandomSeedReadFile`, can be used to set the file name.

The Tauola library used for τ decay also uses `ranmar` and `rnorml`. Pythia generator uses different subroutines to generate random numbers, which uses about 100 words of array data to save random number status. Thus `PythiaGenerator::WriteRandomSeed(char* filename)` and `PythiaGenerator::ReadRandomSeed(char* filename)` are prepared to write/read seeds of random number.

A sample script to be put in the standard `UserAnalysis.C` file is shown below.

```
//_________________________________________________________
void UserInitialize()
{
    JSFLCFULL *lcfull=(JSFLCFULL*)jsf->FindModule("JSFLCFULL");
    lcfull->ReadRandomSeed("seed-lcfull-previous");
    lcfull->PrintRandomSeed();
    PythiaGenerator *pyg=(PythiaGenerator*)jsf->FindModule("PythiaGenerator");
    pyg->ReadRandomSeed("seed-pythia-previous");
    pyg->PrintRandomSeed();
}
//_________________________________________________________
void UserAnalysis()
{
    JSFLCFULL *lcfull=(JSFLCFULL*)jsf->FindModule("JSFLCFULL");
    PythiaGenerator *pyg=(PythiaGenerator*)jsf->FindModule("PythiaGenerator");
    lcfull->WriteRandomSeed("seed-lcfull");
    pyg->WriteRandomSeed("seed-pythia");
}
```

In `UserAnalysis.C` script, a function `UserInitialize()` is called at the begining of job and `UserAnalysis()` is called at the end of event data analysis. In this example, seeds of other run stored in files `seed-lcfull-previous` and `seed-pythia-previous` are obtained at the begining of run, and seeds at the end of event analysis are stored in files, `seed-lcfull` and `seed-pythia`.

1.6 Q6: How to overlap background events with signal events ?

A class `JSFMergeEvent` can be used for such purpose.

Let's consider the case to generate 10000 two-photon events and overlap these events to the signal event, $e^+e^- \to \mu^+\mu^-$.

First we need to generate two-photon event data. A sample JSF Configuration file, `twogam.conf`, is as follows.

```
JSFGUI.Pythia.Process.ZH:0
JSFGUI.Pythia.Process.twophoton:1
JSFGUI.RunMode:1
JSFGUI.OutputFileName: twogam.root
JSFGUI.OutputEventData:1
```

You can set these parameters from JSF Control Panel or use editor to edit directory. We use Pythia to generate two-photon events. Then do,

```
jsf -b -q -conf=twogam.conf --maxevt=10000 gui.C
```

This will create a root file, `twogam.root`, containing 10000 events of two-photon process and cross section data.

Secondly, you generate the signal events, $e^+e^- \to \mu^+\mu^-$ overlaying two-photon background events. A sample JSF configuration file is,

```
JSFGUI.Pythia.Process.ZH:0
JSFGUI.MergeEvent:1
JSFGUI.Pythia.Decay.Z:13
JSFGUI.Pythia.Process.gammaZ:1
JSFMergeEvent.DataFile:twogam.root
JSFMergeEvent.LumPerTrain:0.06
```

where, `LumPerTrain` specifies the integrated luminosity per beam crossing in unit of 1/nb. The value of 0.06 corresponds to the instantaneous luminosity of $9 \times 10^{33}/cm^2/s$ operated at RF pulse frequency of 150Hz. There are about 100 bunches per pulse, but we assumed that we can not distinguish events from different bunch collisions happened in the same RF pulse. The average number of events in signal events is calculated as a product of `LumPerTrain` anda cross sections of background events. The number of background events in particular events is determined from Poisson distribution, and events are randomly selected from the background file, `twogam.root`, and their data are appended to the signal events.