November 4, 2002

`pandora` and `pandora-pythia`: Event Generation for Linear Collider Physics

Masako Iwasaki

Univ. of Tokyo, Dept. of Physics, Aihara Group
7-3-1 Hongo, Bunkyo-ku, Tokyo 113-0033 Japan

and

Michael E. Peskin[1]

Stanford Linear Accelerator Center
Stanford University, Stanford, California 94309 USA

ABSTRACT

The programs `pandora` and `pandora-pythia` provide a simple, unified framework for event generation for future linear colliders. Together, the programs provide event generation for the basic linear collider physics processes, including initial-state radiation and beam effects and full awareness of polarization. Parton showering and hadronization are included through interfaces to PYTHIA 6.1 and TAUOLA 2.6. Our package is written in `C++`. Its modular character allows new or modified physics to be incorporated in a straightforward manner.

[1]Work supported by the Department of Energy, contract DE–AC03–76SF00515.

Contents

1 Introduction

Event generators provide the essential connection between a theorist's image of a high-energy physics process and the experimenter's realization of this process in a detector. Event generators are needed in the planning stages of an experiment, to prove that the proposed technology can indeed see the physics, and they are needed as the experiment is run, to compare the ideal to reality in the measurement of signals and backgrounds.

Planned experiments at linear colliders pose special problems for event generation. In addition to the canonical problems of generating parton-level events and then making the transition to the hadron level, other features of the physics should be taken into account. The energy distributions of the electron and positron in the initial state of annihilation reactions at linear colliders is determined not only by initial state radiation but also by a process known as 'beamstrahlung', coherent hard synchrotron radiation emitted in the collision of the electron and positron bunches. Electron, and perhaps also positron, beams at linear colliders can be polarized, and the polarization-dependence is an extremely useful variable in experiments. Because detectors for e^+e^- colliders typically see all final jets and leptons, spin correlations in the final states can be measured, and these also contain valuable information.

For many of the processes of interest in linear collider physics, it is possible to obtain the full set of these effects by combining a number of available programs: CIRCE [1] for beam effects, PYTHIA [2] or HERWIG [3] for the core processes and hadronization, TAUOLA [4] to treat τ decays. But there is an advantage in combining these functions transparently in a single package. This is what we have tried to achieve with `pandora` and `pandora-pythia`. Using this package, it is possible to straightforwardly produce event samples for the basic processes of linear collider physics taking full account of beam and polarization effects. We use our own beam and parton-scattering simulation effects to produce parton-level events, passing the results to PYTHIA 6.1 and TAUOLA 2.6 to create the final hadrons.

Our programs are written in `C++` and have an intrinsic modular structure. Part of our aim is to make it easy to modify the parton-level physics or even to include new processes. At the same time, the underlying structure is aware of polarization and is built to include all parton-level quantum interference. What we sacrifice in return for this is accuracy. For the most part, we have realized the processes included in pandora only using tree-level Feynman diagrams, giving precision in cross sections only to about the 10and detector development. The final precision experiments will need a more exact, though then necessarily more bulky and cumbersome, realization.

This is the first of two papers that describe the current versions of `pandora` and `pandora-pythia`. In this paper, we describe the basic use of `pandora` as a parton-level event generator, the use of `pandora-pythia` to hadronize the events produced by `pandora`, and the status of the physics processes that are included. These descriptions

apply to `pandora` version 2.3 and `pandora-pythia` version 3.2. A companion paper [5] will describe the inner structure of the parton-level generator and give methods for modifying the current physics processes and for adding new ones.

This paper is organized as follows: In Section 2, we will define the basic ingredients of `pandora` and give the simplest example of a `pandora` simulation. In Section 3, we will describe the operation of `pandora-pythia` and illustrate this with a simple example. In Section 4, we will describe the beam physics incorporated into `pandora`. In Sections 5 and 6, we will provide more details on the operation of the `pandora` event generators for specific processes.

The `pandora` and `pandora-pythia` software is available for download at the Web site [6].

As a matter general convention, all energies in pandora are measured in GeV, all cross sections are returned in femtobarns (fb), and all angles (which are typically forward angular cutoffs) are measured in milliradians.

2 Parton-level simulations

2.1 Beams and processes

The cross section for a general physics process is given by a convolution of beam distributions with a collision cross section. A given beam leads to a probability distribution for partons entering the collision. At an e^+e^- collider, for example, each accelerated electron leads, after consideration of beam energy spread, beamstrahlung, and initial state radiation, to an ensemble of electrons, photons, and positrons distributed in longitudinal momentum. The distribution of each particle type is described by a function

$$\frac{d^n f^a}{dx^n} \tag{1}$$

depending on longitudinal fractions or some other integration variables x^n and on the resulting polarization state a. Similarly, the distribution of final partons is determined by a fully differential cross section

$$\frac{d^m \sigma^{ab}}{dy^m} \ , \tag{2}$$

where the y^m are scattering angles and other parameters of the final states, and ab are the initial polarizations. The complete expression for the differential cross section is then

$$\sigma = \int d^{n_1}x_1 \int d^{n_1}x_2 \int d^m y \ \frac{d^{n_1} f^a}{dx_1^{n_1}} \frac{d^{n_2} f^b}{dx_2^{n_2}} \frac{d^m \sigma^{ab}}{dy^m} \ . \tag{3}$$

An event generator will treat the integrand as a probability distribution and select points in the space of variables according to these probabilities.

The program `pandora` is a straightforward implementation of this idea in `C++`. We define `C++` classes `beam` and `process` which compute, respectively, the beam distributions and the differential cross section in (3). We define a class `pandora` which carries out multidimensional integrals such as (3). The `pandora` class returns both the value of the integral and also selects individual points with the correct probability.

Though the integrand of (3) is constructed in a modular way, `pandora` treats the integrand as a whole and attempts to optimize the selection of point over the entire function. In the current version, `pandora` uses the VEGAS algorithm [7, 8] to carry out Monte Carlo integration of (3), and then selects points using the resulting grid. The same method was used in the general-purpose event generator BASES and SPRING [9].

In a more sophisticated treatment, one might wish to take into account correlations between the two beams. In this case, (3) would be modified, with the two beam distributions replaced by a luminosity distribution

$$\frac{d^n L^{ab}}{dx^n} \tag{4}$$

To implement this idea, it is possible to create a `pandora` event generator from a `luminosity` class rather than from a pair of `beam` classes.

2.2 A first `pandora` program

With this introduction, we can look at a simple `pandora` program. The program in Fig. 1 generates 100 parton-level $e^+e^- \to t\bar{t}$ events and prints them out in a standard format called an `LEvent`, described in the following Section 2.3.

The code is fairly self-explanatory, but we will give more detail about the methods that are used. The constructor for the `pandora` class has the form

```
pandora(beam & B1, beam & B2, process & Pr);   .
```
(5)

This implements the idea expressed in the previous section. A `pandora` class has the methods

```
void prepare(int Ncalls);
LEvent getEvent();
LEvent getEvent(double & weight);
double integral(double & sd);
```
(6)

The first of these is an initialization step. The method `prepare` initializes the event generator. In the current version, this method initializes a grid for the evaluation of (3) by VEGAS. The method `getEvent()` returns a weight-1 event in the `LEvent` format.

```
#include "pandora.h"
#include "eetottbar.h"

int main(){

    double ECM = 500.0;
    double epolarization = -0.8;
    double ppolarization = 0.0;

    ebeam B1(ECM/2.0, epolarization, electron, electron);
    B1.setup(NLC500H);
    ebeam B2(ECM/2.0, ppolarization, positron, positron);
    B2.setup(NLC500H);

    eetottbar Rxn;

    pandora P(B1,B2,Rxn);
    P.prepare(100000);

    for (int i = 1; i <= 10; i++){
       LEvent E = P.getEvent();
       cout << E ;
    }

    return 0;
}
```

Figure 1: A simple `pandora` program. This program generates 10 parton-level $e^+e^- \to t\overline{t}$ events and writes them out to the terminal in a standard format. These events include effects of polarization, spin correlations, beamstrahlung, and initial state radiation.

The next method returns a weighted event, and returns the weight in the argument. The final method returns `pandora`'s current estimate of the value of the integral, with the statistical error returned as `sd`. Some additional methods of `pandora` are described in Appendix A.2.

The $e^+e^- \to t\bar{t}$ process is created in a very simple way: the class `eetottbar` is a subclass of `process` with a constructor that takes no arguments:

```
eetottbar();    .                                    (7)
```

More generally, the constructor of a process might take arguments which supply the values of internal masses, kinematic cutoffs, or nonstandard parameters. Methods for other process classes are described in Section 5 and in Appendix A.4.

The class `ebeam` is a subclass of `beam`. Its constructor specifies the nominal beam energy, beam polarization, and beam particle (`from`), and the beam particle initiating the parton-level reaction (`to`):

```
ebeam(double Ebeam, double polarization, int from, int to);   .  (8)
```

A positive sign of the polarization corresponds to right-handed circular polarization or positive helicity. In a reaction at an e^+e^- collider initiated by a Weiszacker-Williams photon, the variable `to` would be set to `photon` or to the synonomous Particle Data Group code, 22 [10]. The method

```
void setup(int design);                              (9)
```

supplies the accelerator parameters needed to compute the distributions from beam energy spread, beamstrahlung, and initial state radiation. A list of included designs, and some more general beam constructors, are given in Section 4. It is possible to turn off each of the three components of the beam distribution individually, using the `ebeam` class methods

```
void beamspreadoff();
void beamstrahlungoff();
void ISRoff(); .                                     (10)
```

By default, `pandora` produces events from head-on collision of beams. However, a finite crossing angle can be included by calling the method of the `pandora` class

```
void usecrossingangle(double theta);                 (11)
```

For example, if `P` is a pandora class, `P.usecrossingangle(20.0);` will introduce a 20 mrad crossing angle, with the beams positioned symmetrically about the $\hat{3}$ axis in the $\hat{3}\hat{1}$ plane.

It is hard to imagine that event generation could be made much simpler. Perhaps one might ask that the resulting events should be hadronized and returned in a more standard event structure. In Section 3, we will describe how we have implemented this step.

2.3 pandorasBox

A given `pandora` class generates events for a particular physics process. However, for some investigations it is useful to generate an event sample that includes a number of different physics processes, with events appearing in the ratio of the respective cross sections. It is possible to generate events in this way with `pandora` by creating a `pandorasBox` class.

The constructor of `pandorasBox` takes no arguments. Instantiating an element of the class produces an empty box. To build a `pandorasBox` event generator, create single-process `pandora` classes `P1`, `P2`, *etc.*, call `prepare` to initialize each and obtain an estimate of its integral (using (6)) and then add it to the box:

```
pandorasBox box;
box.add(P1)
box.add(P2)    ···                    (12)
```

The `pandorasBox` class has methods `getEvent()` and `getEvent(weight)` with the same structure as the corresponding methods in (6). Events are chosen according to the cross sections estimates of the various `pandora` generators current at the moment that they are added to the box.

2.4 The `LEvent` data structure

To complete the description of parton-level simulations, we will describe `pandora`'s natural event structure, the `LEvent`. An `LEvent` contains a list of 4-vectors for the final state partons and, possibly, intermediate state partons that appear in the simulation. Each parton also has associated with some discrete identifying information. There are methods to read and to set the data elements. It is also possible to send the whole data structure to `cout`, as is done in the program shown in Fig. 1.

Each final parton has assigned to it a values of the integer variables `ID`, `parent`, `final`, `chain`, and `level`. `ID` is the numerical label for the particle type assigned by the Particle Data Group [10]. If a parton has resulted from a decay, `parent` gives the number of the parent parton; otherwise, it is `0`. If a parton appears in the final state, `final` is set to `1`; otherwise, it is `0`.

The remaining two variables give information needed for the hadronization of the partonic final state. The variable `chain` gives the color connection of particles. The

value `-1` denotes the head or quark end of a color string. A positive integer value gives the next parton in the string, moving toward the quark end. This prescription treats color flow only at the leading order in $1/N_c$. Some specific negative values are also assigned. The value `-100` is assigned to the daughter of an unstable parton which retains the color of that parton. For example, when a top quark is produced, the chain variable for the t is `-1`, and the chain variable for the resulting b is `-100`. The values `-11`, `-12`, `-13`, and `-14` denote the τ lepton helicity states: respectively, τ_L^-, τ_R^-, τ_L^+, τ_R^+. The interface `pandora-pythia`, described in the next section, parses this information, supplies PYTHIA with the correct description of the color flow in the process, and sends τ lepton 4-vectors to TAUOLA to be decayed with the assigned longitudinal polarization. We should note that this prescription averages out transverse polarization correlations between τ leptons. This is normally a good approximation at high energies, but there are some processes at the linear collider where transverse polarization plays a role. The specific case of tests of the parity of the Higgs boson through $h \to \tau^+\tau^-$ [11] is discussed further in Section 5.5.

Finally, the variable `level` directs the hadronic showers that will be carried out by PYTHIA. PYTHIA's internal model carries out QCD showers sequentially, producing extra quarks and gluons, in a peaking approximation, along the line between a pair of partons [2]. The values of `level` gives the order in which these showers are to be carried out. Again, `pandora-pythia` parses this information and supplies it to PYTHIA in an appropriate form.

As an example, we show in Fig. 2 the values of the information variables in a typical $e^+e^- \to t\bar{t}$ event. Notice how, by carrying out the parton showers in the order indicated, starting with the $t\bar{t}$ production process, each set of collinear gluons required by QCD is correctly generated [12].

For boosting the parton 4-vectors from one frame to another, we have provided a number of methods. `pandora` provides an `LVector` class, a Lorentz vector indexed from 0 to 3, and an `LMatrix` class which can represent a matrix boost. The `LEvent` class has methods to boost its 4-vectors, and there is also a more primitive class `LVlist`, simply a list of 4-vectors, which has the same methods available. A general boost is carried out by

```
void generalboost(LMatrix & Lambda);    .
```
(13)

However, for almost all purposes, simpler operations are more convenient. For example, the operations

- `void boost(double beta);` : boost in the $\hat{3}$ direction.
- `void rotate(double phi);` : rotate about the $\hat{3}$ axis.
- `void rotateinplane(double cost);` : rotate in the $\hat{3}\hat{1}$ plane by θ such that `cost` $= \cos\theta$.

parton number	ID	parent	final	chain	level
1	6	0	0	-1	1
2	5	1	1	-100	2
3	24	1	0	0	2
4	2	3	1	-1	3
5	-1	3	1	4	3
6	-6	0	0	1	1
7	-5	6	1	-100	4
8	-24	6	0	0	4
9	15	8	1	-11	0
10	-16	8	1	0	0

Figure 2: Example of information variables attached to the partons in an LEvent representing the final state of $e^+e^- \to t\overline{t}$. The t decays to $bW^+ \to bu\overline{d}$. The $\overline{t}$ decays to $\overline{b}W^- \to \overline{b}\tau^-\overline{\nu}$

allow one to take a distribution of decay products in the parent's rest frame and bring it to the lab frame in an orientation specified by Euler angles. The complete list of these boost and rotation methods is given in Appendix A.2.

2.5 Example programs for `pandora`

In the `pandora` distributions, the directories `SMexamples`, `SM3examples`, and `BSMexamples` contain example programs. Programs with the title `AAevents.cpp` produce a fixed number of events and print these to `cout`. Programs with the title `AAexample.cpp` produce some histograms of parton-level quantities in the form of `.gp` files. These files can be viewed with the graphics program `gnuplot`. In either case, to compile the program, simply type

`make AAevents` or `make AAexample` (14)

from within the directory containing the program.

3 Hadron-level simulations

3.1 A first pandora-pythia program

To turn the partonic events produced by pandora into hadronized events, we attach pandora to PYTHIA as an external subprocess. For a default set of parameter choices, the interface pandora-pythia takes care of all of the details. An elementary pandora-pythia program is shown in Fig. 3. The hadronized events—including QCD showers and polarized τ decays as described in the previous section—are written to an external file. Using one of the possible settings of PYTHIA, the events are written in the StdHep format [13]. The resulting file can them be used as input to detector simulations or other physics analysis programs.

3.2 Operation of pandora-pythia

As shown in the Fig. 3, the main part of a pandora-pythia program is simply a pandora program. The user sets up beam and process classes as defined in the previous section and instantiates the pandora class. Then this class can be used to built a pandora-pythia generator, by the command

```
pandorarun PR(P,nEvent,iseed_pan);   .                (15)
```

In this statement, P is the pandora class and nEvent is the number of events to be generated. The variable iseed_pan is an integer label for the seed to be used for the random number generator in pandora; any small positive integer can be supplied.

The pandorarun constructor initializes PYTHIA, choosing the default setup, setting the seed value for PYTHIA to be 19518503, turning on fragmentation, parton showering, and final-state radiation. To change these default choices, pandorarun has the methods

```
void newseed(iseed_new);
void fragmentationoff();
void FSRoff();
void partonshoweroff();    .                          (16)
```

By default, we have the complete event list for the beginning 5 events. Using the method

```
void SetPrintEvent( iEvent );  ,                      (17)
```

in pandorarun class, the number of events to be listed can be changed.

In the Fig. 3, pandora-pythia main processes, initialization, event generation, and termination of pandora-pythia are operated by the following 3 lines:

```
PR.initialize(outfile);
```

```
#include "pandora.h"
#include "ebeams.h"
#include "eetottbar.h"
#include "pandorarun.h"

int main(int argc, char * argv[]){

    int   nEvent    = atoi(argv[1]);   // # of events to be generated
    char* outfile   = argv[2];         // out put file name
    int   iseed_pan = atoi(argv[3]);   // seed number for pandora

    /*  begin pandora code here  */
    double ECM = 500.0;             // Center of Mass energy
    double epolarization = -0.8;    // Electron poralization
    double ppolarization =  0.0;    // Positron poralization

    ebeam B1(ECM/2.0, epolarization, electron, electron);
    B1.setup(NLC500H);
    ebeam B2(ECM/2.0, ppolarization, positron, positron);
    B2.setup(NLC500H);

    eetottbar Rxn;
    pandora P(B1,B2,Rxn);
    /*  end pandora code  */

    /*  begin pandora-pythia code  */
    pandorarun PR(P,nEvent,iseed_pan);
    PR.initialize(outfile);
    PR.getevents();
    PR.terminate();
    /*  end pandora-pythia code  */

    return 0;
}
```

Figure 3: A simple `pandora-pythia` program. The compiled program takes as arguments the number of events to be generated, the name of the file into which they should be written, and an initial seed number for pandora.

```
PR.getevents();
PR.terminate();  .                                    (18)
```

In the initialization step, we call PYTHIA and TAUOLA initialization. If we don't provide the out put file name in this step, there is no out put file.

By the function `getevents()`, we generate the events by `pandora` then go through PYTHIA for parton showering and for fragmentation, call TAUOLA for tau-decay, store the particle 4-vector information into the HEPEVT common, and write the event record to the output xdr binary file (if provides the out put file name). The number of event generation in `getevent()` is inputted as `nEvent` in the `pandorarun` constructor. Instead of this, there is another class `getevent()` for a single event generation. In the termination step, we call `pandora` PYTHIA and TAUOLA termination step.

Some detector simulators decay long lived particles during the detector simulation. If you use such a simulator, you must not allow the decay of such particles, K_s^0, Λ, and so on, in the generator level. In order to turn off the particle decay, there is a function in `pandorarun` class, `void PartDecayOff(int PYTHIA_particle_code)`. We should call this function after calling `initialize(outfile)`.

3.3 pandorasBoxrun

As expressed in Section2.3, there is `pandorasBox` class in `pandora` to generate a number of different physics processes, with events appearing in the ratio of the respective cross sections at once. In `pandora-pythia` program, we also have the corresponding class, `pandorasBoxrun` to take care of the `pandorasBox`.

To use `pandorasBoxrun`, just replace `pandorarun` to `pandorasBoxrun` in the Fig. 3. Here you can create on or more `pandora` classes `P1`, `P2`, *etc.*, and then add it as:

```
pandorasBoxrun  PB(nEvent, iseed_pan);
PB.addPandora(P1);
PB.addPandora(P2);    ···                             (19)
```

Almost all functions in `pandorasBoxrun` are common as in `pandorarun`: for example, the `pandorasBoxrun` class has methods `initialize(outfile)`, `getevents()`, and `terminate()` with the same structure as the corresponding methods in (18).

4 Simulation of beams

4.1 Electron beams: bremsstrahlung

For an electron beam, the beam distribution functions f^a in (1) are convolutions of three distributions which take account of the three factors which produce a spread

of energies: First, the electron beam comes out of the accelerator with a small but nonzero energy spread. Second, the macroscopic electric and magnetic fields created when particle bunches collide produces synchrotron radiation, called beamstrahlung, taking energy from the colliding particles. Finally, photons are emitted by bremsstrahlung in the hard-scattering process itself.

Our approach to the beam modelling is to represent each of these effects by a simple and robust approximation that captures the essential physics. This strategy allows one, for example, to study the effect of the various components of the beam distribution separately, and to study the effect on physical observables of changes in the accelerator design parameters.

We should note that the beam parametrizations that we will now describe are not the only ones that can be used with `pandora`. In principle, any method for generating the distribution functions can be coded to implement the `beam` or `luminosity` class interface. In this form, the beam distribution will fit together neatly with the other components of `pandora`. The `beam` and `luminosity` classes are described further in [5].

We now describe the makeup of the `ebeam` class, appropriate for describing e^+e^- collisions. The treatment of beam energy spread is especially simple. Beam energy spread is often modelled as a Gaussian distribution, but this is not in fact appropriate. The actual distributions that emerge from accelerator simulations have a defined energy interval with a sharp cut-off at the boundary and peaks at the boundary that give the distribution a horned shape. In the `ebeam` class of `pandora`, we model this shape as a flat distribution over a finite interval. For a given accelerator design, we match the rms width, so that the total width of the energy spread in `pandora` is $\sqrt{12}$ times the rms deviation.

The treatment of bremsstrahlung in high-energy e^+e^- collisions is the subject of a substantial literature. The probability of emitting a photon collinear with the e^+e^- collision axis is much greater than α, since it is enhanced by a singularity in the Feynman integral. Fadin and Kuraev suggested treating the energy distribution of an electron which radiates a large number of collinear photons as a parton distribution $f_e(x)$ analogous to those in QCD [14]. They solved for this distribution function as an expansion in

$$\beta = \frac{2\alpha}{\pi}(2\log(2.0E/m_e) - 1) \ , \tag{20}$$

where E is the beam energy and m_e is the mass of the electron. Subsequent efforts, in particular, by Nicrosini and Trentadue [15], improved this approximation to the point where it could achieve the 0.1% level of accuracy needed to predict the Z^0 line shape. In `pandora`, we use a less elaborate approximate formula due to Skrzypek and Jadach [16],

$$f_e(x) = \frac{1}{2}\eta(1-x)^{\eta/2-1}$$

$$\cdot\ (1+\frac{1}{2}\eta)e^{-(\eta+(\pi^2/6-1)\eta^2)/8}\left[\frac{1}{2}(1+x^2)-\frac{\eta}{8}(\frac{1}{2}(1+3x^2)\log x+(1-x)^2)\right] \quad (21)$$

where

$$\eta = -6\log(1-\frac{1}{6}\beta)\ . \quad (22)$$

This approximation is good to about 1% accuracy. The first line of (21) is a simple normalized function that already gives a quite reasonable accounting of the distribution.

The derivation of the distribution function (21) assumes that an arbitrary number of photons with energies up to the beam energy are radiated in the forward direction. For some purposes, it is important to generate these photons explicitly as particles with finite transverse momentum that might appear in a detector. For example, one might wish to simulate the process $e^+e^- \to Z^0\gamma$, with an explicit photon appearing at a nonzero angle. It is double-counting to include both this process and the distribution function with the value of β given above. `pandora` repairs this problem in a relatively crude way by providing a method of the electron beam class

```
void maxpt(double pt);
```  (23)

For a beam `B1`, calling `B1.maxpt(10.0);` will replace the beam energy in (20) by the value 10 GeV. Photons with transverse momentum above 10 GeV can then be generated explicitly without substantial double-counting.

The photons radiated from the electron beam must also be accounted, since they can also initiate hard reactions. For example, two-photon reactions arise from the collision of bremsstrahlung photons from the two beams. Unfortunately, the single-photon distributions corresponding to the Fadin-Kuraev formula and its various improvements have not been worked out. In `pandora`, we model these distributions with approximate solutions of the Fadin-Kuraev equations. It is important to take polarization into account. If a left-handed electron comes out of the accelerator, it remains a left-handed electron at the level of the approximation (21), but it radiates both left- and right-handed photons. For an initial left-handed electron, `pandora` takes

$$f_{\gamma L}(x) = \frac{(1-(1-x)^{\eta/2})}{-2x\log(1-x)}$$
$$f_{\gamma R}(x) = \frac{(1-(1-x)^{\eta/2})(1-x)^2}{-2x\log(1-x)}\ . \quad (24)$$

For an initial right-handed electron, these functions are reversed. The beam photons created with these distributions are assumed to have zero transverse momentum. If one chooses to explicitly generate photons of finite transverse momentum, the method `maxpt` can be used as above to suppress the corresponding part of the beam distribution.

4.2 Electron beams: beamstrahlung

While the modelling of bremsstrahlung has a firm basis in physics, it is much less certain how to model beamstrahlung. Photon distributions from beamstrahlung can be computed directly in the limit of small disruption, that is, small deflection of the electrons in the scattering process [17]. However, realistic collider designs have relatively large disruption. In this case, it is necessary to follow the electron trajectories through the collision by explicit simulation. Programs such as CAIN [23] and GUINEA-PIG [24] carry out the many-particle dynamics simulation of the interaction of electron bunches and predict the final electron and photon distributions.

A physics event generator should then include distributions which arise from a fit to this simulation data. Two approaches have been used in the literature. CIRCE 1.0 attempts to model the output distributions of GUINEA-PIG as accurately as possible, using a piecewise approximation by simple functions [1]. Release 2.0 of CIRCE also includes correlations between the two beams [18].

As an alternative, one might attempt to make a physically motivated parametrization of the results of the simulation programs and use this to produce general beamstrahlung distributions which are functions of accelerator parameters. Yokoya and Chen have constructed a number of formulae which allow this interpolation [19, 20, 21]. This approach allows one to systematically vary accelerator parameters and study the effect on beam distributions and on the physics observables that they produce.

In our opinion, it is very unlikely that beam simulations like GUINEA-PIG will give a precision accounting of the actual electron and photon distributions that will be found the electron bunch collisions at a linear collider. For accurate work, these distributions will need to be measured during the collider operation using acollinear Bhabha and $Z^0\gamma$ events. For this reason, we have made the Yokoya-Chen formulae the default in `pandora`. However, we also provide the CIRCE 1.0 distributions for a number of current machine designs. We also provide a facility for user input of beamstrahlung parametrizations.

In the remainder of this section, we give the formulae for the Yokoya-Chen approach. First, they must construct formulae for the effective synchrotron radiation parameters of beamstrahlung as a function of the accelerator parameters. Then, they generate the distribution functions from these parameters using an exactly solvable approximate integral equation for the bunch evolution in the collision process. In `pandora`, we adopt the formulae of the first type but use a modified form of the Yokoya-Chen integral equation [22].

For a given accelerator design, we first compute the parameters N_γ and Υ, the average number of synchrotron radiation photons emitted and the parameter which distinguishes the classical and quantum regimes [20]. For electron and positron bunches with $\mathcal{N}$ particles per bunch and bunch sizes σ_i, $i = x, y, z$, in the approximation of flat beams, $\sigma_x \gg \sigma_y$, The horizontal and vertical disruption parameters D_x and D_y

are given by

$$D_a = \frac{2\mathcal{N} r_e \sigma_z}{\gamma \sigma_a (\sigma_x + \sigma_y)} \ , \tag{25}$$

where $a = x, y$, r_e is the classical electron radius, and $\gamma = E/m_e$. A fit to simulation data gives the corresponding luminosity enhancement factors

$$H_a = 1 + D_a^{1/4} \frac{D_a^3}{1 + D_a^3} \left[\log(1 + \sqrt{D_a}) + 2 \log(0.8 \beta_a / \sigma_z) \right] \ , \tag{26}$$

where $\beta)a$ are the horizonal and verticle β functions at the interaction point. Let (sic)

$$\overline{\sigma}_x = \sigma_x H_x^{-1/2} \ , \qquad \overline{\sigma}_y = \sigma_y H_x^{-1/3} \ . \tag{27}$$

Then the collider integrated luminosity per bunch crossing is given by

$$\mathcal{L} = \frac{\mathcal{N}^2}{4\pi \overline{\sigma}_x \overline{\sigma}_y} \tag{28}$$

and the effective value of Υ is given by

$$\Upsilon = \frac{5}{6} \frac{r_e^2 \gamma \mathcal{N}}{\alpha \sigma_z (\overline{\sigma}_x + \overline{\sigma}_y)} \ . \tag{29}$$

The classical and quantum radiation rates are given by

$$\nu_{\rm cl} = \frac{5}{2\sqrt{3}} \frac{\alpha^2}{r_e \gamma} \Upsilon \ , \qquad \nu_\gamma = \nu_{\rm cl} (1 + \Upsilon^{2/3})^{-1/2} \ . \tag{30}$$

Finally,

$$N_{\rm cl} = \sqrt{3} \sigma_z \nu_{\rm cl} \ , \qquad N_\gamma = \sqrt{3} \sigma_z \nu_\gamma \ . \tag{31}$$

To produce a distribution function from these parameters, we follow the approach of Yokoya and Chen in writing down an approximate evolution equation for the electron bunch which can then be solved exactly. We write

$$\frac{\partial f_e}{\partial t} = -\nu_\gamma f_e(x,t) + \nu_\gamma \int_x^1 dx' \frac{\kappa}{x} g(\eta - \eta') f_e(x',t) \ , \tag{32}$$

where $\kappa = 2/3\Upsilon$, $\eta = \kappa/x$, and $g(\delta\eta)$ is a translation-invariant kernel suggested by Yokoya and Chen [19]. The solution of this equation is

$$f_e(x) = e^{-N} \left[\delta(x-1) + \frac{e^{-\kappa(1-x)/x}}{x(1-x)} h(y) \right] \ , \tag{33}$$

where $N = \nu_\gamma t$, $y = N(\kappa(1-x)/x)^{1/3}$, and

$$h(y) \ = \ \sum_{n=1}^{\infty} \frac{y^n}{n! \Gamma(n/3)} \ = \ \left(\frac{3z}{8\pi} \right)^{1/2} e^{4z} \left[1 - \frac{35}{288} \frac{1}{z} - \frac{1295}{16588} \frac{1}{z^2} - \cdots \right] \ , \tag{34}$$

with $z = (y/3)^{3/4}$. This expression exactly satisfies the constraint that $f_e(x)$ should have integral 1. To represent the electron spectrum at a typical collision within the bunch interaction, we set $N = N_\gamma/2$.

Ideally, the photon distribution in beamstrahlung should be obtained from an integral equation similar to (32), constructed in such a way that the final electrons and photons sum to the fixed initial longitudinal momentum. To obtain an analytic expression for the photon distribution, we make the further approximation of replacing the electron distribution by a delta function at $x = 1$. This gives

$$f_\gamma(x) = N \frac{\kappa^{1/3}}{\Gamma(1/3) x^{2/3} (1-x)^{4/3}} e^{-\kappa x/(1-x)} \ . \tag{35}$$

The beamstrahlung photons are taken to be unpolarized. The beamstrahlung electrons are taken to retain their original polarization. This ignores a small (1%) depolarization in the bunch-bunch collision.

4.3 Yokoya-Chen electron beams in `pandora`

Now that we have discussed the physics input to Yokoya-Chen beams in `pandora`, we will explain how a user can manipulate `pandora`'s description of beam effects.

In Section 2.2, we have presented the `ebeam` constructor (8), which depends on the beam energy and polarization and on the nominal and final partons. The information on the accelerator parameters is input through the `setup` function, of which an example is given in (9). The `setup` function actually comes in a number of forms:

```
void setup(int design);
void setup(int design, double & luminosity);
void setup(int design, double Nfraction, double & luminosity);
void setup(int Upsilon, double Ngamma, double spread);
void setup(double N, double sigmax, double sigmay, double sigmaz,
    double betax, double betay, double spread);
void setup(double N, double sigmax, double sigmay, double sigmaz,
    double betax, double betay, double spread, double f,
    double & luminosity);
```
(36)

The parameter `design` can take a number of preassigned values, including

```
NLC350, NLC500A, NLC500B, NLC500C, NLC500H, NLC1000A, NLC1000B,
      NLC1000C, NLC1000H, NLC1500
JLC350, JLC500A, JLC500B, JLC500C, JLC500H, JLC1000A, JLC1000B,
```

```
      JLC1000C, JLC1000H
NLC250TR, NLC500TR, NLC800TR, NLC1000TR, NLC1200TR, NLC1500TR,
       NLC500TR2, NLC1000TR2, NLC1500TR2
TESLA500, TESLA800
CLIC500, CLIC1000, CLIC3000, CLIC5000                    (37)
```

In most cases, thse parameter sets correspond to our best understanding of the current official designs of the collaborations. However, the designs labeled `TR` represented some additional parameter sets created by Raubenheimer [25] to test the dependence of physics observables on collider energy. The designs `TR2` are alternatives that are more aggressive in raising the luminosity at the expense of beamstrahlung.

When the second `setup` function in (36) is used, the luminosity (in $\mathrm{cm}^{-2}\mathrm{sec}^{-1}$) is returned in the last variable. The third `setup` function multiplies the nominal number of particles per bunch $\mathcal{N}$ by `Nfraction`; this allows one to study the tradeoff between luminosity and beamstrahlung energy spread. The fourth constructor allows direct specification of the beamstrahlung distributions from Υ and N_γ. This might be useful at high energies where the estimation of these values from the accelerator parameters fails [26]. The last two constructors allow input of arbitrary accelerator design parameters. The parameter `f` is the number of bunch crossings per second.

One should note that specifying the accelerator parameters with the `setup` function does not set the beam energy. This is always taken to be the energy specified in the original beam construction. This energy determines the value of the parameter γ used, for example, in (29). Thus, to simulate e^+e^- collisions at 400 GeV, one might fix the beam energy at 200 GeV and input the `NLC500H` parameter set. This will give a beamstrahlung distribution that will be realistic for 400 GeV, though it might correspond to a somewhat non-optimal set of accelerator parameters. A realistic value of the luminosity, corresponding to the 400 GeV energy choice, will also be returned.

4.4 CIRCE and user-defined electron beams

In CIRCE 1.0 [1], Ohl fit simulation data to obtain the best parametrization of electron and photon beams of the form

$$\begin{aligned} f_e(x) &= a_0\delta(x-1) + a_1 x^{a_2}(1-x)^{a3} \\ f_\gamma(x) &= a_4 x^{a_5}(1-x)^{a6} \end{aligned} \tag{38}$$

The CIRCE code contains an elaborate table of the parameters $a_0 \ldots a_6$ for a variety of machine designs. The `pandora` class `CIRCEebeam` provides a beamstrahlung distribution of this form either with one of Ohl's parameter sets or with a general set of user-specified parameters.

As with the `ebeam`, the machine design is specified in the `setup` class function. A `CIRCEebeam` has three choices for this function:

```
void setup(int design);
void setup(int design, double & luminosity);
void setup(double a0, double a1, double a2, double a3, double a4,
    double a5, double a6, double a7, double beamspread);
```
(39)

In the last function, the parameter a_7 should be the integral over the photon distribution. (The integral over the continuum part of the electron distribution is $(1-a0)$.) The possible choices for `design` are

```
NLC500B, NLC500H, NLC800B, NLC800H, NLC1000B, NLC100H
JLC500B, JLC500H, JLC800B, JLC800H, JLC1000B, JLC100H
NLC250TR, NLC500TR, NLC800TR, NLC1000TR, NLC1200TR, NLC1500TR,
       NLC500TR2, NLC1000TR2, NLC1500TR2
TESLA90, TESLA170, TESLA350, TESLA500, TESLA800
```
(40)

Most of these choices correspond to accelerator parameters in (37). The NLC/JLC beams at 800 GeV are obtained by interpolation between the 500 GeV and 1000 GeV values. Note also that this parameter list includes a number of additional energy points for TESLA.

The `CIRCEbeam` class also includes the functions

```
double electronIntegral(); \qquad double photonIntegral();
```
(41)

which return the integrals under the electron and photon continuum distributions.

For the analysis of experimental determinations of the beamstrahlung spectrum, it might be useful for users to specify their own functional forms of the beamstrahlung parameterization [27]. To allow this, `pandora` includes a class `userebeam`. The constructor for this class has the standard form

```
userebeam(double  EBeam, double Polarization, int From,  int To);
```
(42)

However, instead of a `setup` function, this class contains two virtual functions

```
virtual double myF(double x);     virtual double myG(double x);
```
(43)

To create a beam whose beamstrahlung distributions satisfy

$$\begin{aligned} f_e(x) &= A\delta(x-1) + F(x) \\ f_\gamma(x) &= G(x) \ , \end{aligned} \tag{44}$$

with

$$A + \int_0^1 dx F(x) = 1 \ , \tag{45}$$

one should form a subclass of this class that defines the two virtual functions to be $F(x)$ and $G(x)$, respectively. The class should also set the beam spread variable `spread` to the desired full with of the accelerator energy distribution ($\sqrt{12}$ times the rms). The functions (10) that allow one to turn the various beam features on and off should work transparently. As an illustration, the class `CIRCEebeam` is actually defined in `pandora` as a subclass of `userebeam`.

4.5 e^-e^- beams

The description of beamstrahlung in terms of effective parameters N_γ and Υ can be extended to e^-e^- collisions. For e^-e^-, the dependence of the beamstrahlung parameters on the underlying accelerator parameters is different from e^+e^-, because the colliding bunches repel rather than attract. For the same reason, the intrinsic luminosity of e^-e^- designs is lower, and there is more incentive to compromise the beam characteristics to achieve more luminosity. The `eminusbeam` class gives a simple realization of these features, following the work of Chen and Thompson [28, 29].

The basic structure of the `eminusbeam` class follows that of the `ebeam` class. We begin again from the formula (25). For e^-e^- and the case of flat beams, $\sigma_x \gg \sigma_y$, Chen and Thompson [28] have offered the following parametrization of the simulation data: Let

$$f_R = 1 + \frac{\sigma_y^2}{\sigma_x^2} \ , \qquad f_{ch}(D) = \begin{cases} 1 & D < 1 \\ 1 + 0.1 \log D & D > 1 \end{cases} \ . \tag{46}$$

Then the luminosity enhancement or decrement H is given by

$$H = \left[1 - 0.15 f_R \left(\frac{\sigma_z/\beta_y}{1 + 0.4 D_y} \right)^{f_R} \right] \exp\left(-\frac{f_r D_y}{3\sqrt{\pi}} \right) I_0\left(\frac{f_r D_y}{3\sqrt{\pi}} \right) f_{ch}(D_y) \ , \tag{47}$$

where $I_0(z)$ is the Bessel function. Then, after writing (sic)

$$\overline{\sigma}_x = \sigma_x H^{-1/2} \ , \qquad \overline{\sigma}_y = \sigma_y H^{-1/2} \ , \tag{48}$$

we can continue from (28) with the same formulae as in the e^+e^- case.

Thompson has suggested one further change to find a better compromise between luminosity and energy spread: for a given set of e^+e^- machine parameters, multiply $\mathcal{N}$ by $\sqrt{2}$ and divide σ_z by $\sqrt{2}$. The `setup` function for the `eminusbeam` class in `pandora` accepts the standard parameter set of e^+e^- colliders but makes this change before computing the beamstrahlung parameters and the luminosity. To omit this change, one can call

```
void simplesetup(int design);
void simplesetup(int design, double & luminosity);  .
```
(49)

4.6 Backscattered photon beams

The availability at a linear collider of small intense high-energy electron beams makes it possible to imagine creating intense photon beams by Compton backscattering of laser light [30]. `pandora` includes a `pbeam` class which contains the luminosity functions for an idealized photon collider.

The formulae for the photon and electron spectra in a Compton backscattered photon beam are given in [30]. For an electron beam of energy E and a photon beam of energy ω, let

$$x = \frac{4E\omega}{m_e^2} \ . \tag{50}$$

Let z be the longitudinal fraction of the final photon. For simplicity, ignore the angular spread of the Compton-backscattered beam. Then there are sixteen polarized beam distribution functions. In the notation in which $f(-+-+)$ represents the distribution function for $e^-(L)\gamma(R) \to e^-(L)\gamma(R)$,

$$\begin{aligned}
f(----) = f(++++) &= N^{-1} \cdot (x - z(1+x))(x-z)^2/(1-z)^2 \\
f(---+) = f(+++-) &= N^{-1} \cdot (x - z(1+x))z^2/(1-z)^2 \\
f(--+-) = f(++-+) &= N^{-1} \cdot (x - z(1+x))^2 z/(1-z)^2 \\
f(--++) = f(++--) &= N^{-1} \cdot z^3/(1-z)^2 \\
f(-+--) = f(+-++) &= N^{-1} \cdot (x - z(1+x))z^2/(1-z)^2 \\
f(-+-+) = f(+-+-) &= N^{-1} \cdot (x - z(1+x))^3/(1-z)^2 \\
f(-++-) = f(+--+) &= N^{-1} \cdot (1+x)^2 z^3/(1-z)^2 \\
f(-+++) = f(+---) &= N^{-1} \cdot (x - z(1+x))^2 z/(1-z)^2 \ ,
\end{aligned} \tag{51}$$

where N is chosen so that, if $P_i = (1 \pm p)/2$ for $i = +-$ is the probability of a particle with the given helicity in a beam of polarization p,

$$\int_0^1 dz \sum_{ijkl} P_i P_j f(ijkl) = 1 \ . \tag{52}$$

The photon distributions are only defined up to a maximum value of z equal to $x/(1+x)$. `pandora`uses these formulae to compute polarized distributions for both photons and electrons for laser backscattering on an initial monochromatic electron beam. The value of x can be input, but by default it is chosen to take Telnov's suggested value $x = \texttt{xTelnov} = 2+\sqrt{8}$ [31]. `pandora` also adds initial state radiation, using the prescription of Section 4.1, for the scattered electron. This decreases the energy of the electron component and adds soft photons.

The backscattered Compton beam is defined in `pandora` by a constructor

```
pbeam(double Ebeam, double epolarization, double lpolarization,
      int from, int to, double x = xTelnov);   .          (53)
```

Features of the backscattered Compton beam can be turned off using the methods

```
void ISRoff(); .
void Comptonoff();                                        (54)
```

The first of these methods turns off initial state radiation from the electron component of the beam. The second gives a photon beam which is a delta function in energy at the Compton endpoint $z = x/(1+x)$, with the polarization set for the initial electron beam. `pandora` also includes a class that produces a pure photon beam with the stated beam energy and polarization:

```
simplepbeam(double Ebeam, double polarization);           (55)
```

Because of correlations between the two beams induced by Compton scattering at finite angle and other realistic aspects of the Compton process, the actual single-beam distributions that will be found at a photon collider differ significantly from the ideal shapes just presented. Gronberg has produced an more realistic beam description by binning the distributions produced by CAIN simulation of the beams produced from the LLNL photon collider design [32].

5 Simulation of particle reactions

5.1 Structure of a process class

We now turn to the description of scattering process in `pandora`. Our discussion will concentrate on the parton level; we have already discussed in Section 3 how parton-level events are translated to the hadron level.

At its present stage, `pandora` simulates e^+e^- reactions according to the tree-level Feynman diagrams. We set $\alpha = 1/128$, appropriate to TeV energies, and $\sin^2\theta_w = 0.231$. These variables and other electroweak and QCD parameters are defined globally in the file `hepdata.h`. Pandora considers all quarks and leptons except for the t quark to have zero mass. In `pandora-pythia`'s conversion to from the `LEvent` format to PYTHIA data, a light quark or lepton with mass m in PYTHIA is given energy $(m^2 + |\vec{p}|^2)^{1/2}$. This prescription conserves momentum, and the small shift in total energy is typically much smaller than the beamstrahlung and radiation energy losses.

Each scattering process in `pandora` has an associated `process` class. The name of the process `Pr`, *e.g.*, `e+e- -> q qbar`, is returned from `Pr.name`. Each process also knows the number of integration variables it requires to fully represent its final state, and the number of initial and final helicity states it is representing. The internal structure of a `process` class should be transparent to a user who is only making use of the pre-coded physics reactions. In this section, we will give only a few details of that structure, to indicate its basic form. Those who would like to write their own `pandora process` classes should consult the much more detailed description given in [5].

A `process` is required to implement a number of methods used by `pandora`, of which the most important are

```
int computeKinematics(double & J, DVector & X, double s, double beta);
void crosssection();
LEvent buildEvent();                                                    (56)
```

The purposes of the last two methods are evident: `crosssection` computes the polarized, fully differential cross section at a fixed kinematic point and stores this information in an internal matrix called `cs` for use by the `pandora` integrator. When a specific kinematic point is chosen by the event generator, `buildEvent` constructs the parton-level event in the center of mass frame and returns it as an `LEvent`.

The `computeKinematics` function defines the kinematic point at which the cross section is evaluated. It provides the interface between the integrator, which works with general positive function on the domain $[0,1]^N$, and the physics description. This function depends on the squared center of mass energy s, the boost of the center of mass β, and a vector `X` of integration variables $X_i \in [0,1]$. The function determines whether the choice of the X_i corresponds to a kinematically sensible point (*e.g.*, above threshold or inside the Dalitz plot contour). If the point is allowed, the function goes on to compute the relevant kinematic variables (production and decay angles and masses of particles with nonzero width) and store them in the class data members. The variable `J` returns the Jacobian of the transformation from the variables X_i to the more usual kinematic variables used to define the differential cross section. Finally, `computeKinematics` can make kinematic cuts on these variables. After successfully completing all of these steps, the function returns `1`. At any intermediate step, if any required test fails, `computeKinematics` can return `0`, bypassing all subsequent steps of the function. This causes the point X in the integration domain to be assigned weight 0.0.

5.2 Generic to specific

In using a given `process`, one might wish to alter it in some basic ways. If the process is generic, one might wish to make it more specific. For example, a generic simulation

of the reaction $e^+e^- \to t\overline{t}$ includes all final states of the W decays. One might wish to restrict this to events in which the W's decay only to leptons, or in which for example, the W^+ decays only to $c\overline{s}$.

For this purpose, a `pandora process` class with unstable particles in the final state has a method `onlyDecay`. For each unstable particle, this method takes an integer argument which specifies the decay modes allowed. For the case of $e^+e^- \to t\overline{t}$, the method takes the form

```
void onlyDecay(int for1, int for2);
```
, (57)

where `for1` is a code for the decays of the t or W^+ and `for2` is a code for the decays of the $\overline{t}$ or W^-. These variables take values from a list of constants defined in `hepdata.h` which includes such entries as `allStates`, `leptonsOnly`, and `cOnly`. The complete list of these variables in `pandora` 2.3 is given in Appendix A.

Though it should now be clear that the `pandora` process corresponding to $e^+e^- \to W^+W^-Z^0$ should have a method `onlyDecay(for1,for2,for3)`, it is not obvious which particle is controlled by which argument of `onlyDecay`. This should be documented in the `.h` file that defines the specific `process` class. It is also straightforward to determine the assignments by trial and error.

5.3 Restricting phase space

Reactions with massless particles in the final state typically have differential cross sections that are singular at some boundaries of the phase space. In order that the integral (3) be finite, it is necessary to place some kinematic cuts on the final state variables. We have explained above how these cuts are implemented in `pandora`. But we must still explain how the locations of the cuts are chosen by the user.

Processes with strong peaking or genuine divergences at points on the boundary of phase space will have `pandora process` classes with multiple constructors. For example, the `process` for Bhabha scattering, $e^+e^- \to e^+e^-$, has three constructors

```
eetoee();
eetoee(double thetamin, double ptmin, double ECMmin);
eetoee(double thetamin, double thetamax, double ptmin,
           double ptmax, double ECMmin, double ECMmax);
```
(58)

The second constructor specifies minimum values of the e^+e^- center of mass energy, the electron or positron transverse momentum, and the minimum of the electron or positron angle from the beam direction in the lab frame. The first constructor uses the same set of kinematic cuts, with the default values $\theta_{\rm min} = 10$ mrad, $p_{T\rm min} = 10$ GeV, $E_{CM\rm min} = 20$ GeV. The third constructor gives maximum as well as minimum values of the three variables, so that nonoverlapping event samples can be generated. For

any other process, the multiple constructors that are avaiable to the user should be documented in the `.h` file in which the `process` class is defined.

If the user would like to place additional cuts on the kinematic variables before event generation, this can be done straightforwardly with some additional knowledge of the structure of the `process` class. The strategy is to create a new class derived from the basic `process` class in which the `computeKinematics` function is modified. Typically, a `process` class contains in its class data an `LVlist` (a list of Lorentz vectors as described in Section 2.4) which contains the lab-frame 4-vectors of the final state particles, before implementation of particle decays. For $e^+e^- \to t\bar{t}$, for example, this is a 2-element list containing the 4-vectors of the t and $\overline{t}$. The class defined in Fig. 4 accepts only events in which both the t and the $\overline{t}$ are both produced in the forward direction in the lab frame. In principle, cuts of any complexity can be implemented. However, `pandora` does not offer a similar shortcut to cut on the distributions of the final partons after all decays.

5.4 Beams and processes included in pandora 2.3

We will now give the list of processes that are included in the current distribution `pandora` 2.3. The distribution is arranged into a number of subdirectories. The directory `engine` contains the fundamental classes, including the definition of the `pandora` class and the integrator. The `beam` classes described in Section 4 are also contained in this directory.

The directory `SMprocesses` contains the basis $2 \to 2$ Standard Model processes at linear colliders. From e^+e^-,

$$\begin{aligned}
\texttt{eetopairs}: &\quad e^+e^- \to \ell^+\ell^- \ , \ q\overline{q} \ , \ e^+e^- \ , \ \gamma\gamma \\
\texttt{eetobosons}: &\quad e^+e^- \to Z^0\gamma \ , \ Z^0Z^0 \ , \ W^+W^- \\
\texttt{eetottbar}: &\quad e^+e^- \to t\bar{t} \\
\texttt{eetoZHiggs}: &\quad e^+e^- \to Z^0h^0
\end{aligned} \tag{59}$$

From $\gamma\gamma$,

$$\begin{aligned}
\texttt{ggtopairs}: &\quad \gamma\gamma \to \ell^+\ell^- \ , \ q\overline{q} \ , \ e^+e^- \\
\texttt{ggtoWW}: &\quad \gamma\gamma \to W^+W^- \\
\texttt{ggtottbar}: &\quad \gamma\gamma \to t\bar{t} \\
\texttt{ggtoHiggs}: &\quad \gamma\gamma \to h^0
\end{aligned} \tag{60}$$

From $e\gamma$,

$$\texttt{egtoboson}: \quad e\gamma \to e\gamma \ , eZ^0, \nu W \tag{61}$$

From e^-e^-,

$$\texttt{eeminustoeeminus}: \quad e^-e^- \to e^-e^- \tag{62}$$

```
class eetottbarforward : public eetottbar{

 public:

    eetottbarforward(double Ctheta): ctheta(Ctheta){};

    int computeKinematics(double & J, DVector & X, double S, double beta){
       int valid = eetottbar::computeKinematics(J,X,S,beta);
       if (valid == 0) return 0;
       double ptop = sqrt(SQR(labvectors[1][1]) + SQR(labvectors[1][2])
                                        + SQR(labvectors[1][3]);
       double costop = labvectors[1][3]/ptop;
       if (costop < ctheta) return 0;
       double ptbar = sqrt(SQR(labvectors[2][1]) + SQR(labvectors[2][2])
                                        + SQR(labvectors[2][3]);
       double costbar = labvectors[2][3]/ptbar;
       if (costbar < ctheta) return 0;
       return 1;
    }

 protected:
    double ctheta;
};
```

Figure 4: Definition of an `process` class for $e^+e^- \to t\bar{t}$ which accepts only events in which both t and $\overline{t}$ are produced with $\cos\theta >$ `ctheta` in the lab frame. The `eetottbar` constructor takes no arguments; otherwise, this class would need to be initialized explicitly in the constructor of the subclass.

The directory `SM3processes` contains some $2 \to 3$ Standard Model processes

$$\begin{aligned} \texttt{eetoHiggsnunubar}: &\quad e^+e^- \to h^0\nu\overline{\nu}\,, h^0e^+e^- \\ \texttt{eetognunubar}: &\quad e^+e^- \to \gamma\nu\overline{\nu}\,. \end{aligned} \tag{63}$$

The directory `BSMprocesses` contains some illustrative beyond-the-Standard Model processes.

The classes `eetollbar`, `eetoqqbar`, `ggtollbar`, and `ggtoqqbar` have methods

```
void produceOnly(int Only);
```
(64)

that allow one to select as specific final state as in (57). For $e\gamma$ processes, the four options $e^-\gamma$, γe^- (in which the electron comes from the beam entering in the $-\hat{3}$ direction), $e^+\gamma$ and γe^+ are present as separate `process` classes. For example, $e\gamma \to e\gamma$ is represented by the classes `egtoeg`, `getoeg`, `epgtoepg`, and `geptoepg`.

All processes with massive particles in the final state are built from the tree-level helicity amplitudes. The expression for the complete production and decay process is built by combining these production helicity amplitudes with corresponding polarized decay amplitudes computed by `decay` classes. The coherent sum over helicity states is squared, preserving all final state spin correlations. The `decay` classes for W, Z, t, and the Minimal Standard Model Higgs boson h^0 are defined in the files `WZtdecay` and `Higgsdecay` in the `SMprocesses` directory.

Processes involving the Higgs require input of the Higgs boson mass. This is done through the constructors

```
eetoZHiggs(double mh);          ggtoHiggs(double mh);
```
(65)

These two classes also contain the method

```
void newHiggsmass(double mh);
```
(66)

which sets the Higgs mass to a new value and resets the branching ratios to the corresponding branching ratios of the Minimal Standard Model. By using `pandora` at a deeper level, it is possible to modify the Higgs branching ratios and even to add new decay modes. This is discussed in the more extensive documentation of the Higgs decay class given in [5].

The process $\gamma\gamma \to h^0$ has one further ideosyncrasy. The Higgs boson is described as a Breit-Wigner resonance. In order to collect events under the integral in the case where the Higgs width is small, the Breit-Wigner is artificially expanded to a width of 1 GeV. The total cross section for the reaction is kept fixed at the Standard Model value, as computed from $\Gamma(h^0 \to \gamma\gamma)$. Using the method

```
void resetMinWidth(double width);
```
(67)

the actual width of the resonance can be reset to any smaller value. Of course, the smaller the width, the less efficient the event generation.

6 Conclusions and outlook

In this paper, we have described the current `pandora-pythia` system for linear collider event generation. This system includes treatment of initial beams with energy smearing, bremmstrahlung, and beamstrahlung with realistic parameters, leading-order parton-level physics cross sections, full treatment of polarization, and hadronization of the final partons with PYTHIA and TAUOLA. We believe that this system will be useful for many studies that are needed to prepare for the linear collider experiments.

There are, of course, extensions of this program that would be useful, and that are now under development. We see a need to include reactions with e^+e^- or $\gamma\gamma$ annhilation to multi-parton final states. This requires, among other things, a more clever integrator and event selector than VEGAS. For many purposes, it is useful to include the photons or pairs emitted in initial- and final-state radiation, with their explicit transverse momenta. This requires a new structure for the `pandora` beam classes. It would be useful to include a broader range of beyond-the-Standard Model processes, including a full treatment of minimal supersymmetry. All of these features are under development. We hope they will appear in future releases of `pandora` and `pandora-pythia`.

ACKNOWLEDGEMENTS

We thank Yue Chen, Keisuke Fujii, Howard Haber, Stanislaw Jadach, Willy Langeveld, Tom Markiewicz, Walter Ogburn, Carl Schmidt, Kathy Thompson, and Marvin Weinstein for useful advice. MI thanks Ray Frey and the University of Oregon group, under whose guidance much of this work was done. We are especially grateful to Tim Barklow for encouraging us to begin this project and for guiding its early stages.

A Public methods of `pandora` classes

A.1 Fundamental clases

Pandora is built from its own set of classes implementing complex number, vector, and matrix operations and other mathematical structures. These classes are contained in the directory `engine`. The default random number generator, defined in the files `random.*`, is `ran2()` from [8]. The Monte Carlo selection makes use of a general Monte Carlo interface defined in `MonteCarlo.*`. The `pandora` class is defined to be a subclass of `MonteCarlo`.

The vector and matrix classes of `pandora` are called `IVector`, `IMatrix` for integers, `DVector`, `DMatrix` for doubles, `CVector`, `CMatrix` for complexes. These classes allow arbitrary lower and upper indices. For example, the constructor of a matrix of

`complex` indexed from -1 to 1 in the first argument and from 1 to 9 in the second argument is

```
CMatrix Amps(-1,1,1,9);
```
(68)

Such a matrix might actually be useful to hold helicity amplitudes for e^+e^- annihilation into a final state with two massive vector particles. The vector and matrix classes have overloaded + and * operations and other operations convenient for vector algebra. However, it is important to note that all vectors and matrices have dynamically allocated memory. Thus, it is quite expensive to use these vector operators in operations that are called for each evaluation of the cross section function. In the predefined classes of pandora, all needed vectors and matrices are allocated in advance.

There is no specific object persistence method in `pandora`. Our assumption is that `pandora` event samples used at the parton level will be regenerated for each new analysis, and that samples used at the hadron level will be converted by PYTHIA to an archivable format.

The working of the `pandora` class has been described in Section 2.2. The `pandora` class has the following public methods:

```
pandora(luminosity & Lu, process & Pr);
pandora(beam & B1, beam & B2, process & Pr);
LEvent getEvent();
LEvent getEvent(double & weight);
LEvent initialvectors();
void usecrossingangle(double theta);
ostream &  printEventLabel(ostream & out, int EventNumber = -1);
void prepare(int nevents, int nseed =1);
double setThreshold(double x);
double integral(double & sd);
void printIntegral();
void printStatistics();
void quiet();
```
(69)

The first two of these are the constructors from `beam` and `luminosity` classes. The next two methods return one unweighted or weighted event. The next returns the initial beam vectors as an `LEvent` class with six 4-vector entries, the nominal particles in beams 1 and 2, the colliding partons in beams 1 and 2, and the missing 4-momentum from beam 1 and 2. The next method introduces into all events a crossing angle in the (13) plane, measured in mrad. The next method prints to `cout` an event label giving the process, beam energies, and beam polarizations.

The methods that follow are inherited from `MonteCarlo`. The method `prepare` adapts the Monte Carlo selection, using the given number of function calls. The next method adjusts the estimate of the maximum weight used in event selection to be x times the highest weight observed. (The default value is $x = 2$.) The next method returns the cross section integral. The next two methods print the cross section integral and the statistics of the event selection to `cout`. Finally, the last method suppresses `pandora`'s normal printing of Monte Carlo adaptation information.

The `pandora` class also has public data members

```
s, beta, missingEforward, missingEbackward
```
(70)

After `getEvent` is called, this data is filled in with the parton-parton CM energy squared and the boost of the most recently selected event, and the energy going down the beampipe in the $+\hat{3}$ and $-\hat{3}$ directions.

All of these methods and data members are also defined for the `pandorasBox` class. In addition, the `pandorasBox` class defines the methods

```
int checkID(process & Pr);
IVector chosen();
DVector fractions();
void reset();
```
(71)

The first of these methods returns 1 if the most recently selected event is of the process `Pr`, and 0 otherwise. The next two methods return vectors that give the number of events selected and the fraction of the total for each process. Finally `reset` resets these counters.

To test particle decay simulations, `pandora` also includes a class `decaypandora`, which is created from a decay class. For reasons to be explained in [5], all decay classes in `pandora` are referred to by pointers rather than references. So the constructor for this class is

```
decaypandora(decay * D, double mass, int J3);
```
(72)

The additional variables passed are the mass of the particle decaying and the angular momentum state J^3. For unpolarized decays, call the class method

```
unpolarized(int states);,
```
(73)

setting `states` to be $(2J + 1)$. The `decaypandora` class has the `getEvent` methods from (69) and the additional methods beginning with `prepare`.

A.2 Methods of the 4-vector and event classes

A 4-vector or Lorentz vector is represented in `pandora` as an `LVector` class. This is a `DVector` indexed from 0 to 3 with additional methods defined. An `LVector` can be constructed from four doubles

```
LVector LV(V0,V1,V2,V3);
```
(74)

as well as by the more general `DVector` operations.

The methods defined for the `LVector` class include

```
LVector lower();
LVector raise();
double masssq();
```
(75)

The first two lower and raise 4-vector indices. The third returns V^2 for a 4-vector V. For a 4-vector that could be a particle 4-vector, the following operations are useful:

```
LVector lower();
double E();
double p();
double cost();
double sint();
double sinphi();
double cosphi();
double mass();
```
(76)

In `pandora`, these operations are defined for all 4-vectors, without checking that, for example, $E \geq p$. The angles are all defined with respect to the fixed lab coordinate system of `pandora` which takes the beams to be along the $\hat{3}$ axis or, in the case of a crossing angle, in the (13) plane.

Often, we wish to deal with 4-vectors in groups, for example, the decay products of a given particle. For this purpose, `pandora` defines an `LVlist` class, which is a matrix indexed in its first argument from 1 to N and in its second from 0 to 3. Basic methods of the `LVlist` are

```
LVlist(int N);
int n()
void read(int m, const LVector & V);
void read(int m, double V0, double V1, double V2, double V3);
void readlist(int n, int nfirst, int mfirst, const LVlist & L);
LVector readout(int m);
```
(77)

The first of these is the constructor. The second returns the number of vectors in the list. The next two input the vector m, and the next reads in a list of n vectors in the `LVlist L`, beginning with `nfirst`, into this list, beginning with the row `mfirst`. The last method reads out the vector m as an `LVector`.

Lorentz transformations can be applied to the whole list of vectors in an `LVlist`. It is possible to apply a general Lorentz transformation, but it is often easier to apply a sequence of Lorentz transformations along fixed axes. The methods provided by `pandora` are:

```
void generalboost(const LMatrix & Lambda);
void boost(double beta);
void transverseboost(double beta);
void rotate(double phi);
void rotateinplane(double cost);
void rotatebackinplane(double cost);
void reverseinplane();
void boostto(const LVector & V);
void boostfrom(const LVector & V);                    (78)
```

The first of these methods transforms the `LVlist` by a general Lorentz transformation. The remainder are more specialized; they are, respectively, a boost along the $\hat{3}$ axis by `beta`, a boost along the $\hat{1}$ axis by `beta`, a rotation about the $\hat{3}$ axis by `phi` (in radians), a rotation in the (31) plane by $\theta = \cos^{-1}(\texttt{cost})$, a rotation in (31) plane in the opposite direction, reflection in the (31) plane, a boost to the rest frame of `V`, and a boost from the rest frame of `V` to the lab frame.

Finally, the event structure in `pandora` is given by the class `LEvent`, which contains an `LVlist` of 4-vectors plus the additional information described in Section 2.4. The various pieces of information in an `LEvent` can be read out by the methods

```
LVector readout(int m);
int id(int m);
int parent(int m);
int final(int m);
int chain(int m);
int level(int m);
LVector readout(int m, int & id, int & parent, int & final,
 int & chain, int& level);
  double E(int m);
  double p(int m);
```

```
double cost(int m);
double sint(int m);
double sinphi(int m);
double cosphi(int m);
double mass(int m);                                    (79)
```

which return the various properties of the mth entry. The content of an `LEvent LE` can also be listed by the command

```
cout << LE;                                            (80)
```

The Lorentz transformation methods (78) are also defined as methods of the `LEvent` class and can be used to transform the set of vectors in an `LEvent`.

B Public methods of `pandora-pythia`

Common methods in `pandorarun` and `pandorasBoxrun`

void initialize(char* ofile)	Initialization step, calling the initialization of pandora, PYTHIA and TAUOLA. Input argument is an output file name.
void terminate()	Termination step. Print summary of the event generation.
void getevent()	Generate a single event.
void getevents()	Event generations. The number of the event generated is provided by an input of pandorarun or pandorasBoxrun constructor.
void fragmentationoff()	Turn off the fragmentation process. item[void FSRoff()] Turn off final-state QCD and QED radiations.
void partonshoweroff()	Turn off parton showering.
void newseed(int seed)	Change the initial seed for pandora to be seed.
void PrintEvent()	List the event summary.
void SetPrintEvent(int i)	Set the number of events being listed to be i.
void StdHepIOoff()	Do not use the STDHEP I/O.

void StdHepIOon()	Use the STDHEP I/O.
void SetASCII(int unit)	Output the HEPEVT common to an ascii file. The input arguments is for the logical unit of the output file.
void PartDecayOff(int id)	Not allow to decay the particle with PDG particle ID number id.
void PartDecayOn(int id)	Allow to decay the particle with PDG particle ID number id.
int getPyjetN()	Get N in PYJET common.
int getPyjetNPAD()	Get NPAD in PYJET common.
int getPyjetK(int ip, int i)	Get K(ip,i) in PYJET common.
double getPyjetP(int ip, int i)	Get P(ip,i) in PYJET common.
double getPyjetV(int ip, int i)	Get V(ip,i) in PYJET common.
int Ngenerate()	Get number of the event generated in this event generation.
int getPandoraNpart()	Get number of partons from the pandora event generation.
int getPandoraPartID(int ip)	Get particle ID of the ip'th parton from pandora.
double getPandoraMom(int ip, int i)	Get 4-momentum element of ip'th parton from pandora. (i= 0 px, 1 py, 2 pz, 3 E).

Methods in pandorarun

pandora *P	The pandora pointer to be used for event generation.
pandorarun(pandora *P, int Ngen, int Iseed, int Ncall=100000)	Constructor of the pandorarun. The inputs are pointer of pandora, the number of events to be generated and an initial seed for pandora. The fourth argument is an option, which is used for VEGAS grid determination.

Methods in pandorasBoxrun

pandorasBox Box	The pandorasBox object to be used for event generation.
pandorasBoxrun(int Ngen, int Iseed, int Ncall=100000)	Constructor of the pandorasBoxrun. The inputs are number of events to be generated and an initial seed for pandora. The third argument is an option, which is used for VEGAS grid determination.
void addPandora(pandora &P)	Add the pandora pointers to be generated to pandorasBox.

C pandora-pythia's example programs

Example 1 : Program to generate $e^+e^- \to Z^0H^0$, Z^0 decaying to leptons.

```
#include "pandora.h"
#include "ebeams.h"
#include "eetoZHiggs.h"
#include "pandorarun.h"

int main(int argc, char * argv[]){
    int   nEvent    = atoi(argv[1]);   // # of events to be generated
    char* outfile   = argv[2];         // out put file name
    int   iseed_pan = atoi(argv[3]);   // Seed number for pandora

    /*  begin pandora code here  */
    double ECM = 500.0;             // Center of Mass energy
    double epolarization = -0.8;    // Electron poralization
    double ppolarization =  0.0;    // Positron poralization

    ebeam B1(ECM/2.0, epolarization, electron, electron);
    B1.setup(NLC500H);
  //B1.ISRoff();            // Turn off initial-state radiation
  //B1.beamstrahlungoff();  // Turn off beamstrahlung

    ebeam B2(ECM/2.0, ppolarization, positron, positron);
    B2.setup(NLC500H);
  //B2.ISRoff();            // Turn off initial-state radiation
  //B2.beamstrahlungoff();  // Turn off beamstrahlung

    double Hmass = 140.; // Higgs Mass
```

```
    eetoZHiggs Rxn(Hmass);
  //For example:: Z0 decays to leptons. H0 decays to anything.
    Rxn.onlyDecay(leptonsOnly,allStates);
    pandora P(B1,B2,Rxn);
    /*  end pandora code  */

    /*  begin pandora-pythia code  */
    pandorarun PR(P,nEvent,iseed_pan);

    int iseed_pyth = 22846;
    PR.newseed(iseed_pyth); // Change PYTHIA Seed for example
  //PR.partonshoweroff();   // Turn off parton shower
  //PR.fragmentationoff();  // Turn off fragmentation
  //PR.FSRoff();            // Turn off final-state QCD QED radiation

    PR.initialize(outfile);

  //If you don't want to decay some specific particle(s),
  //you can call PartDecayOff here.
  //For example: Do not allow to decay K0S and Lambda
    PR.PartDecayOff(310);  // K0S
    PR.PartDecayOff(3122); // Lambda

    PR.getevents();
    PR.terminate();
    /*  end pandora-pythia code  */

    return 0;
}
```

Example 2 : Program to generate $e^+e^- \to t\overline{t}$ and $e^+e^- \to q\overline{q}$ ($q = u, d, s, c, b$) using the `pandorasBoxrun` class.

```
#include "pandora.h"
#include "ebeams.h"
#include "eetottbar.h"    // For ee -> tt-bar
#include "eetopairs.h"    // For ee -> qq-bar
#include "pandorasBoxrun.h"

int main(){
    int   nEvent    = 1000;          // # events to be generated
    char* outfile   = ``output.dat'';// Out put file name
```

```
    int   iseed_pan = 1;             // pandora initial seed

    /*  begin pandora code here  */
    double ECM  = 500.;
    double PolE = -0.8; // Polarization Electron
    double PolP =  0.0; // Polarization Positron

    ebeam B1(ECM/2.0, PolE, electron, electron);
    B1.setup(NLC500);

    ebeam B2(ECM/2.0, PolP, positron, positron);
    B2.setup(NLC500);
    /*  end pandora code  */

    /*  begin pandora-pythia code  */
    pandorasBoxrun  PB(nEvent, iseed_pan);

  //process 1  ee -> tt-bar
    eetottbar Rxn1;
    pandora P1(B1,B2,Rxn1);
    PB.addPandora(P1);

  //process 2  ee -> qq-bar
    double ECMmin = 20.;
    eetoqqbar Rxn2(ECMmin);
    pandora P2(B1,B2,Rxn2);
    PB.addPandora(P2);

    PB.SetPrintEvent(2);   // Set # events to be listed (default = 5 )

    PB.initialize(outfile);
    PB.getevents();
    PB.terminate();
    /*  end pandora-pythia code  */

    return 0;
}
```

References

[1] T. Ohl, Comput. Phys. Commun. **101**, 269 (1997) [arXiv:hep-ph/9607454].

[2] PYTHIA: T.Sjöstrand *et al.*, Comput. Phys. Commun. **135** 238 (2001).

[3] G. Corcella *et al.*, JHEP **0101**, 010 (2001) [arXiv:hep-ph/0011363].

[4] TAUOLA: S.Jadach *et al.*, Comput. Phys. Commun. **76** 361 (1993).

[5] M. E. Peskin, in preparation.

[6] The pandora and pandora-pythia programs are available for download at `ftp://ftp.slac.stanford.edu/public/groups/nld/Generators/PANDORA` and `PANDORA_PYTHIA`, respectively. The distribution current as of the date of this paper is pandora 2.3 and pandora-pythia 3.2. Very recent bug corrections and patches for pandora are posted at w̆ww.slac.stanford.edu/ mpeskin/pandora.html.

[7] G. P. Lepage, J. Comput. Phys. **27**, 192 (1978).

[8] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes in C* (Cambridge University Press, 1992).

[9] S. Kawabata, Comput. Phys. Commun. **41**, 127 (1986); Comput. Phys. Commun. **88**, 309 (1995).

[10] K. Hagiwara *et al.* [Particle Data Group Collaboration], Phys. Rev. D **66**, 010001 (2002).

[11] M. Kramer, J. H. Kuhn, M. L. Stong and P. M. Zerwas, Z. Phys. C **64**, 21 (1994) [arXiv:hep-ph/9404280].

[12] We are very grateful to K. Fujii for discussions of this point.

[13] `http://www-pat.fnal.gov/stdhep.html`

[14] E. A. Kuraev and V. S. Fadin, Sov. J. Nucl. Phys. **41**, 466 (1985) [Yad. Fiz. **41**, 733 (1985)].

[15] O. Nicrosini and L. Trentadue, Z. Phys. C **39**, 479 (1988).

[16] M. Skrzypek and S. Jadach, Z. Phys. C **49**, 577 (1991).

[17] R. Blankenbecler and S. D. Drell, Phys. Rev. D **36**, 277 (1987).

[18] `http://heplix.ikp.physik.tu-darmstadt.de/lc/beam.html`.

[19] K. Yokoya and P. Chen, in *Frontiers of Particle Beams: Intensity Limitations*, M. Dienes, M. Month,and S. turner, eds. (Springer-Verlag, 1992).

[20] P. Chen, Phys. Rev. D **46**, 1186 (1992).

[21] P. Chen, T. L. Barklow and M. E. Peskin, Phys. Rev. D **49**, 3209 (1994) [arXiv:hep-ph/9305247].

[22] For more details of the beam parametrization, see M. E. Peskin, LC-0010 (1999), available at `http://www-project.slac.stanford.edu/lc/ilc/TechNotes/LCCNotes/lcc_notes_index.htm`.

[23] P. Chen, G. Horton-Smith, T. Ohgaki, A. W. Weidemann and K. Yokoya, Nucl. Instrum. Meth. A **355**, 107 (1995).

[24] `http:/www.slac.stanford.edu/xorg/icap98/papers/F-Th24.pdf`.

[25] T. Raubenheimer, personal communication.

[26] M. Battaglia, personal communication.

[27] S. Jadach has emphasized this point to us.

[28] K. A. Thompson and P. Chen, SLAC-PUB-8230.

[29] K. A. Thompson, Int. J. Mod. Phys. A **15**, 2485 (2000).

[30] I. F. Ginzburg, G. L. Kotkin, V. G. Serbo and V. I. Telnov, Nucl. Instrum. Meth. **205**, 47 (1983).

[31] V. I. Telnov, Nucl. Instrum. Meth. A **294**, 72 (1990).

[32] D. M. Asner, J. B. Gronberg and J. F. Gunion, arXiv:hep-ph/0110320.