`Circe2`: Beam Spectra for Simulating Linear Collider and Photon Collider Physics*

Thorsten Ohl†

Institut für Theoretische Physik und Astrophysik
Universität Würzburg
Emil-Hilb-Weg 22
97074 Würzburg
Germany

WUE-ITP-2002-006
LC-TOOL-2002-???
hep-ph/yymmnnn
August 2002
DRAFT: June 2, 2014

Abstract

...

*Supported by Bundesministerium für Bildung, Wissenschaft, Forschung und Technologie, Germany.

†e-mail: `ohl@physik.uni-wuerzburg.de`

Program Summary:

- **Title of program:** `Circe2` (August 2002)
- **Program obtainable** from `http://theorie.physik.uni-wuerzburg.de/~ohl/circe2/`.
- **Licensing provisions:** Free software under the GNU General Public License.
- **Programming languages used:** originally FORTRAN77, transferred to FORTRAN90, Objective Caml[8] (available from `http://caml.inria.fr/ocaml/`).
- **Number of program lines in distributed program** $\approx$ 800 lines of FORTRAN90 (excluding comments) for the library; $\approx$ 4000 lines of Objective Caml for the utility program
- **Computer/Operating System:** Any with a FORTRAN90 programming environment.
- **Memory required to execute with typical data:** Negligible on the scale of typical applications calling the library.
- **Typical running time:** A negligible fraction of the running time of applications calling the library.
- **Purpose of program:** Provide efficient, realistic and reproducible parameterizations of the correlated $e^{\pm}$- and γ-beam spectra for linear colliders and photon colliders.
- **Nature of physical problem:** The intricate beam dynamics in the interaction region of a high luminosity linear collider at $\sqrt{s} = 500\text{GeV}$ result in non-trivial energy spectra of the scattering electrons, positrons and photons. Physics simulations require efficient, reproducible, realistic and easy-to-use parameterizations of these spectra.
- **Method of solution:** Parameterization, curve fitting, adaptive sampling, Monte Carlo event generation.
- **Keywords:** Event generation, beamstrahlung, linear colliders, photon colliders.

1 Introduction

The expeditious construction of a high-energy, high-luminosity e^+e^- Linear Collider (LC) to complement the Large Hadron Collider (LHC) has been identified as the next world wide project for High Energy Physics (HEP). The dynamics of the dense colliding beams providing the high luminosities required by such a facility is highly non-trivial and detailed simulations have to be performed to predict the energy spectra provided by these beams. The microscopic simulations of the beam dynamics require too much computer time and memory for direct use in physics programs. Nevertheless, the results of such simulations have to be available as input for physics studies, since these spectra affect the sensitivity of experiments for the search for deviations from the standard model and to new physics.

`Circe1` [1] has become a de-facto standard for inclusion of realistic energy spectra of TeV-scale e^+e^- LCs in physics calculations and event generators. It is supported by the major multi purpose event generators [2, 3] and has been used in many dedicated analysises. `Circe2` provides a fast, concise and convenient parameterization of the results of such simulations.

`Circe1` assumed strictly factorized distributions with a very restricted functional form (see [1] for details). This approach was sufficient for exploratory studies of physics at TeV-scale e^+e^- LCs. Future studies of physics at e^+e^- LCs will require a more detailed description and the estimation of non-factorized contributions. In particular, all distributions at laser backscattering $\gamma\gamma$ colliders [4] and at multi-TeV e^+e^- LCs are correlated and can not be approximated by `Circe1` at all. In addition, the proliferation of accelerator designs since the release of `Circe1` has made the maintenance of parameterizations as FORTRAN77 `BLOCK DATA` unwieldy.

`Circe2` successfully addresses these shortcomings of `Circe1`, as can be seen in figure 1. It should be noted that the large z region and the blown-up $z \to 0$ region are taken from the *same* pair of datasets. In section 6.2 below, figures 3 to 9 demonstrate the interplay of `Circe2`'s features. The algorithms implemented[1] in `Circe2` should suffice for all studies until e^+e^- LCs and photon colliders come on-line and probably beyond. The implementation `Circe2` bears no resemblance at all with the implementation of `Circe1`.

`Circe2` describes the distributions by two-dimensional grids that are optimized using an algorithm derived from VEGAS [5]. The implementation was modeled on the implementation in VAMP [6], but changes were required for sampling static event sets instead of distributions given as functions. The

[1] A small number of well defined extensions that has have not been implemented yet are identified in section 3 below.

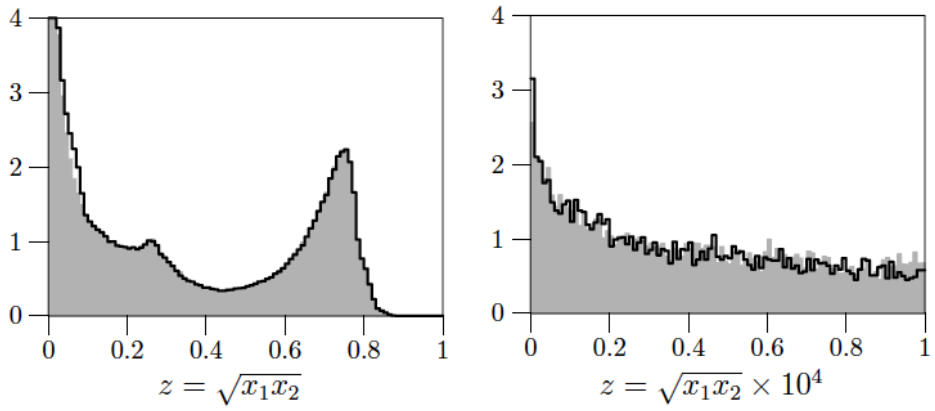

Figure 1: Comparison of a simulated realistic $\gamma\gamma$ luminosity spectrum (helicities: $(+,+)$) for a 500 GeV photon collider at TESLA [7] (filled area) with its `Circe2` parameterization (solid line) using 50 bins in both directions. The 10^4-fold blow-up of the $z \to 0$ region is taken from the same pair of datasets as the plot including the large z region.

problem solved by `Circe2` is rather different from the Monte Carlo integration with importance or stratified sampling that is the focus of VEGAS and VAMP. In the case of VEGAS/VAMP the function is given as a mathematical function, either analytically or numerically. In this case, while the adapted grid is being be refined, resources can be invested for studying the function more closely in problematic regions. `Circe2` does not have this luxury, because it must reconstruct ("*guess*") a function from a *fixed* and *finite* sample. Therefore it cannot avoid to introduce biases, either through a fixed global functional form (as in `Circe1`) through step functions (histograms). `Circe2` combines the two approaches and uses automatically adapted histograms mapped by a patchwork of functions.

1.1 Notes on the Implementation

The FORTRAN90 library is extremely simple (about 800 lines) and performs only two tasks: one small set of subroutines efficiently generates pairs of random numbers distributed according to two dimensional histograms with factorized non-uniform bins stored in a file. A second set of functions calculates the value of the corresponding distributions.

In the original implementation, FORTRAN77 has been chosen solely for practical reasons: at the time of the original writing, the majority of pro-

grams expected to use the `Circe2` have been legacy applications written in FORTRAN77. The simple functionality of the FORTRAN90 library can however be reproduced trivially in any other programming language that will be needed in the future.

The non-trivial part of constructing an optimized histogram from an arbitrary distribution is performed by a utility program `circe2_tool` written in Objective Caml [8] (or O'Caml for short). O'Caml is available as Free Software for almost all computers and operating systems currently used in high energy physics. Bootstrapping the O'Caml compiler is straightforward and quick. Furthermore, parameterizations are distributed together with `Circe2`, and most users will not even need to compile `circe2_tool`. Therefore there are no practical problems in using a modern programming language like O'Caml that allows—in the author's experience—a both more rapid and safer development than FORTRAN77 or C++.

1.2 Overview

The remainder of this paper is organized as follows. For the benefit of users of the library, the Application Program Interface (API) is described immediately in section 3 after defining the notation in section 2. Section 4 shows some examples using the procedures described in section 3.

A description of the inner workings of `Circe2` that is more detailed than required for using the library starts in section 5. An understanding of the algorithms employed is helpful for preparing beam descriptions using the program `circe2_tool` which is described in section 6. Details of the implementation of `circe2_tool` can be found in section 7, where also the benefits provided by modern functional programming languages for program organization in the large are discussed.

2 Physics

The customary parametrization of polarization in beam physics [9, 10] is in terms of density matrices for the leptons

$$\rho_{e^\pm}(\zeta) = \frac{1}{2}\left(1 + \zeta_i \sigma_i\right) \tag{1}$$

and the so-called Stokes' parameters for photons

$$\rho_{\gamma}(\xi) = \frac{1}{2}\left(1 + \xi_i \sigma_i\right) \tag{2}$$

where the pseudo density matrix 2×2-matrix ρ_γ for a pure polarization state ϵ_μ is given by

$$[\rho_\gamma]_{ij} = \langle (\epsilon e_i)(\epsilon^* e_j) \rangle \tag{3}$$

using two unit vectors $e_{1/2}$ orthogonal to the momentum. Keeping in mind the different interpretations of ζ and ξ, we will from now on unify the mathematical treatment and use the two interchangably, since the correct interpretation will always be clear from the context. Using the notation $\sigma_0 = 1$, the joint polarization density matrix for two colliding particles can be written

$$\rho(\chi) = \sum_{a,a'=0}^{3} \frac{\chi_{aa'}}{4} \sigma_a \otimes \sigma_{a'} \tag{4}$$

with $\chi_{0,0} = \operatorname{tr} \rho(\chi) = 1$. Averaging density matrices will in general lead to correlated density matrices, even if the density matrices that are being averaged are factorized or correspond to pure states.

The most complete description B of a pair of colliding beams is therefore provided by a probability density and a density matrix for each pair (x_1, x_2) of energy fractions:

$$\begin{aligned} B : [0,1] \times [0,1] &\to \mathbf{R}^+ \times M \\ (x_1, x_2) &\mapsto (D(x_1, x_2), \rho(x_1, x_2)) \end{aligned} \tag{5}$$

where $\rho(x_1, x_2)$ will conveniently be given using the parametrization (4). Sophisticated event generators can use $D(x_1, x_2)$ and $\rho(x_1, x_2)$ to account for all spin correlations with the on-shell transition matrix T

$$\mathrm{d}\sigma = \int \mathrm{d}x_1 \wedge \mathrm{d}x_2 \, D(x_1, x_2) \operatorname{tr}\left(P_\Omega T(x_1 x_2 s) \rho(x_1, x_2) T^\dagger(x_1 x_2 s) \right) \mathrm{dLIPS} \tag{6}$$

2.1 Polarization Averaged Distributions

Physics applications that either ignore polarization (this is often not advisable, but can be a necessary compromise in some cases) or know that polarization will play no significant role can ignore the density matrix, which amounts to summing over all polarization states. If the microscopic simulations that have been used to obtain the distributions described by `Circe2` do not keep track of polarization, 93% of disk space can be saved by supporting simplified interfaces that ignore polarization altogether.

2.2 Helicity Distributions

Between the extremes of polarization averaged distributions on one end and full correlated density matrices on the other end, there is one particularly

important case for typical applications, that deserves a dedicated implementation.

In the approximation of projecting on the subspace consisting of circular polarizations

$$\rho(\chi) = \frac{1}{4}\left(\chi_{0,0}\cdot 1\otimes 1 + \chi_{0,3}\cdot 1\otimes\sigma_3 + \chi_{3,0}\cdot\sigma_3\otimes 1 + \chi_{3,3}\cdot\sigma_3\otimes\sigma_3\right) \tag{7}$$

the density matrix can be rewritten as a convex combination of manifest projection operators build out of $\sigma_\pm = (1\pm\sigma_3)/2$:

$$\rho(\chi) = \chi_{++}\cdot\sigma_+\otimes\sigma_+ + \chi_{+-}\cdot\sigma_+\otimes\sigma_- + \chi_{-+}\cdot\sigma_-\otimes\sigma_+ + \chi_{--}\cdot\sigma_-\otimes\sigma_- \tag{8}$$

The coefficients are given by

$$\chi_{++} = \frac{1}{4}\left(\chi_{0,0} + \chi_{0,3} + \chi_{3,0} + \chi_{3,3}\right) \geq 0 \tag{9a}$$

$$\chi_{+-} = \frac{1}{4}\left(\chi_{0,0} - \chi_{0,3} + \chi_{3,0} - \chi_{3,3}\right) \geq 0 \tag{9b}$$

$$\chi_{-+} = \frac{1}{4}\left(\chi_{0,0} + \chi_{0,3} - \chi_{3,0} - \chi_{3,3}\right) \geq 0 \tag{9c}$$

$$\chi_{--} = \frac{1}{4}\left(\chi_{0,0} - \chi_{0,3} - \chi_{3,0} + \chi_{3,3}\right) \geq 0 \tag{9d}$$

and satisfy

$$\chi_{++} + \chi_{+-} + \chi_{-+} + \chi_{--} = \operatorname{tr}\rho(\chi) = 1 \tag{10}$$

Of course, the $\chi_{\epsilon_1\epsilon_2}$ are recognized as the probabilities for finding a particular combination of helicities for particles moving along the $\pm\vec{e}_3$ direction and we can introduce partial probability distributions

$$D^{\epsilon_1\epsilon_2}_{p_1p_2}(x_1,x_2) = \chi_{\epsilon_1\epsilon_2}\cdot D_{p_1p_2}(x_1,x_2) \geq 0 \tag{11}$$

that are to be combined with the polarized cross sections

$$\frac{\mathrm{d}\sigma}{\mathrm{d}\Omega}(s) = \sum_{\epsilon_1,\epsilon_2=\pm}\int \mathrm{d}x_1\wedge\mathrm{d}x_2\, D^{\epsilon_1\epsilon_2}(x_1,x_2)\,\left(\frac{\mathrm{d}\sigma}{\mathrm{d}\Omega}\right)^{\epsilon_1\epsilon_2}(x_1x_2s) \tag{12}$$

This case deserves special consideration because it is a good approximation for a majority of applications and, at the same time, it is the most general case that allows an interpretation as classical probabilities. The latter feature allows the preparation of separately tuned probability densities for all four helicity combinations. In practical applications this turns out to be useful because the power law behaviour of the extreme low energy tails turns out to have a mild polarization dependence.

load beam	from file	`cir2ld`	(p. 9)
	from `block data`	`cir2lb`	(optional, p. 16)
distributions	luminosity	`cir2lm`	(p. 11)
	probability density	`cir2dn`	(p. 14)
	density matrix	`cir2dm`	(extension, p. 16)
event generation	flavors/helicities	`cir2ch`	(p. 12)
	(x_1, x_2)	`cir2gn`	(p. 12)
	general polarization	`cir2gp`	(extension, p. 14)
internal	current beam	`/cir2cm/`	(p. 16)
	beam data base	`cir2bd`	(optional, p. 16)
	(cont'd)	`/cir2cd/`	(optional, p. 16)

Table 1: Summary of all functions, procedures and comon blocks.

3 API

All floating point numbers in the interfaces are declared as `double precision`. In most applications, the accuracy provided by single precision floating point numbers is likely to suffice. However most application programs will use double precision floating point numbers anyway so the most convenient choice is to use double precision in the interfaces as well.

In all interfaces, the integer particle codes follow the conventions of the Particle Data Group [11]. In particular

`p = 11`: electrons

`p = -11`: positrons

`p = 22`: photons

while other particles are unlikely to appear in the context of `Circe2` before the design of μ-colliders enters a more concrete stage. Similarly, in all interfaces, the sign of the helicities are denoted by integers

`h = 1`: helicity $+1$ for photons or $+1/2$ for leptons (electrons and positrons)

`h = -1`: helicity -1 for photons or $-1/2$ for leptons (electrons and positrons)

As part of tis API, we also define a few extensions, which will be available in future versions, but have not been implemented yet. This allows application programs to anticipate these extensions.

3.1 Initialization

Before any of the event generation routines or the functions computing probability densities can be used, beam descriptions have to be loaded. This is accomplished by the routine `cir2ld` (mnemonic: *LoaD*), which must have been called at least once before any other procedure is invoked:

```
subroutine cir2ld (file, design, roots, ierror)
```

`character*(*) file` (input): name of a `Circe2` parameter file in the format described in table 2. Conventions for filenames are system dependent and the names of files will consequently be installation dependent. This can not be avoided.

`character*(*) design` (input): name of the accelerator design. The name must not be longer than 72 characters. It is expected that design names follow the following naming scheme for e^+e^- LCs

`TESLA`: TESLA supercoducting design (DESY)

`XBAND`: NLC/JLC X-band design (KEK, SLAC)

`CLIC`: CLIC two-beam design (CERN)

Special operating modes should be designated by a qualifier

`/GG`: laser backscattering $\gamma\gamma$ collider (e. g. `'TESLA/GG'`)

`/GE`: laser backscattering γe^- collider

`/EE`: e^-e^- collider

If there is more than one matching beam description, the *last* of them is used. If `design` contains a `'*'`, only the characters *before* the `'*'` matter in the match. E. g.:

`design = 'TESLA'` matches only `'TESLA'`

`design = 'TESLA*'` matches any of `'TESLA (Higgs factory)'`, `'TESLA (GigaZ)'`, `'TESLA'`, etc.

`design = '*'` matches everything and is a convenient shorthand for the case that there is only a single design per file

NB: `'*'` is not a real wildcard: everything after the first `'*'` is ignored.

`double precision roots` (input): $\sqrt{s}$/GeV of the accelerator. This must match within $\Delta\sqrt{s} = 1\,\mathrm{GeV}$. There is currently no facility for interpolation between fixed energy designs (see section 4.3, however).

`integer ierror` (input/output): if `ierror` > 0 on input, comments will be echoed to the standard output stream. On output, if no errors have been encountered `cir2ld` guarantees that `ierror` $= 0$. If `ierror` < 0, an error has occured:

- `ierror = -1`: file not found
- `ierror = -2`: no match for design and $\sqrt{s}$
- `ierror = -3`: invalid format of parameter file
- `ierror = -4`: parameter file too large

A typical application, assuming that a file named `photon_colliders.circe` contains beam descriptions for photon colliders (including `TESLA/GG`) is

```
integer ierror
...
ierror = 1
call cir2ld ('photon_colliders.circe', 'TESLA/GG', 500D0, ierror)
if (ierror .lt. 0)
  print *, 'error: cir2ld failed: ', ierror
  stop
end if
...
```

In order to allow application programs to be as independent from operating system dependent file naming conventions, the file formal has been designed so beam descriptions can be concatenated and application programs can hide file names from the user completely, as in

```
subroutine ldbeam (design, roots, ierror)
implicit none
character*(*) design
double precision roots
integer ierror
call cir2ld ('beam_descriptions.circe', design, roots, ierror)
if (ierror .eq. -1)
  print *, 'ldbeam: internal error: file not found'
  stop
end if
end
```

The other extreme uses one file per design and uses the `'*'` wildcard to make the `design` argument superfluous.

```
subroutine ldfile (name, roots, ierror)
implicit none
character*(*) name
double precision roots
integer ierror
call cir2ld (name, '*', roots, ierror)
end
```

Note that while it is in principle possible to use a data file intended for helicity states for polarization averaged distributions instead, no convenience procedures for this purpose are provided.

3.2 Luminosities

One of the results of the simulations that provide the input for `Circe2` are the partial luminosities for all combinations of flavors and helicities. The luminosities for a combination of flavors and helicities can be inspected with the function `cir2lm` (*LuMinosity*). The return value is given in the convenient units

$$\mathrm{fb}^{-1}\upsilon^{-1} = 10^{32}\mathrm{cm}^{-2}\mathrm{sec}^{-1} \tag{13}$$

where $\upsilon = 10^7\,\mathrm{sec} \approx \mathrm{year}/\pi$ is an "effective year" of running with about 30% up-time

`function cir2lm (p1, h1, p2, h2)`

`integer ::` `p1` (input): particle code for the first particle

`integer ::` `h1` (input): helicity of the first particle

`integer ::` `p2` (input): particle code for the second particle

`integer ::` `h2` (input): helicity of the second particle

For the particle codes and helicities the special value 0 can be used to imply a sum over all flavors and helicities. E. g. the total luminosity is obtained with

```
lumi = cir2lm (0, 0, 0, 0)
```

and the $\gamma\gamma$ luminosity summed over all helicities

```
lumigg = cir2lm (22, 0, 22, 0)
```

3.3 Sampling and Event Generation

Given a combination of flavors and helicities, the routine `cir2gn` (*GeNerate*) can be called repeatedly to obtain a sample of pairs (x_1, x_2) distributed according to the currently loaded beam description:

```
subroutine cir2gn (p1, h1, p2, h2, x1, x2, rng)
```

`integer p1` (input): particle code for the first particle

`integer h1` (input): helicity of the first particle

`integer p2` (input): particle code for the second particle

`integer h2` (input): helicity of the second particle

`double precision x1` (output): fraction of the beam energy carried by the first particle

`double precision x2` (output): fraction of the beam energy carried by the second particle

`external rng`: subroutine

```
subroutine rng (u)
double precision u
u = ...
end
```

generating a uniform deviate, i. e. a random number uniformly distributed in $[0, 1]$.

If the combination of flavors and helicities has zero luminosity for the selected accelerator design parameters, *no error code* is available (`x1` and `x2` are set to a very large negative value in this case). Applications should use `cir2lm` to test that the luminosity is non vanishing.

Instead of scanning the luminosities for all possible combinations of flavors and helicities, applications can call the procedure `cir2ch` (*CHannel*) which chooses a "channel" (a combination of flavors and helicities) for the currently loaded beam description with the relative probabilities given by the luminosities:

```
subroutine cir2ch (p1, h1, p2, h2, rng)
```

`integer p1` (output): particle code for the first particle

`integer h1` (output): helicity of the first particle

`integer p2` (output): particle code for the second particle

`integer h2` (output): helicity of the second particle

`external rng`: subroutine generating a uniform deviate (as above)

Many applications will use these two functions only in the combination

```
subroutine circe2 (p1, h1, p2, h2, x1, x2, rng)
integer p1, h1, p2, h2
double precision x1, x2
external rng
call cir2ch (p1, h1, p2, h2, rng)
call cir2gn (p1, h1, p2, h2, x1, x2, rng)
end
```

after which randomly distributed `p1`, `h1`, `p2`, `h2`, `x1`, and `x2` are available for further processing.

NB: a function like `circe2` has not been added to the default FORTRAN90 API, because `cir2gn` and `circe2` have the same number and types of arguments, differing only in the input/output direction of four of the arguments. This is a source of errors that a FORTRAN90 compiler can not help the application programmer to spot. The current design should be less error prone and is only minimally less convenient because of the additional procedure call

```
      integer p1, h1, p2, h2
      double precision x1, x2
      integer n, nevent
      external rng
      ...
      do 10 n = 1, nevent
        call cir2ch (p1, h1, p2, h2, rng)
        call cir2gn (p1, h1, p2, h2, x1, x2, rng)
        ...
10    continue
      ...
```

Implementations in more modern programming languages (Fortran90/95, C++, Java, O'Caml, etc.) can and will provide a richer API with reduced name space pollution and danger of confusion.

3.3.1 Extensions: General Polarizations

Given a pair of flavors, triples (x_1, x_2, ρ) of momentum fractions together with density matrices for the polarizations distributed according to the cur-

rently loaded beam descriptions can be obtained by repeatedly calling `cir2gp` (*GeneratePolarized*):

```
subroutine cir2gp (p1, p2, x1, x2, pol, rng)
```

`integer p1` (input): particle code for the first particle

`integer p2` (input): particle code for the second particle

`double precision x1` (output): fraction of the beam energy carried by the first particle

`double precision x2` (output): fraction of the beam energy carried by the second particle

`double precision pol(0:3,0:3)` (output): the joint density matrix of the two polarizations is parametrized by a real 4×4-matrix

$$\rho(\chi) = \sum_{a,a'=0}^{3} \frac{\chi_{aa'}}{4}\, \sigma_a \otimes \sigma_{a'} \tag{14}$$

using the notation $\sigma_0 = 1$. We have `pol(0,0)` $= 1$ since $\operatorname{tr} \rho = 1$.

`external rng`: subroutine generating a uniform deviate

This procedure has not been implemented in version 2.0 and will be provided in release 2.1.

3.4 Distributions

The normalized luminosity density $D_{p_1p_2}(x_1, x_2)$ for the given flavor and helicity combination for the currently loaded beam description satisfies

$$\int \mathrm{d}x_1 \wedge \mathrm{d}x_2\, D_{p_1p_2}(x_1, x_2) = 1 \tag{15}$$

and is calculated by `cir2dn` *(DistributioN)*:

```
double precision function cir2dn (p1, h1, p2, h2, x1, x2)
```

`integer p1` (input): particle code for the first particle

`integer h1` (input): helicity of the first particle

`integer p2` (input): particle code for the second particle

`integer h2` (input): helicity of the second particle

`double precision x1` (input): fraction of the beam energy carried by the first particle

`double precision x2` (input): fraction of the beam energy carried by the second particle

If any of the helicities is 0 and the loaded beam description is not summed over polarizations, the result is *not* the polarization summed distribution and 0 is returned instead. Application programs must either sum by themselves or load a more efficient abbreviated beam description.

`Circe1` users should take note that the densities are now normalized *individually* and no longer relative to a master e^+e^- distribution. Users of `Circe1` should also take note that the special treatment of δ-distributions at the endpoints has been removed. The corresponding contributions have been included in small bins close to the endpoints. For small enough bins, this approach is sufficiently accurate and avoids the pitfalls of the approach of `Circe1`.

Applications that convolute the `Circe2` distributions with other distributions can benefit from accessing the map employed by `Circe2` internally through `cir2mp` (*MaP*):

`subroutine cir2mp (p1, h1, p2, h2, x1, x2, m1, m2, d)`

`integer p1` (input): particle code for the first particle

`integer h1` (input): helicity of the first particle

`integer p2` (input): particle code for the second particle

`integer h2` (input): helicity of the second particle

`double precision x1` (input): fraction of the beam energy carried by the first particle

`double precision x2` (input): fraction of the beam energy carried by the second particle

`integer m1` (output): map

`integer m2` (output): map

`double precision d` (output):

3.4.1 Extensions: General Polarizations

The product of the normalized luminosity density $D_{p_1p_2}(x_1, x_2)$ and the joint polarization density mattrix for the given flavor and helicity combination for the currently loaded beam description is calculated by `cir2dm` (*DensityMatrices*):

`double precision function cir2dm (p1, p2, x1, x2, pol)`

`integer p1` (input): particle code for the first particle

`integer p2` (input): particle code for the second particle

`double precision x1` (input): fraction of the beam energy carried by the first particle

`double precision x2` (input): fraction of the beam energy carried by the second particle

`double precision pol(0:3,0:3)` (output): the joint density matrix multiplied by the normalized probability density. The density matrix is parametrized by a real 4×4-matrix

$$D_{p_1p_2}(x_1, x_2) \cdot \rho(\chi) = \sum_{a,a'=0}^{3} \frac{1}{4} \chi_{p_1p_2,aa'}(x_1, x_2)\, \sigma_a \otimes \sigma_{a'} \quad (16)$$

using the notation $\sigma_0 = 1$. We have `pol(0,0)` $= D_{p_1p_2}(x_1, x_2)$ since $\operatorname{tr} \rho = 1$.

This procedure has not been implemented in version 2.0 and will be provided in release 2.1.

3.5 Private Parts

The following need not concern application programmer, except that there must be no clash with any other global name in the application program:

`common /cir2cm/`: the internal data store for `Circe2`, which *must not* be accessed by application programs.

3.6 Optional API

The following is part of a separate library that can be loaded optionally. The beam description can be loaded from the `block data` segment `cir2bd`by `cir2lb` (*LoadBlockdata*):

`subroutine cir2lb (design, roots, ierror)`

`common /cir2cd/`: an optional and very big internal data store for `Circe2`, which *must not* be accessed by application programs.

`block data cir2bd`: data for `/cir2bd/`

This procedure has not been implemented yet and will be provided in a future release.

4 Examples

In this section, we collect some simple yet complete examples using the API described in section 3. In all examples, the role of the physics application is played by a `write` statement, which would be replaced by an appropriate event generator for hard scattering physics or background events. The examples assume the existence of either a file `default.circe` describing polarized $\sqrt{s} = 500\,\mathrm{GeV}$ beams or an abbreviated file `default_polavg.circe` where the helicities are summed over.

4.1 Unweighted Event Generation

`Circe2` has been designed for the efficient generation of unweighted events, i. e. event samples that are distributed according to the given probability density. Examples of weighted events are discussed in section 4.2 below.

4.1.1 Mixed Flavors and Helicities

The most straightforward application uses a stream of events with a mixture of flavors and helicities in *random* order. If the application can consume events without the need for costly reinitializations when the flavors are changed, a simple loop around `cir2ch` and `cir2gn` suffices:

```
program demo1
  use kinds
  use circe2
  implicit none
  integer :: p1, h1, p2, h2, n, nevent, ierror
  real(kind=double) :: x1, x2
  external random
  nevent = 20
  ierror = 1
  call cir2ld ('default.circe', '*', 500D0, ierror)
  if (ierror .lt. 0) stop
  write (*, '(A7,4(1X,A4),2(1X,A10))') &
       '#', 'pdg1', 'hel1', 'pdg2', 'hel2', 'x1', 'x2'
  do n = 1, nevent
    call cir2ch (p1, h1, p2, h2, random)
    call cir2gn (p1, h1, p2, h2, x1, x2, random)
    write (*, '(I7,4(1X,I4),2(1X,F10.8))') n, p1, h1, p2, h2, x1, x2
  end do
end program demo1
```

The following minimalistic linear congruential random number generator can be used for demonstrating the interface, but it is known to produce correlations and *must* be replaced by a more sophisticated one in real applications:

```
subroutine random (r)
  use kinds
  real(kind=double) :: r
  integer, parameter :: M = 259200, A = 7141, C = 54773
  integer, save :: n = 0
  n = mod (n*A + C, M)
  r = real (n, kind=double) / real (M, kind=double)
end subroutine random
```

4.1.2 Separated Flavors and Helicities

If the application can not switch efficiently among flavors and helicities, another approach is more useful. It walks through the flavors and helicities sequentially and uses the partial luminosities `cir2lm` to determine the correct number of events for each combination:

```
program demo2
  use kinds
  use circe2
  implicit none
  integer :: i1, i2, h1, h2, i, n, nevent, nev, ierror
  integer, dimension(3), save :: pdg = (/ 22, 11, -11 /)
  real(kind=double) :: x1, x2, lumi
  external random
  nevent = 20
  ierror = 1
  call cir2ld ('default.circe', '*', 500D0, ierror)
  if (ierror .lt. 0) stop
  lumi = cir2lm (0, 0, 0, 0)
  write (*, '(A7,4(1X,A4),2(1X,A10))') &
      '#', 'pdg1', 'hel1', 'pdg2', 'hel2', 'x1', 'x2'
  i = 0
  do i1 = 1, 3
    do i2 = 1, 3
      do h1 = -1, 1, 2
        do h2 = -1, 1, 2
          nev = nevent * cir2lm (pdg(i1), h1, pdg(i2), h2) / lumi
          do n = 1, nev
            call cir2gn (pdg(i1), h1, pdg(i2), h2, x1, x2, random)
            i = i + 1
```

```
            write (*, '(I7,4(1X,I4),2(1X,F10.8))') &
                 i, pdg(i1), h1, pdg(i2), h2, x1, x2
          end do
        end do
      end do
    end do
  end do
end program demo2
```

More care can be taken to guarantee that the total number of events is not reduced by rounding `new` towards 0, but the error will be negligible for reasonably high statistics anyway.

4.1.3 Polarization Averaged

If the helicities are to be ignored, the abbreviated file `default_polavg.circe` can be read. The code remains unchanged, but the variables `h1` and `h2` will always be set to 0.

```
program demo3
  use kinds
  use circe2
  implicit none
  integer :: p1, h1, p2, h2, n, nevent, ierror
  real(kind=double) :: x1, x2
  external random
  nevent = 20
  ierror = 1
  call cir2ld ('default_polavg.circe', '*', 500D0, ierror)
  if (ierror .lt. 0) stop
  write (*, '(A7,2(1X,A4),2(1X,A10))') &
       '#', 'pdg1', 'pdg2', 'x1', 'x2'
  do n = 1, nevent
    call cir2ch (p1, h1, p2, h2, random)
    call cir2gn (p1, h1, p2, h2, x1, x2, random)
    write (*, '(I7,2(1X,I4),2(1X,F10.8))') n, p1, p2, x1, x2
  end do
end program demo3
```

4.1.4 Flavors and Helicity Projections

There are three ways to produce samples with a fixed subset of flavors or helicities. As an example, we generate a sample of two photon events with $L = 0$. The first approach generates the two channels $++$ and $--$ sequentially:

```
program demo4
  use kinds
  use circe2
  implicit none
  real(kind=double) :: x1, x2, lumipp, lumimm
  integer :: n, nevent, npp, nmm, ierror
  external random
  nevent = 20
  ierror = 1
  call cir2ld ('default.circe', '*', 500D0, ierror)
  if (ierror .lt. 0) stop
  lumipp = cir2lm (22,  1, 22,  1)
  lumimm = cir2lm (22, -1, 22, -1)
  npp = nevent * lumipp / (lumipp + lumimm)
  nmm = nevent - npp
  write (*, '(A7,2(1X,A10))') '#', 'x1', 'x2'
  do n = 1, npp
    call cir2gn (22,  1, 22,  1, x1, x2, random)
    write (*, '(I7,2(1X,F10.8))') n, x1, x2
  end do
  do n = 1, nmm
    call cir2gn (22, -1, 22, -1, x1, x2, random)
    write (*, '(I7,2(1X,F10.8))') n, x1, x2
  end do
end program demo4
```

a second approach alternates between the two possibilities

```
program demo5
  use kinds
  use circe2
  implicit none
  real(kind=double) :: x1, x2, u, lumipp, lumimm
  integer :: n, nevent, ierror
  external random
  nevent = 20
  ierror = 1
  call cir2ld ('default.circe', '*', 500D0, ierror)
  if (ierror .lt. 0) stop
  lumipp = cir2lm (22,  1, 22,  1)
  lumimm = cir2lm (22, -1, 22, -1)
  write (*, '(A7,2(1X,A10))') '#', 'x1', 'x2'
  do n = 1, nevent
    call random (u)
```

```
    if (u * (lumipp + lumimm) .lt. lumipp) then
      call cir2gn (22,  1, 22,  1, x1, x2, random)
    else
      call cir2gn (22, -1, 22, -1, x1, x2, random)
    endif
    write (*, '(I7,2(1X,F10.8))') n, x1, x2
  end do
end program demo5
```

finally, the third approach uses rejection to select the desired flavors and helicities

```
program demo6
  use kinds
  use circe2
  implicit none
  integer :: p1, h1, p2, h2, n, nevent, ierror
  real(kind=double) :: x1, x2
  external random
  nevent = 20
  ierror = 1
  call cir2ld ('default.circe', '*', 500D0, ierror)
  if (ierror .lt. 0) stop
  write (*, '(A7,2(1X,A10))') '#', 'x1', 'x2'
  n = 0
  do
    call cir2ch (p1, h1, p2, h2, random)
    call cir2gn (p1, h1, p2, h2, x1, x2, random)
    if ((p1 .eq. 22) .and. (p2 .eq. 22) .and.  &
       (((h1 .eq.  1) .and. (h2 .eq.  1)) .or. &
       ((h1 .eq. -1) .and. (h2 .eq. -1)))) then
      n = n + 1
      write (*, '(I7,2(1X,F10.8))') n, x1, x2
    end if
    if (n .ge. nevent) exit
  end do
end program demo6
```

All generated distributions are equivalent, but the chosen subsequences of random numbers will be different. It depends on the application and the channels under consideration, which approach is the most appropriate.

4.2 Distributions and Weighted Event Generation

If no events are to be generated, `cir2dn` can be used to calculate the probability density $D(x_1, x_2)$ at a given point. This can be used for numerical integration other than Monte Carlo or for importance sampling in the case that the distribution to be folded with D is more rapidly varying than D itself.

Depending on the beam descriptions, these distributions are available either for fixed helicities

```
program demo7
  use kinds
  use circe2
  implicit none
  integer :: n, nevent, ierror
  real(kind=double) :: x1, x2, w
  nevent = 20
  ierror = 1
  call cir2ld ('default.circe', '*', 500D0, ierror)
  if (ierror .lt. 0) stop
  write (*, '(A7,3(1X,A10))') '#', 'x1', 'x2', 'weight'
  do n = 1, nevent
    call random (x1)
    call random (x2)
    w = cir2dn (22, 1, 22, 1, x1, x2)
    write (*, '(I7,2(1X,F10.8),1X,E10.4)') n, x1, x2, w
  end do
end program demo7
```

or summed over all helicities if the beam description is polarization averaged:

```
program demo8
  use kinds
  use circe2
  implicit none
  integer n, nevent, ierror
  real(kind=double) :: x1, x2, w
  nevent = 20
  ierror = 1
  call cir2ld ('default_polavg.circe', '*', 500D0, ierror)
  if (ierror .lt. 0) stop
  write (*, '(A7,3(1X,A10))') '#', 'x1', 'x2', 'weight'
  do n = 1, nevent
    call random (x1)
```

```
    call random (x2)
    w = cir2dn (22, 0, 22, 0, x1, x2)
    write (*, '(I7,2(1X,F10.8),1X,E10.4)') n, x1, x2, w
  end do
end program demo8
```

If the beam description is not polarization averaged, the application can perform the averaging itself (note that each distribution is normalized):

```
program demo9
  use kinds
  use circe2
  implicit none
  integer :: n, nevent, ierror
  real(kind=double) :: x1, x2, w
  real(kind=double) :: lumi, lumipp, lumimp, lumipm, lumimm
  nevent = 20
  ierror = 1
  call cir2ld ('default.circe', '*', 500D0, ierror)
  if (ierror .lt. 0) stop
  lumipp = cir2lm (22,  1, 22,  1)
  lumipm = cir2lm (22,  1, 22, -1)
  lumimp = cir2lm (22, -1, 22,  1)
  lumimm = cir2lm (22, -1, 22, -1)
  lumi = lumipp + lumimp + lumipm + lumimm
  write (*, '(A7,3(1X,A10))') '#', 'x1', 'x2', 'weight'
  do n = 1, nevent
    call random (x1)
    call random (x2)
    w = (   lumipp * cir2dn (22,  1, 22,  1, x1, x2) &
         + lumipm * cir2dn (22,  1, 22, -1, x1, x2)  &
         + lumimp * cir2dn (22, -1, 22,  1, x1, x2)  &
         + lumimm * cir2dn (22, -1, 22, -1, x1, x2)) / lumi
    write (*, '(I7,2(1X,F10.8),1X,E10.4)') n, x1, x2, w
  end do
end program demo9
```

The results produced by the preceeding pair of examples will differ point-by-point, because the polarized and the polarization summed distribution will be binned differently. However, all histograms of the results with reasonable bin sizes will agree.

4.3 Scans and Interpolations

Currently there is no supported mechanism for interpolating among distributions for the discrete parameter sets. The most useful application of such a facility would be a scan of the energy dependence of an observable

$$\mathcal{O}(s) = \int \mathrm{d}x_1 \mathrm{d}x_2 \mathrm{d}\Omega \, D(x_1, x_2, s) \frac{\mathrm{d}\sigma}{\mathrm{d}\Omega}(x_1, x_2, s, \Omega) O(x_1, x_2, s, \Omega) \tag{17a}$$

which has to take into account the s-dependence of the distribution $D(x_1, x_2, s)$. Full simulations of the beam dynamics for each value of s are too costly and `Circe1` [1] supported linear interpolation

$$\bar{D}(x_1, x_2, s) = \frac{(s - s_-)D(x_1, x_2, s_+) + (s_+ - s)D(x_1, x_2, s_-)}{s_+ - s_-} \tag{17b}$$

as an economical compromise. However, since $\mathcal{O}$ in (17) is a strictly *linear* functional of D, it is mathematically equivalent to interpolating $\mathcal{O}$ itself

$$\bar{\mathcal{O}}(s) = \frac{(s - s_-)\tilde{\mathcal{O}}(s, s_+) + (s_+ - s)\tilde{\mathcal{O}}(s, s_-)}{s_+ - s_-} \tag{18a}$$

where

$$\tilde{\mathcal{O}}(s, s_0) = \int \mathrm{d}x_1 \mathrm{d}x_2 \mathrm{d}\Omega \, D(x_1, x_2, s_0) \frac{\mathrm{d}\sigma}{\mathrm{d}\Omega}(x_1, x_2, s, \Omega) O(x_1, x_2, s, \Omega) \tag{18b}$$

Of course, evaluating the two integrals in (18) with comparable accuracy demands four times the calculational effort of the single integral in (17). Therefore, if overwhelming demand arises, support for (17) can be reinstated, but at the price of a considerably more involved API for loading distributions.

5 Algorithms

`Circe2` attempts to recover a probability density $w(x_1, x_2)$ from a finite set of triples $\{(x_{1,i}, x_{2,i}, w_i)\}|_{i=1,\ldots,N}$ that are known to be distributed according to $w(x_1, x_2)$. This recovery should introduce as little bias as possible. The solution should provide a computable form of $w(x_1, x_2)$ as well as a procedure for generating more sets of triples $\{(x_{1,i}, x_{2,i}, w_i)\}$ with "the same" distribution.

The discrete distribution

$$\hat{w}(x_1, w_2) = \sum_i w_i \delta(x_1 - x_{1,i}) \delta(x_2 - x_{2,i}) \tag{19}$$

adds no bias, but is obviously not an adequate solution of the problem, because it depends qualitatively on the sample. While the sought after distribution may contain singularities, their number and the dimension of their support must not depend on the sample size. There is, of course, no unique solution to this problem and we must allow some prejudices to enter in order to single out the most adequate solution.

The method employed by `Circe1` was to select a family of analytical distributions that are satisfy reasonable criteria suggested by physics [1] and select representatives by fitting the parameters of these distributions. This has been unreasonably successful for modelling the general properties, but must fail eventually if finer details are studied. Enlarging the families is theoretically possible but empirically it turns out that the number of free parameters grows faster than the descriptive power of the families.

Another approach is to forego functions that are defined globally by an analytical expression and to perform interpolation of binned samples, requiring continuity of the distribution and their derivatives. Again, this fails in practice, this time because such interpolations tend to create wild fluctuations for statistically distributed data and the resulting distributions will often violate basic conditions like positivity.

Any attempt to recover the distributions that uses local properties will have to bin the data

$$N_i = \int_{\Delta_i} \mathrm{d}x\, w(x) \tag{20}$$

with

$$\Delta_i \cap \Delta_j = \emptyset \quad (i \neq j), \qquad \bigcup_i \Delta_i = [0,1] \times [0,1] \tag{21}$$

Therefore it appears to be eminently reasonable to approximate w by a piecewise constant

$$\hat{w}(x) = \sum_i \frac{N_i}{|\Delta_i|} \Theta(x \in \Delta_i)\,. \tag{22}$$

However, this procedure alse introduces a bias and if the number of bins is to remain finite, this bias cannot be removed.

Nevertheless, one can tune this bias to the problem under study and obtain better approximations by making use of the well known fact that probability distributions are not invariant under coordinate transformations, as described in section **??** below.

5.1 Histograms

The obvious approach to histogramming is to cover the unit square $[0,1] \times [0,1]$ uniformly with n_b^2 squares, but this approach is not economical in its

use of storage. For example, high energy physics studies at a $\sqrt{s} = 500\,\mathrm{GeV}$ LC will require an energy resolution of better than $1\,\mathrm{GeV}$ and we should bin each beam in steps of $500\,\mathrm{MeV}$, i. e. $n_b = 500$. This results in a two dimensional histogram of $500^2 = 25000$ bins for each combination of flavor and helicity. Using non-portable binary storage, this amounts to 100 KB for typical single precision floating point numbers and 200 KB for typical double precision floating point numbers.

Obviously, binary storage is not a useful exchange format and we have to use an ASCII exchange format, which in its human readable form uses 14 bytes for single precision and 22 bytes for double precision and the above estimates have to be changed to 350 KB and 550 KB respectively. We have four flavor combinations if pair creation is ignored and nine flavor combinations if it is taken into account. For each flavor combination there are four helicity combinations and we arrive at 16 or 36 combinations.

Altogether, a fixed bin histogram requires up to 20 MB of data for *each* accelerator design at *each* energy step for a mere 1% energy resolution. While this could be handled with modern hardware, we have to keep in mind that the storage requirements grow quadratically with the resolution and that several generations of designs should be kept available for comparison studies.

For background studies, low energy tails down to the pair production threshold $m_e = 511\,\mathrm{KeV} \approx 10^{-6} \cdot \sqrt{s}$ have to be described correctly. Obviously, fixed bin histograms are not an option at all in this case.

- mention 2-D Delauney triangulations here
- mention Stazsek's FOAM [14] here
- praise VEGAS/VAMP

5.2 Coordinate Dependence of Sampling Distributions

The contents of this section is well known to all practitioners and is repeated only for establishing notation. For any sufficiently smooth (piecewise differentiable suffices) map

$$\begin{aligned} \phi : D_x &\to D_y \\ x &\mapsto y = \phi(x) \end{aligned} \tag{23}$$

integrals of distribution functions $w : D_y \to \mathbf{R}$ are invariant, as long as we apply the correct Jacobian factor

$$\int_{D_y} \mathrm{d}y\, w(y) = \int_{D_x} \mathrm{d}x\, \frac{\mathrm{d}\phi}{\mathrm{d}x} \cdot (w \circ \phi)(x) = \int_{D_x} \mathrm{d}x\, w^\phi(x) \tag{24a}$$

where

$$w^{\phi}(x) = (w \circ \phi)(x) \cdot \frac{\mathrm{d}\phi}{\mathrm{d}x}(x) = \frac{(w \circ \phi)(x)}{\left(\dfrac{\mathrm{d}\phi^{-1}}{\mathrm{d}y} \circ \phi\right)(x)} \tag{24b}$$

The fraction can be thought of as being defined by the product, if the map ϕ is not invertible. Below, we will always deal with invertible maps and the fraction is more suggestive for our purposes. Therefore, ϕ induces a pull-back map ϕ^* on the space of integrable functions

$$\begin{aligned} \phi^* : L_1(D_y, \mathbf{R}) &\to L_1(D_x, \mathbf{R}) \\ w &\mapsto w^{\phi} = \frac{w \circ \phi}{\left(\dfrac{\mathrm{d}\phi^{-1}}{\mathrm{d}y} \circ \phi\right)} \end{aligned} \tag{25}$$

If we find a map ϕ_w with $\mathrm{d}\phi^{-1}/\mathrm{d}y \sim w$, then sampling the transformed weight w^{ϕ_w} will be very stable, even if sampling the original weight w is not.

On the other hand, the inverse map

$$\begin{aligned} (\phi^*)^{-1} : L_1(D_x, \mathbf{R}) &\to L_1(D_y, \mathbf{R}) \\ w &\mapsto w^{(\phi^{-1})} = \left(\frac{\mathrm{d}\phi^{-1}}{\mathrm{d}y}\right) \cdot (w \circ \phi^{-1}) \end{aligned} \tag{26}$$

with $(\phi^{-1})^* = (\phi^*)^{-1}$ can be used to transform a uniform distribution into the potentially much more interesting $\mathrm{d}\phi^{-1}/\mathrm{d}y$.

5.3 Sampling Distributions With Integrable Singularities

A typical example appearing in `Circe1`

$$\int^1 \mathrm{d}x\, w(x) \approx \int^1 \mathrm{d}x\, (1-x)^{\beta} \tag{27}$$

converges for $\beta > -1$, while the variance

$$\int^1 \mathrm{d}x\, (w(x))^2 \approx \int^1 \mathrm{d}x\, (1-x)^{2\beta} \tag{28}$$

does not converge for $\beta \leq -1/2$. Indeed, this case is the typical case for realistic beamstrahlung spectra at $\mathrm{e}^+\mathrm{e}^-$ LCs and has to covered.

Attempting a naive VEGAS/VAMP adaption fails, because the *nonintegrable* variance density acts as a sink for bins, even though the density itself is integrable.

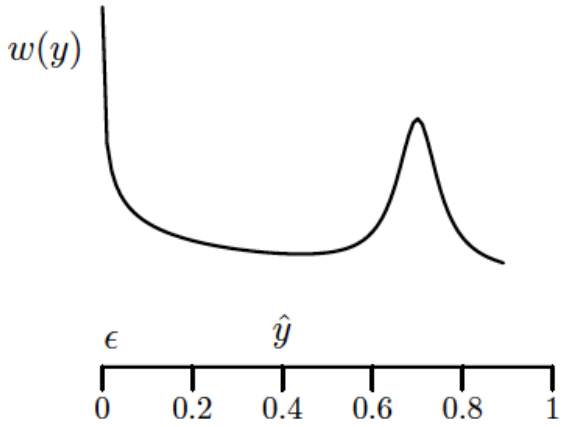

Figure 2: Distribution with both an integrable singularity $\propto x^{-0.2}$ and a peak at finite $x \approx 0.7$.

- examples show that moments of distributions are reproduced *much* better after mapping, even if histograms look indistinguishable.
- biasing doesn't appear to work as well as fences

The distributions that we want to describe can contain integrable singularities and δ-distributions at the endpoints. Since there is always a finite resolution, both contributions can be handled by a finite binsize at the endpoints. However, we can expect to improve the convergence of the grid adaption in neighborhoods of the singularities by canceling the singularities with the Jacobian of a power map. Also the description of the distribution *inside* each bin will be improved for reasonable maps.

5.4 Piecewise Differentiable Maps

blah, blah, blah

Ansatz:

$$\begin{aligned} \Phi_{\{\phi\}} : [X_0, X_1] &\to [Y_0, Y_1] \\ x \mapsto \Phi_{\{\phi\}}(x) &= \sum_{i=1}^{n} \Theta(x_i - x)\Theta(x - x_{i-1})\phi(x) \end{aligned} \tag{29}$$

with $x_0 = X_0$, $x_n = X_1$ and $x_i > x_{i-1}$. In each interval

$$\begin{aligned} \phi_i : [x_{i-1}, x_i] &\to [y_{i-1}, y_i] \\ x &\mapsto y = \phi_i(x) \end{aligned} \tag{30}$$

with $y_0 = Y_0$, $y_n = Y_1$

5.4.1 Powers

 integrable singularities

$$\begin{aligned} \psi_{a_i,b_i}^{\alpha_i,\xi_i,\eta_i} &: [x_{i-1}, x_i] \to [y_{i-1}, y_i] \\ x &\mapsto \psi_{a_i,b_i}^{\alpha_i,\xi_i,\eta_i}(x) = \frac{1}{b_i}(a_i(x-\xi_i))^{\alpha_i} + \eta_i \end{aligned} \tag{31}$$

We assume $\alpha_i \neq 0$, $a_i \neq 0$ and $b_i \neq 0$. Note that $\psi_{a,b}^{\alpha,\xi,\eta}$ encompasses both typical cases for integrable endpoint singularities $x \in [0,1]$:

$$\psi_{1,1}^{\alpha,0,0}(x) = x^\alpha \tag{32a}$$

$$\psi_{-1,-1}^{\alpha,1,1}(x) = 1 - (1-x)^\alpha \tag{32b}$$

The inverse maps are

$$\begin{aligned} (\psi_{a_i,b_i}^{\alpha_i,\xi_i,\eta_i})^{-1} &: [y_{i-1}, y_i] \to [x_{i-1}, x_i] \\ y &\mapsto (\psi_{a_i,b_i}^{\alpha_i,\xi_i,\eta_i})^{-1}(y) = \frac{1}{a_i}(b_i(y-\eta_i))^{1/\alpha_i} + \xi_i \end{aligned} \tag{33}$$

and incidentally:

$$(\psi_{a,b}^{\alpha,\xi,\eta})^{-1} = \psi_{b,a}^{1/\alpha,\eta,\xi} \tag{34}$$

The Jacobians are

$$\frac{\mathrm{d}y}{\mathrm{d}x}(x) = \frac{a\alpha}{b}(a(x-\xi))^{\alpha-1} \tag{35a}$$

$$\frac{\mathrm{d}x}{\mathrm{d}y}(y) = \frac{b}{a\alpha}(b(y-\eta))^{1/\alpha-1} \tag{35b}$$

and satisfy, of course,

$$\frac{\mathrm{d}x}{\mathrm{d}y}(y(x)) = \frac{1}{\dfrac{\mathrm{d}y}{\mathrm{d}x}(x)} \tag{36}$$

In order to get a strictly monotonous function, we require

$$\frac{a\alpha}{b} > 0 \tag{37}$$

Since we will see below that almost always in practical applications $\alpha > 0$, this means $\epsilon(a) = \epsilon(b)$.

From (25) and (35b), we see that this map is useful for handling weights[2]

$$w(y) \propto (y-\eta)^{\beta} \tag{38}$$

for $\beta > -1$, if we choose $\beta - (1/\alpha - 1) \geq 0$, i. e. $\alpha \gtrsim 1/(1+\beta)$.

The five parameters $(\alpha, \xi, \eta, a, b)$ are partially redundant. Indeed, there is a one parameter semigroup of transformations

$$(\alpha, \xi, \eta, a, b) \to (\alpha, \xi, \eta, at, bt^{\alpha}), \qquad (t > 0) \tag{39}$$

that leaves $\psi_{a,b}^{\alpha,\xi,\eta}$ invariant:

$$\psi_{a,b}^{\alpha,\xi,\eta} = \psi_{at,bt^{\alpha}}^{\alpha,\xi,\eta} \tag{40}$$

Assuming that multiplications are more efficient than sign transfers, the redundant representation is advantageous. Unless sign transfers are implemented directly in hardware, they involve a branch in the code and the assumption appears to be reasonable.

5.4.2 Identity

The identity map

$$\begin{aligned} \iota : [x_{i-1}, x_i] &\to [y_{i-1}, y_i] = [x_{i-1}, x_i] \\ x &\mapsto \iota(x) = x \end{aligned} \tag{41}$$

is a special case of the power map $\iota = \psi_{1,1}^{1,0,0}$, but, for efficiency, it is useful to provide a dedicated "implementation" anyway.

5.4.3 Resonances

- not really needed in the applications so far, because the variance remains integrable.
- no clear example for significantly reduced numbers of bins for the same qualitaty with mapping.
- added for illustration.

[2]The limiting case $(y-\eta)^{-1}$ could be covered by maps $x \mapsto \mathrm{e}^{a(x-\xi)}/b + \eta$, where the non-integrability of the density is reflected in the fact that the domain of the map is semi-infinite (i. e. $x \to -\epsilon(a) \cdot \infty$). In physical applications, the densities are usually integrable and we do not consider this case in the following.

$$\begin{aligned}\rho^{\xi_i,\eta_i}_{a_i,b_i} &: [x_{i-1}, x_i] \to [y_{i-1}, y_i] \\ x &\mapsto \rho^{\xi_i,\eta_i}_{a_i,b_i}(x) = a_i \tan\left(\frac{a_i}{b_i^2}(x-\xi_i)\right) + \eta_i\end{aligned} \tag{42}$$

Inverse

$$\begin{aligned}(\rho^{\xi_i,\eta_i}_{a_i,b_i})^{-1} &: [y_{i-1}, y_i] \to [x_{i-1}, x_i] \\ y &\mapsto (\rho^{\xi_i,\eta_i}_{a_i,b_i})^{-1}(y) = \frac{b_i^2}{a_i}\operatorname{atan}\left(\frac{y-\eta_i}{a_i}\right) + \xi_i\end{aligned} \tag{43}$$

is useful for mapping *known* peaks, since

$$\frac{\mathrm{d}\phi^{-1}}{\mathrm{d}y}(y) = \frac{\mathrm{d}x}{\mathrm{d}y}(y) = \frac{b^2}{(y-\eta)^2 + a^2} \tag{44}$$

5.4.4 Patching Up

Given a collection of intervals with associated maps, it remains to construct a combined map. Since *any* two intervals can be mapped onto each other by a map with constant Jacobian, we have a "gauge" freedom and must treat x_{i-1} and x_i as free parameters in

$$\psi^{\alpha_i,\xi_i,\eta_i}_{a_i,b_i} : [x_{i-1}, x_i] \to [y_{i-1}, y_i] \tag{45}$$

i. e.

$$x_j = (\psi^{\alpha_i,\xi_i,\eta_i}_{a_i,b_i})^{-1}(y_j) = \frac{1}{a_i}(b_i(y_j - \eta_i))^{1/\alpha_i} + \xi_i \qquad \text{for } j \in \{i-1, i\} \tag{46}$$

Since α and η denote the strength and the location of the singularity, respectively, they are the relevant input parameters and we must solve the constraints (46) for ξ_i, a_i and b_i. Indeed a family of solutions is

$$a_i = \frac{(b_i(y_i - \eta_i))^{1/\alpha_i} - (b_i(y_{i-1} - \eta_i))^{1/\alpha_i}}{x_i - x_{i-1}} \tag{47a}$$

$$\xi_i = \frac{x_{i-1}|y_i - \eta_i|^{1/\alpha_i} - x_i|y_{i-1} - \eta_i|^{1/\alpha_i}}{|y_i - \eta_i|^{1/\alpha_i} - |y_{i-1} - \eta_i|^{1/\alpha_i}} \tag{47b}$$

which is unique up to (39). The degeneracy (39) can finally be resolved by demanding $|b| = 1$ in (47a).

It remains to perform a 'gauge fixing' and choose the domains $[x_{i-1}, x_i]$. The minimal solution is $x_i = y_i$ for all i, which maps the boundaries between different mappings onto themselves and we need only to store either $\{x_0, x_1, \ldots, x_n\}$ or $\{y_0, y_1, \ldots, y_n\}$.

For the resonance map

$$x_j = (\rho_{a_i,b_i}^{\xi_i,\eta_i})^{-1}(y_j) = \frac{b_i^2}{a_i}\mathrm{atan}\left(\frac{y_j-\eta_i}{a_i}\right) + \xi_i \qquad \text{for } j \in \{i-1, i\} \tag{48}$$

i. e.

$$b_i = \sqrt{a_i \frac{x_i - x_{i-1}}{\mathrm{atan}\left(\frac{y_i-\eta_i}{a_i}\right) - \mathrm{atan}\left(\frac{y_{i-1}-\eta_i}{a_i}\right)}} \tag{49a}$$

$$\xi_i = \frac{x_{i-1}\mathrm{atan}\left(\frac{y_i-\eta_i}{a_i}\right) - x_i\mathrm{atan}\left(\frac{y_{i-1}-\eta_i}{a_i}\right)}{x_i - x_{i-1}} \tag{49b}$$

as a function of the physical peak location η and width a.

6 Preparing Beam Descriptions with `circe2_tool`


rationale

6.1 `circe2_tool` Files


{ and }

6.1.1 Per File Options

`file`: a double quote delimited string denoting the name of the output file that will be read by `cir2ld` (in the format described in table 2).

6.1.2 Per Design Options

`design`: a double quote delimited string denoting a name for the design. See the description of `cir2ld` on page 9 for conventions for these names.

`roots`: $\sqrt{s}$/GeV of the accelerator design.

`bins`: number of bins for the histograms in both directions. `bins/1` and `bins/2` apply only to x_1 and x_2 respectively. This number can be overwritten by channel options.

`comment`: a double quote delimited string denoting a one line comment that will be copied to the output file. This command can be repeated.

6.1.3 Per Channel Options

If an option can apply to either beam or both, it can be qualified by `/1` or `/2`. For example, `bins` applies to both beams, while `bins/1` and `bins/2` apply only to x_1 and x_2 respectively.

`bins`: number of bins for the histograms. These overwrite the per-design option.

`pid`: particle identification: either a PDG code [11] (see page 3) or one of `gamma`, `photon`, `electron`, `positron`.

`pol`: polarization: one of $\{-1, 0, 1\}$, where 0 means unpolarized (see page 3).

`min`: minimum value of the coordinate(s). The default is 0.

`max`: maximum value of the coordinate(s). The default is 1.

`fix`

`free`

`map`: apply a map to a subinterval. Currently, three types of maps are supported:

`id { `n `[`$x_{\min}$`,`$x_{\max}$`] }`: apply an identity map in the interval $[x_{\min}, x_{\max}]$ subdivided into n bins. The non-trivial effect of this map is that the endpoints $x_{\min}$ and $x_{\max}$ are frozen.

`power {`n `[`$x_{\min}$`,`$x_{\max}$`] beta =` β `eta =` η `}`: apply a power map in the interval $[x_{\min}, x_{\max}]$ subdivided into n bins. $\alpha = 1/(1+\beta)$, such that an integrable singularity at η with power β is mapped away. This is the most important map in practical applications and manual fine tuning is rewarded.

`resonance {`n `[`$x_{\min}$`,`$x_{\max}$`] center =` η `width =` a `}`: apply a resonance map in the interval $[x_{\min}, x_{\max}]$ subdivided into n bins. This map is hardly ever needed, since VEGAS/VAMP appears to be able to handle non-singular peaks very well.

`triangle`

`notriangle`

`lumi`: luminosity of the beam design, it units of

$$\mathrm{fb}^{-1}\upsilon^{-1} = 10^{32}\mathrm{cm}^{-2}\mathrm{sec}^{-1} \tag{50}$$

where $\upsilon = 10^7\,\mathrm{sec} \approx \mathrm{year}/\pi$ is an "effective year" of running with about 30% up-time

`events`: a double quote delimited string denoting the name of the input file.

`ascii`: input file contains formatted ASCII numbers.

`binary`: input file is in raw binary format that can be accessed by fast memory mapped I/O. Such files are not portable and must not contain Fortran record markers.

`columns`: number of columns in a binary file.

`iterations`: maximum number of iterations of the VEGAS/VAMP refinement. It is not necessary to set this parameter, but e. g. `iterations = 0` is useful for illustrating the effect of adaption.

6.2 `circe2_tool` Demonstration

We can use the example of figure 1 (a simulated realistic $\gamma\gamma$ luminosity spectrum (helicities: $(+,+)$) for a 500 GeV photon collider at TESLA [7]) to demonstrate the effects of different options. In order to amplify the effects, only 20 bins are used in each direction, but figure 8 will show that adequate results are achievable in this case too.

In figure 3, 20 equidistant bins in each direction

```
bins = 20 iterations = 0
```

produce an acceptable description of the high energy peak but are clearly inadequate for $z < 0.2$. In the blown up region, neither 20 equidistant bins nor 50 equidistant bin produce more than a handful of events and remain almost invisible. The bad low energy behaviour can be understood from the convolution of the obviously coarse approximations in left figure of figure 4. Letting the grid adapt

```
bins = 20
```

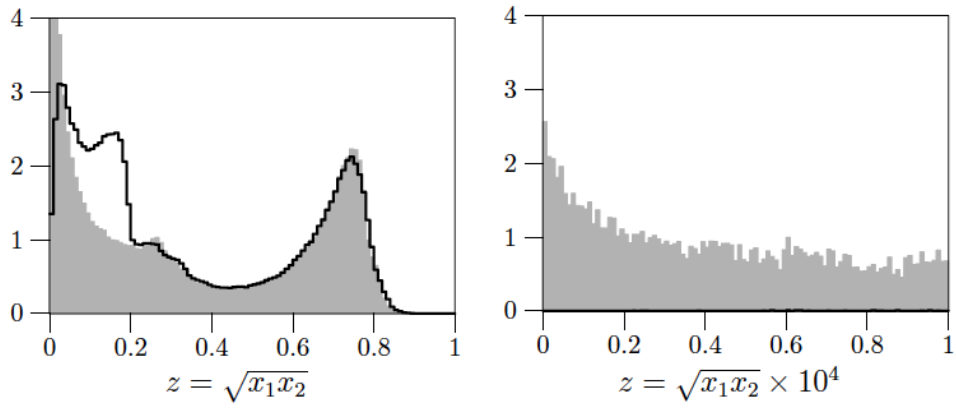

Figure 3: Using 20 equidistant bins in each direction. In the region blown up on the right neither 20 equidistant bins nor 50 equidistant bin produce enough events to be visible. In this and all following plots, the simulated input data is shown as a filled histogram, and the `Circe2` parametrization is shown as a solid line.

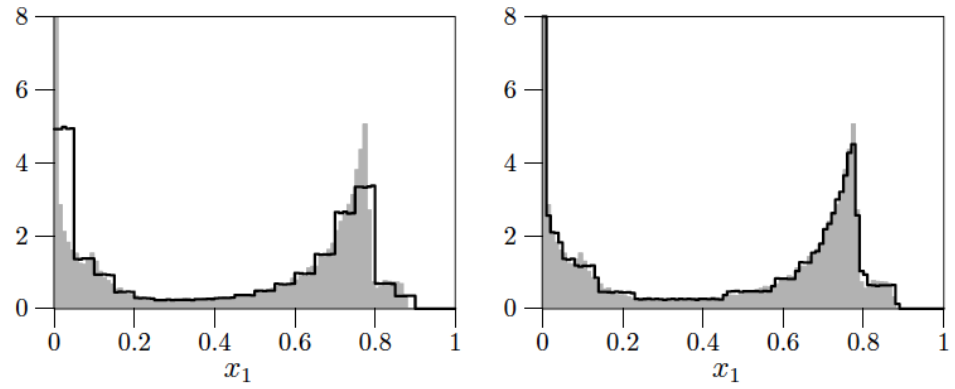

Figure 4: Using 20 bins, both equidistant (left) and adapted (right).

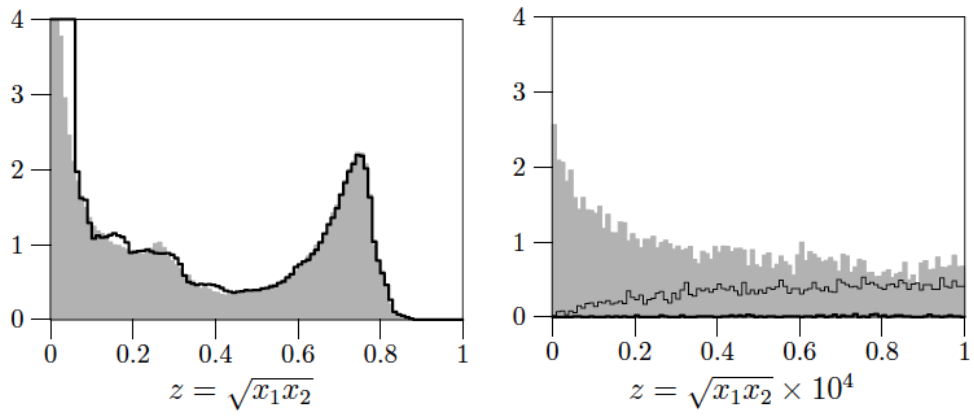

Figure 5: 20 adapted bins. In the blow-up, the 50 bin result is shown for illustration as a thin line, while the 20 bin result remains almost invisible.

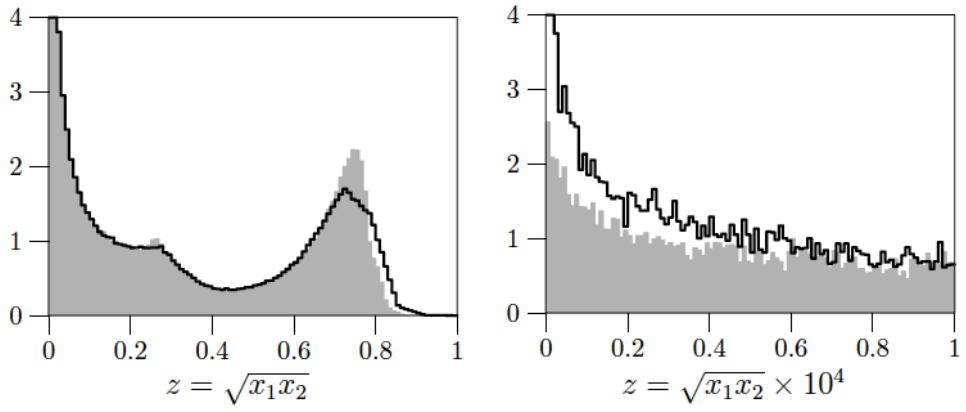

Figure 6: Using 20 equidistant bins in each direction with a power map appropriate for $x^{-0.67}$.

produces a much better approximation in the right figure of figure 4. And indeed, the convolution in figure 5 is significantly improved for $x \lesssim 0.2$, but remains completely inadequate in the very low energy region, blown up on the right hand side.

A better description of the low energy tail requires a power map and figure 6 shows that equidistant bins

```
map = power { 20 [0,1] beta = -0.67 eta = 0 } iterations = 0
```

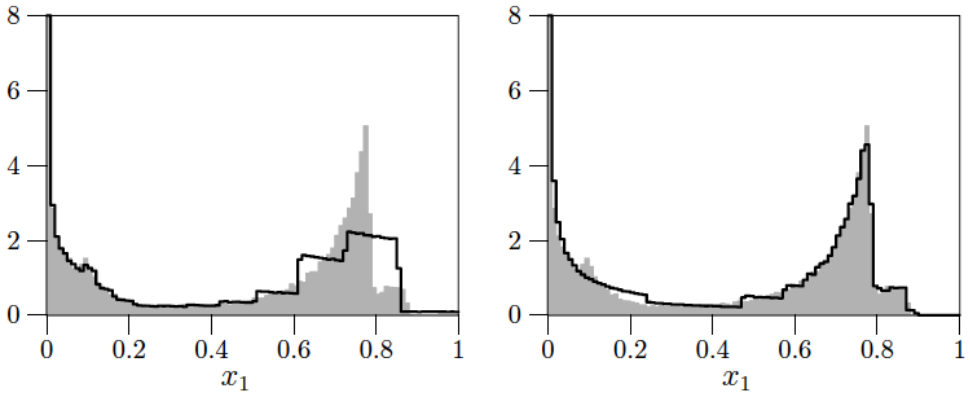

Figure 7: Using 20 bins with a power map appropriate for $x^{-0.67}$, equidistant (left) and adapted (right).

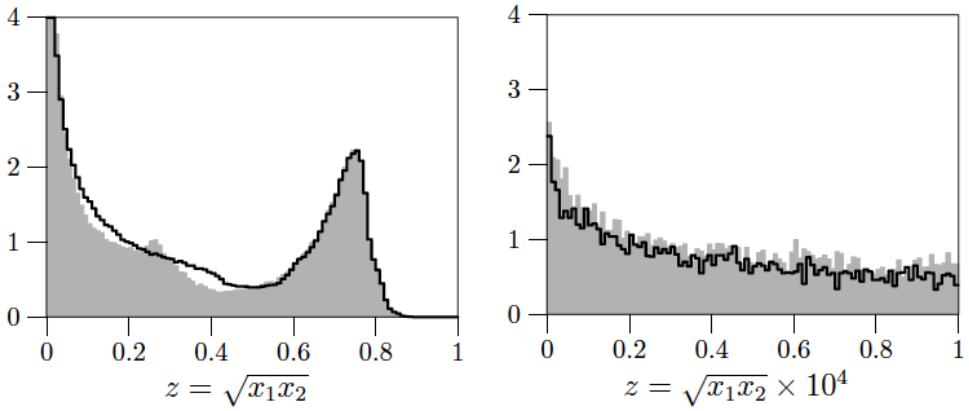

Figure 8: Using 20 adapted bins in each direction with a power map appropriate for $x^{-0.67}$.

already produce a much improved description of the low energy region, including the blow-up on the right hand side. However, the description of the peak has gotten much worse, which is explained by the coarsening of the bins in the high energy region, as shown in figure 7. The best result is obtained by combining a power map with adaption

```
map = power { 20 [0,1] beta = -0.67 eta = 0 }
```

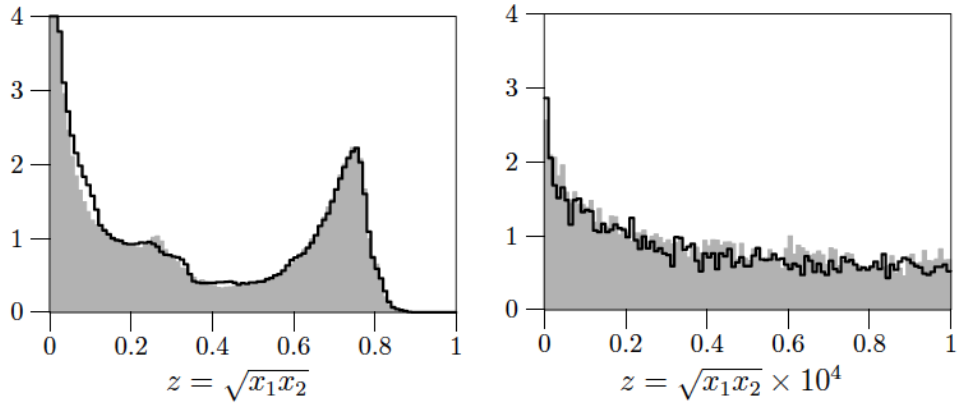

Figure 9: Using $4 + 16$ adapted bins in each direction with a power map appropriate for $x^{-0.67}$ in the 4 bins below $x < 0.05$.

with the results depicted in figure 8. Balancing the number of bins used for a neighborhood of the integrable singularity at $x_i \to 0$ and the remainder can be improved by allocating a fixed number of bins for each

```
map = power { 4 [0,0.05] beta = -0.67 eta = 0 }
map = id { 16 [0.05,1] }
```

as shown in figure 9. If the data were not stochastic, this manual allocation would not be necessary, because the neighborhood of the singularity would not contribute to the variance and consequently use few bins. However, the stochastic variance an not be suppressed and will pull in more bins than useful. If the power of the map were overcompensating the power of the singularity, instead of being tuned to it, the limit $x_i \to 0$ would would be suppressed automatically. But in this case, the low-energy tail could not be described accurately.

The description with 20 bins in figure 9 is not as precise as the 50 bins

```
map = power { 10 [0,0.05] beta = -0.67 eta = 0 }
map = id { 40 [0.05,1] }
```

in figure 1, but can suffice for many studies and requires less than one sixth of the storage space.

6.3 More `circe2_tool` Examples

Here is an example that can be used to demonstrate the beneficial effects of powermaps. The simulated events in `teslagg_500.gg.++.events` are

processed once with map and once without a map. Both times 50 bins are used in each direction.

```
{ file = "test_mappings.circe"
  { design = "TESLA" roots = 500
    { pid/1 = 22 pid/2 = 11 pol/1 = 1 pol/2 = 1
      events = "teslagg_500.gg.++.events" binary lumi = 110.719
      bins/1 = 50
      map/2 = id { 49 [0,0.9999999999] }
      map/2 = id { 1 [0.9999999999,1] } } }
  { design = "TESLA (mapped)" roots = 500
    { pid/1 = 22 pid/2 = 11 pol/1 = 1 pol/2 = 1
      events = "teslagg_500.gg.++.events" binary lumi = 110.719
      map/1 = power { 50 [0,1] beta = -0.67 eta = 0 }
      map/2 = power { 49 [0,0.9999999999] beta = -0.6 eta = 1 }
      map/2 = id { 1 [0.9999999999,1] } } } }
```

In a second step, the distributions generated from both designs in `test_mappings.circe` can be compared with the original distribution.

7 On the Implementation of `circe2_tool`

7.1 Divisions


VEGAS/VAMP, basically ...

7.2 Differentiable Maps

7.3 Polydivisions


patched divisions ...

7.4 Grids

8 The Next Generation

Future generations can try to implement the following features:

`!` *any comment*		optional, repeatable
`CIRCE2 FORMAT#1`		mandatory start line
`design, roots`		
'name' $\sqrt{s}$		mandatory quotes!
`#channels, pol.support`		
n_c *'name'*		mandatory quotes!
`pid1, pol1, pid2, pol2, lumi`	⌉	repeat n_c times
$p_1\ h_1\ p_2\ h_2\ \int\mathcal{L}$		
`#bins1, #bins2, triangle?`		
$n_1\ n_2\ t$		
`x1, map1, alpha1, xi1, eta1, a1, b1`		
$x_{1,0}$		
$x_{1,1}\ \ m_{1,1}\ \ \alpha_{1,1}\ \ \xi_{1,1}\ \ \eta_{1,1}\ \ a_{1,1}\ \ b_{1,1}$		
$\ldots$		
$x_{1,n_1}\ m_{1,n_1}\ \alpha_{1,n_1}\ \xi_{1,n_1}\ \eta_{1,n_1}\ a_{1,n_1}\ b_{1,n_1}$		
`x2, map2, alpha2, xi2, eta2, a2, b2`		
$x_{2,0}$		
$x_{2,1}\ \ m_{2,1}\ \ \alpha_{2,1}\ \ \xi_{2,1}\ \ \eta_{2,1}\ \ a_{2,1}\ \ b_{2,1}$		
$\ldots$		
$x_{2,n_2}\ m_{2,n_2}\ \alpha_{2,n_2}\ \xi_{2,n_2}\ \eta_{2,n_2}\ a_{2,n_2}\ b_{2,n_2}$		
`weights`		
$w_1\ \ [w_1\chi_1^{0,1}\ \ w_1\chi_1^{0,2}\ \ \ldots\ \ w_1\chi_1^{3,3}]$		optional $w\cdot\chi$
$w_2\ \ [w_1\chi_2^{0,1}\ \ w_1\chi_2^{0,2}\ \ \ldots\ \ w_1\chi_2^{3,3}]$		
$\ldots$		
$w_{n_1n_2}\ \ [w_1\chi_{n_1n_2}^{0,1}\ \ w_1\chi_{n_1n_2}^{0,2}\ \ \ldots\ \ w_1\chi_{n_1n_2}^{3,3}]$	⌋	end repeat
`ECRIC2`		mandatory end line

Table 2: File format. The variable input lines (except comments) are designed to be readable by FORTRAN90 'list-directed' input. The files are generated from simulation data with the program `circe2_tool` and are read transparently by the procedure `cir2ld`. The format is documented here only for completeness.

```
module type Division =
  sig
    type t
    val copy : t -> t
    val record : t -> float -> float -> unit
    val rebin : ?power:float -> t -> t
    val find : t -> float -> int
    val bins : t -> float array
    val to_channel : out_channel -> t -> unit
  end
```

Figure 10: O'Caml signature for divisions. `Division.t` is the abstract data type for division of a real interval. Note that `Division` does *not* contain a function `create : ... -> t` for constructing maps. This is provided by concrete implementations (see figures 11 and 14), that can be projected on `Diffmap`

```
module type Mono_Division =
  sig
    include Division
    val create : int -> float -> float -> t
  end
```

Figure 11: O'Caml signature for simple divisions of an interval. The `create` function returns an equidistant starting division.

8.1 Variable # of Bins

One can monitor the total variance in each interval of the polydivisions and move bins from smooth intervals to wildly varying intervals, keeping tho total number of bins constant.

8.2 Adapting Maps Per-Cell

Iff there is enough statistics, one can adapt the mapping class and parameters per bin.

There's a nice duality between adapting bins for a constant mapping on one side and adapting mappings for constant bins. Can one merge the two approaches.

```
module type Diffmap =
  sig
    type t
    type domain
    val x_min : t -> domain
    val x_max : t -> domain
    type codomain
    val y_min : t -> codomain
    val y_max : t -> codomain
    val phi : t -> domain -> codomain
    val ihp : t -> codomain -> domain
    val jac : t -> domain -> float
    val caj : t -> codomain -> float
  end

module type Real_Diffmap =
  T with type domain = float and type codomain = float
```

Figure 12: O'Caml signature for differentiable maps. `Diffmap.t` is the abstract data type for differentiable maps. Note that `Diffmap` does *not* contain a functions like `create : ... -> t` for constructing maps. These are provided by concrete implementations, that can be projected onto `Diffmap`.

```
module type Real_Diffmaps =
  sig
    include Real_Diffmap
    val id : float -> float -> t
  end
```

Figure 13: Collections of real differentiable maps, including at least the identity. The function `id` returns an identity map from a real interval onto itself.

8.3 Non-Factorized Polygrids

One could think of a non-factorized distribution of mappings.

```
module type Poly_Division =
  sig
    include Division
    module M : Real_Diffmaps
    val create :
      (int * M.t) list -> int -> float -> float -> t
  end

module Make_Poly_Division (M : Real_Diffmaps) :
  Poly_Division with module Diffmaps = M
```

Figure 14: O'Caml signature for divisions of an interval, with piecewise differentiable mappings, as specified by the first argument of `create`. The functor `Make_Poly_Division` ...

```
module type Grid =
  sig
    module D : Division
    type t
    val create : D.t -> D.t -> t
    val copy : t -> t
    val record : t -> float -> float -> float -> unit
    val rebin : ?power:float -> t -> t
    val variance : t -> float
    val to_channel : out_channel -> t -> unit
  end

module Make_Grid (D : Division) : Grid with module D = D
```

Figure 15: O'Caml signature for grids. The functor `Make_Grid` can be applied to *any* module of type `Division`, in particular both `Mono_Division` and `Poly_Division`.

9 Conclusions

Acknowledgements

Thanks to Valery Telnov for useful discussions. Thanks to the worldwide Linear Collider community and the ECFA/DESY study groups in particular for encouragement. This research is supported by Bundesministerium für

Bildung und Forschung, Germany, (05 HT9RDA).

References

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

[2] T. Sjostrand, L. Lonnblad and S. Mrenna, *PYTHIA 6.2: Physics and manual*, LU-TP 01-21, [hep-ph/0108264].

[3] G. Corcella et al., *Herwig 6.3 release note*, CAVENDISH-HEP 01-08, CERN-TH 2001-173, DAMTP 2001-61, [hep-ph/0107071].

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

[5] G. P. Lepage, J. Comp. Phys. **27**, 192 (1978); G. P. Lepage, Technical Report No. CLNS-80/447, Cornell (1980).

[6] T. Ohl, Comput. Phys. Commun. **120** (1999) 13 [hep-ph/9806432]; T. Ohl, *Electroweak Gauge Bosons at Future Electron-Positron Colliders*, Darmstadt University of Technology, 1999, IKDA 99/11, LC-REV-1999-005 [hep-ph/9911437].

[7] V. Telnov, 2001 (private communication).

[8] Xavier Leroy, *The Objective Caml System, Release 3.02, Documentation and User's Guide*, Technical Report, INRIA, 2001, `http://pauillac.inria.fr/ocaml/`.

[9] I. F. Ginzburg, G. L. Kotkin, S. L. Panfil, V. G. Serbo and V. I. Telnov, Nucl. Instrum. Meth. A **219** (1984) 5.

[10] P. Chen, G. Horton-Smith, T. Ohgaki, A. W. Weidemann and K. Yokoya, Nucl. Instrum. Meth. **A355** (1995) 107.

[11] D. E. Groom *et al.* [Particle Data Group Collaboration], *Review of particle physics*, Eur. Phys. J. **C15** (2000) 1.

[12] D.E. Knuth, *The Art of Computer Programming, Vol. 2*, (3rd ed.), Addison-Wesley, Reading, MA, 1997.

[13] George Marsaglia, *The Marsaglia Random Number CD-ROM*, FSU, Dept. of Statistics and SCRI, 1996.

[14] S. Jadach, Comput. Phys. Commun. **130** (2000) 244 [arXiv:physics/9910004].

10 Implementation of `circe2`

45a ⟨`implicit none` 45a⟩≡
```
  implicit none
```

45b ⟨`circe2.f90` 45b⟩≡
```
! circe2.f90 -- beam spectra for linear colliders and photon colliders
```
⟨*Copyleft notice* 45d⟩
⟨*Separator* 45c⟩
```
module circe2
  use kinds

```
⟨`implicit none` 45a⟩
```
  private

```
⟨*Public procedures* 48d⟩

⟨*Public types* 56b⟩

⟨`parameter` *declarations* 51b⟩

⟨*Declaration: circe2 parameters* 48e⟩

```
  type(circe2_params_t), public, save :: c2p

```
⟨*Abstract types* 56c⟩

⟨*Abstract interfaces* 56a⟩

```
contains

```
⟨*Procedures* 55e⟩
```
end module circe2
```

45c ⟨*Separator* 45c⟩≡
```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

45d ⟨*Copyleft notice* 45d⟩≡
```
! $Id: circe2.nw,v 1.56 2002/10/14 10:12:06 ohl Exp $
! Copyright (C) 2001-2014 by
```

```
!      Wolfgang Kilian <kilian@physik.uni-siegen.de>
!      Thorsten Ohl <ohl@physik.uni-wuerzburg.de>
!      Juergen Reuter <juergen.reuter@desy.de>
!      Christian Speckner <cnspeckn@googlemail.com>
!
! Circe2 is free software; you can redistribute it and/or modify it
! under the terms of the GNU General Public License as published by
! the Free Software Foundation; either version 2, or (at your option)
! any later version.
!
! Circe2 is distributed in the hope that it will be useful, but
! WITHOUT ANY WARRANTY; without even the implied warranty of
! MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
! GNU General Public License for more details.
!
! You should have received a copy of the GNU General Public License
! along with this program; if not, write to the Free Software
! Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
```

46a ⟨circe2_c.c 46a⟩≡

```
/* circe2_c.c -- beam spectra for linear colliders and photon colliders */
⟨Copyleft notice (C) 47b⟩
⟨Separator2 (C) 46b⟩
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <math.h>
#undef min
#define min(a,b) ((a) < (b) ? (a) : (b))
#undef max
#define max(a,b) ((a) > (b) ? (a) : (b))
⟨Separator2 (C) 46b⟩
⟨Macros (C) 64a⟩
⟨Separator2 (C) 46b⟩
⟨Well known constants (C) 58b⟩
⟨Separator2 (C) 46b⟩
⟨Data type declarations (C) 49a⟩
⟨Separator2 (C) 46b⟩
⟨Private Procedures (C) 47c⟩
⟨Separator2 (C) 46b⟩
⟨Public Procedures (C) 57a⟩
```

46b ⟨Separator2 (C) 46b⟩≡

⟨*Separator (C)* 47a⟩

47a ⟨*Separator (C)* 47a⟩≡
```
/*******************************************************************/
```

47b ⟨*Copyleft notice (C)* 47b⟩≡
```
/* $Id: circe2.nw,v 1.56 2002/10/14 10:12:06 ohl Exp $
   Copyright (C) 2002-2011 by
        Wolfgang Kilian <kilian@physik.uni-siegen.de>
        Thorsten Ohl <ohl@physik.uni-wuerzburg.de>
        Juergen Reuter <juergen.reuter@desy.de>
        Christian Speckner <cnspeckn@googlemail.com>

   Circe2 is free software; you can redistribute it and/or modify it
   under the terms of the GNU General Public License as published by
   the Free Software Foundation; either version 2, or (at your option)
   any later version.

   Circe2 is distributed in the hope that it will be useful, but
   WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
   GNU General Public License for more details.

   You should have received a copy of the GNU General Public License
   along with this program; if not, write to the Free Software
   Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA. */
```

Can't live without it:

47c ⟨*Private Procedures (C)* 47c⟩≡
```
void *
xmalloc (size_t size)
{
  void *ptr = malloc (size);
  if (ptr == NULL) {
    fprintf (stderr, "can't get %d bytes ... exiting!\n", size);
    exit (-1);
  }
  return ptr;
}
```

47d ⟨circe2_cpp.cc 47d⟩≡
```
// circe2_cpp.cc -- beam spectra for linear colliders and photon colliders
```
⟨*Copyleft notice (C++)* 48c⟩
⟨*Separator2 (C++)* 48a⟩
⟨*Data type declarations (C++)* (never defined)⟩

⟨*Procedures (C++)* (never defined)⟩

48a ⟨*Separator2 (C++)* 48a⟩≡
```
//
⟨Separator (C++) 48b⟩
//
```

48b ⟨*Separator (C++)* 48b⟩≡
```
// ------------------------------------------------------------------------
```

48c ⟨*Copyleft notice (C++)* 48c⟩≡
```
// $Id: circe2.nw,v 1.56 2002/10/14 10:12:06 ohl Exp $
// Copyright (C) 2002-2011 by
//      Wolfgang Kilian <kilian@physik.uni-siegen.de>
//      Thorsten Ohl <ohl@physik.uni-wuerzburg.de>
//      Juergen Reuter <juergen.reuter@desy.de>
//      Christian Speckner <cnspeckn@googlemail.com>
//
// Circe2 is free software; you can redistribute it and/or modify it
// under the terms of the GNU General Public License as published by
// the Free Software Foundation; either version 2, or (at your option)
// any later version.
//
// Circe2 is distributed in the hope that it will be useful, but
// WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
// GNU General Public License for more details.
//
// You should have received a copy of the GNU General Public License
// along with this program; if not, write to the Free Software
// Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
```

11 Data

The `circe2_params_t` type (old `/circ2cm/` common block which had to be arranged such that the constant `parameter`s come first and the variables in order of decreasing alignment constraints):

48d ⟨*Public procedures* 48d⟩≡
```
public :: circe2_params_t
```

48e ⟨*Declaration: circe2 parameters* 48e⟩≡
```
type :: circe2_params_t
  ⟨8-byte aligned part of circe2 parameters 50b⟩
  ⟨4-byte aligned part of circe2 parameters 51a⟩
```

```
  end type circe2_params_t
```

49a ⟨*Data type declarations (C)* 49a⟩≡

```
typedef struct
{
  int n;
  ⟨circe2_division components 53a⟩
} circe2_division;
```

49b ⟨*Private Procedures (C)* 47c⟩+≡

```
static circe2_division *
circe2_new_division (int nbins)
{
  circe2_division *d;
  d = xmalloc (sizeof (circe2_division));
  d->n = nbins;
  ⟨Allocate circe2_division components *d 53b⟩
  return d;
}
```

49c ⟨*Data type declarations (C)* 49a⟩+≡

```
typedef struct
{
  circe2_division *d1, *d2;
  ⟨circe2_channel components 51c⟩
} circe2_channel;
```

49d ⟨*Private Procedures (C)* 47c⟩+≡

```
static circe2_channel *
circe2_new_channel (int nbins1, int nbins2)
{
  circe2_channel *p;
  int i;
  p = xmalloc (sizeof (circe2_channel));
  p->d1 = circe2_new_division (nbins1);
  p->d2 = circe2_new_division (nbins2);
  ⟨Allocate circe2_channel components *p 51d⟩
  return p;
}
```

49e ⟨*Data type declarations (C)* 49a⟩+≡

```
typedef struct
{
  int n;
```

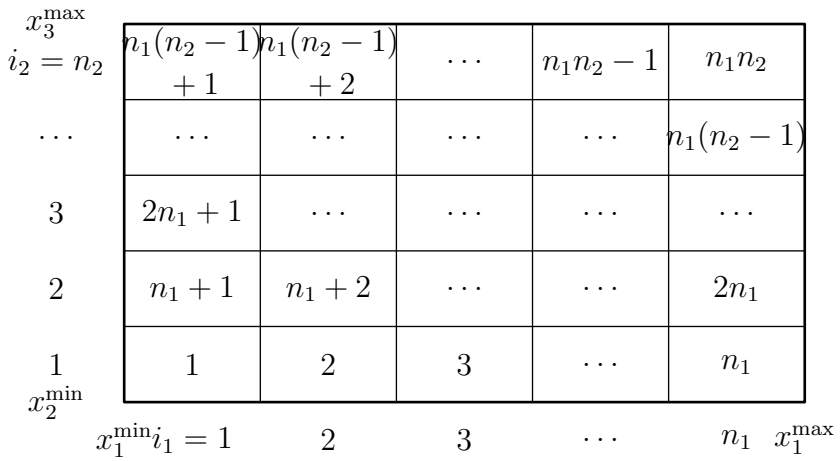

Figure 16: Enumerating the bins linearly, starting from 1 (Fortran style). Probability distribution functions will have a sentinel at 0 that's always 0.

```
  circe2_channel **ch;
  ⟨circe2_channels components 57c⟩
} circe2_channels;
```

50a ⟨*Private Procedures (C)* 47c⟩+≡

```
static circe2_channels *
circe2_new_channels (int nchannels, int nbins1, int nbins2)
{
  int i;
  circe2_channels *p;
  p = xmalloc (sizeof (circe2_channels));
  p->n = nchannels;
  p->ch = xmalloc (p->n * sizeof (circe2_channel *));
  for (i = 0; i < p->n; i++)
    p->ch[i] = circe2_new_channel (nbins1, nbins2);
  return p;
}
```

We store the probability distribution function as a one-dimensional array `wgt`[3], since this simplifies the binary search used for inverting the distribution. [wgt(0,ic)] is always 0 and serves as a convenient sentinel for the binary search. It is *not* written in the file, which contains the normalized weight of the bins.

[3]The second "dimension" is just an index for the channel.

50b ⟨*8-byte aligned part of circe2 parameters* 50b⟩≡
```
real(kind=double), dimension(0:NBMAX*NBMAX,NCMAX) :: wgt
```

The actual number of bins in each direction is

51a ⟨*4-byte aligned part of circe2 parameters* 51a⟩≡
```
integer, dimension(NCMAX) :: nb1, nb2
```

Of course, we can't make *any* exceptions to the rule `nb` $\leq$ `NBMAX` (similarly for `nc` $\leq$ `NCMAX` below). `NCMAX` $= (3 \cdot 2)^2$ for three flavors ($\{e^+, e^-, \gamma\}$) and two helicity states.

51b ⟨`parameter` *declarations* 51b⟩≡
```
integer, parameter :: NBMAX = 100, NCMAX = 36
```

51c ⟨`circe2_channel` *components* 51c⟩≡
```
double *weight;
```

51d ⟨*Allocate* `circe2_channel` *components* `*p` 51d⟩≡
```
p->weight = xmalloc ((p->d1->n * p->d2->n + 1) * sizeof(double));
```

Using figure 16, calculating the index of a bin from the two-dimensional coordinates is straightforward, of course:

51e ⟨`i` $\leftarrow$ (`i1`, `i2`) 51e⟩≡
```
i = i1 + (i2 - 1) * c2p%nb1(ic)
```

51f ⟨`i` $\leftarrow$ (`i1`, `i2`) *(C)* 51f⟩≡
```
i = i1 + (i2 - 1) * ch->d1->n
```

The inverse

$$i_1 = 1 + ((i-1) \mod n_1) \tag{51a}$$
$$i_2 = 1 + \lfloor (i-1)/n_1 \rfloor \tag{51b}$$

can also be written

$$i_2 = 1 + \lfloor (i-1)/n_1 \rfloor \tag{52a}$$
$$i_1 = i - (i_2 - 1)n_1 \tag{52b}$$

51g ⟨(`i1`, `i2`) $\leftarrow$ `i` 51g⟩≡
```
i2 = 1 + (i - 1) / c2p%nb1(ic)
i1 = i - (i2 - 1) * c2p%nb1(ic)
```

51h ⟨*Private Procedures (C)* 47c⟩+≡
```
static inline void
split_index (circe2_channel *ch, int *i1, int *i2, int i) {
  *i2 = 1 + (i - 1) / ch->d1->n;
  *i1 = i - (*i2 - 1) * ch->d1->n;
}
```

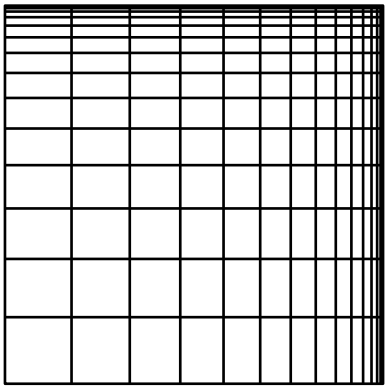

Figure 17: Almost factorizable distributions, like e^+e^-.

The density normalized to the bin size

$$v = \frac{w}{\Delta x_1 \Delta x_2}$$

such that

$$\int \mathrm{d}x_1 \mathrm{d}x_2 \; v = \sum w = 1$$

For mapped distributions, on the level of bins, we can either use the area of the domain and apply a jacobian or the area of the codomain directly

$$\frac{\mathrm{d}x}{\mathrm{d}y} \cdot \frac{1}{\Delta x} \approx \frac{1}{\Delta y} \tag{53}$$

We elect to use the former, because this reflects the distribution of the events generated by `cir2gn` *inside* the bins as well. This quantity is more conveniently stored as a true two-dimensional array:

52a ⟨*8-byte aligned part of circe2 parameters* 50b⟩+≡

```
real(kind=double), dimension(NBMAX,NBMAX,NCMAX) :: val
```

52b ⟨`circe2_channel` *components* 51c⟩+≡

```
double **value;
```

52c ⟨*Allocate* `circe2_channel` *components* `*p` 51d⟩+≡

```
p->value = xmalloc (p->d1->n * sizeof(double *));
for (i = 0; i < p->d1->n; i++)
  p->value[i] = xmalloc (p->d2->n * sizeof(double));
```

52d ⟨*8-byte aligned part of circe2 parameters* 50b⟩+≡

```
real(kind=double), dimension(0:NBMAX,NCMAX) :: xb1, xb2
```

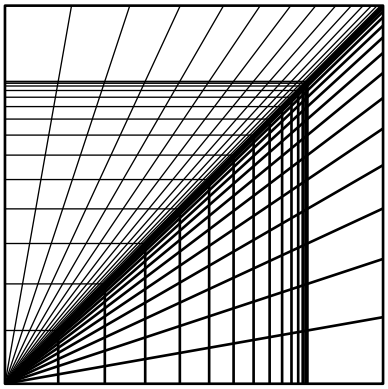

Figure 18: Symmetrical, strongly correlated distributions, e. g. with a ridge on the diagonal, like $\gamma\gamma$ at a γ-collider.

53a ⟨circe2_division *components* 53a⟩≡
```
double *x;
```

53b ⟨*Allocate* circe2_division *components* *d 53b⟩≡
```
d->x = xmalloc ((d->n + 1) * sizeof(double));
```

53c ⟨*4-byte aligned part of circe2 parameters* 51a⟩+≡
```
logical, dimension(NCMAX) :: triang
```

53d ⟨circe2_channel *components* 51c⟩+≡
```
int triangle;
```

11.1 Channels

The actual number of channels $\gamma\gamma$, $e^-\gamma$, e^-e^+, etc.

53e ⟨*4-byte aligned part of circe2 parameters* 51a⟩+≡
```
integer :: nc
```

The particles that are described by this channel and their polarizations:

53f ⟨*4-byte aligned part of circe2 parameters* 51a⟩+≡
```
integer, dimension(NCMAX) :: pid1, pid2
integer, dimension(NCMAX) :: pol1, pol2
```

53g ⟨circe2_division *components* 53a⟩+≡
```
int pid;
int pol;
```

The integrated luminosity of the channel

54a ⟨*8-byte aligned part of circe2 parameters* 50b⟩+≡
```
real(kind=double), dimension(NCMAX) :: lumi
```

54b ⟨`circe2_channel` *components* 51c⟩+≡
```
double lumi;
```

The integrated luminosity of the channel

54c ⟨*8-byte aligned part of circe2 parameters* 50b⟩+≡
```
real(kind=double), dimension(0:NCMAX) :: cwgt
```

54d ⟨`circe2_channel` *components* 51c⟩+≡
```
double channel_weight;
```

11.2 Maps

54e ⟨*4-byte aligned part of circe2 parameters* 51a⟩+≡
```
integer, dimension(NBMAX,NCMAX) :: map1, map2
```

54f ⟨`circe2_division` *components* 53a⟩+≡
```
int *map;
```

54g ⟨*Allocate* `circe2_division` *components* `*d` 53b⟩+≡
```
d->map = xmalloc (d->n * sizeof(int));
```

54h ⟨*8-byte aligned part of circe2 parameters* 50b⟩+≡
```
real(kind=double), dimension(0:NBMAX,NCMAX) :: yb1, yb2
```

54i ⟨`circe2_division` *components* 53a⟩+≡
```
double *y;
```

54j ⟨*Allocate* `circe2_division` *components* `*d` 53b⟩+≡
```
d->y = xmalloc ((d->n + 1) * sizeof(double));
```

54k ⟨*8-byte aligned part of circe2 parameters* 50b⟩+≡
```
real(kind=double), dimension(NBMAX,NCMAX) :: alpha1, alpha2
real(kind=double), dimension(NBMAX,NCMAX) :: xi1, xi2
real(kind=double), dimension(NBMAX,NCMAX) :: eta1, eta2
real(kind=double), dimension(NBMAX,NCMAX) :: a1, a2
real(kind=double), dimension(NBMAX,NCMAX) :: b1, b2
```

54l ⟨`circe2_division` *components* 53a⟩+≡
```
double *alpha;
double *xi, *eta;
double *a, *b;
```

55a ⟨*Allocate* `circe2_division` *components* `*d` 53b⟩+≡

```
d->alpha = xmalloc (d->n * sizeof(double));
d->xi = xmalloc (d->n * sizeof(double));
d->eta = xmalloc (d->n * sizeof(double));
d->a = xmalloc (d->n * sizeof(double));
d->b = xmalloc (d->n * sizeof(double));
```

12 Random-Number Generator

The generator routines do not fix or provide a random-number generator. The caller has to provide an implementation which is transferred to the subroutines in one of two possible forms:

1. as a subroutine which generates a single random number, working on an implicit external state
2. as an object with a method the generates a single random number, working on an internal state

We allow for using both forms, only in the Fortran API.

These snippets should be used by the procedures that use a RNG:

55b ⟨*RNG dummy arguments* 55b⟩≡

```
rng, rng_obj
```

55c ⟨*RNG dummy declarations* 55c⟩≡

```
procedure(rng_proc), optional :: rng
class(rng_type), intent(inout), optional :: rng_obj
```

55d ⟨*RNG: generate* `u` 55d⟩≡

```
call rng_call (u, ⟨RNG dummy arguments 55b⟩)
```

55e ⟨*Procedures* 55e⟩≡

```
subroutine rng_call (u, ⟨RNG dummy arguments 55b⟩)
  real(kind=double), intent(out) :: u
  ⟨RNG dummy declarations 55c⟩
  if (present (rng)) then
     call rng (u)
  else if (present (rng_obj)) then
     call rng_obj%generate (u)
  else
     stop "circe2: internal error: generator requires either rng &
          &or rng_obj argument"
  end if
end subroutine rng_call
```

This defines the procedure version of the RNG, corresponding to the traditional F77 `external` interface. The abstract interface enables the compiler to check conformance.

56a ⟨*Abstract interfaces* 56a⟩≡
```
abstract interface
  subroutine rng_proc (u)
    import :: double
    real(kind=double), intent(out) :: u
  end subroutine rng_proc
end interface
```

Here we define the object version of the RNG. It has to implement a `generate` method which parallels the `rng_proc` procedure above.

56b ⟨*Public types* 56b⟩≡
```
public :: rng_type
```

56c ⟨*Abstract types* 56c⟩≡
```
type, abstract :: rng_type
 contains
   procedure(rng_generate), deferred :: generate
end type rng_type
```

56d ⟨*Abstract interfaces* 56a⟩+≡
```
abstract interface
   subroutine rng_generate (rng_obj, u)
     import :: rng_type, double
     class(rng_type), intent(inout) :: rng_obj
     real(kind=double), intent(out) :: u
   end subroutine rng_generate
end interface
```

13 Event Generation

Generate a two-dimensional distribution for (`x1`, `x2`) according to the histogram for channel `ic`.

56e ⟨*Public procedures* 48d⟩+≡
```
public :: cir2gn
```

56f ⟨*Procedures* 55e⟩+≡
```
subroutine cir2gn (p1, h1, p2, h2, y1, y2, ⟨RNG dummy arguments 55b⟩)
  integer :: p1, h1, p2, h2
```

```
    real(kind=double) :: y1, y2
    ⟨RNG dummy declarations 55c⟩
    integer :: i, ic, i1, i2, ibot, itop
    real(kind=double) :: x1, x2
    real(kind=double) :: u, tmp
    ⟨Find ic for p1, h1, p2 and h2 58c⟩
    ⟨Complain and return iff ic ≤ 0 59a⟩
    ⟨RNG: generate u 55d⟩
    ⟨Do a binary search for wgt(i − 1) ≤ u < wgt(i) 59c⟩
    ⟨(i1, i2) ← i 51g⟩
    ⟨x1 ∈ [xb1(i1 − 1), xb1(i1)] 60b⟩
    ⟨x2 ∈ [xb2(i2 − 1), xb2(i2)] 60c⟩
    ⟨y1 ← x1 61a⟩
    ⟨y2 ← x2 61b⟩
    ⟨Inverse triangle map 62a⟩
  end subroutine cir2gn
```

57a ⟨*Public Procedures (C)* 57a⟩≡

```
  void
  circe2_generate (circe2_channels *channels,
                   int p1, int h1, int p2, int h2,
                   double *y1, double *y2,
                   void (*rng)(double *))
  {
    circe2_channel *ch = NULL;
    int i, i1, i2;
    double u, x1, x2;
    ch = circe2_find_channel (channels, p1, h1, p2, h2);
    ⟨Complain and return iff ch = NULL (C) 59b⟩
    rng (&u);
    i = binary_search (ch->weight, 0, ch->d1->n * ch->d2->n, u);
    split_index (ch, &i1, &i2, i);
    ⟨x1 ∈ [ch− > d1− > x[i1 − 1] , ch− > d1− > x[i1] ] (C) 60d⟩
    ⟨x2 ∈ [ch− > d2− > x[i2 − 1] , ch− > d2− > x[i2] ] (C) 60e⟩
    ⟨y1 ← x1 (C) 61c⟩
    ⟨y2 ← x2 (C) 61d⟩
    ⟨Inverse triangle map (C) 62b⟩
  }
```

57b ⟨*4-byte aligned part of circe2 parameters* 51a⟩+≡

```
    integer :: polspt
```

57c ⟨circe2_channels *components* 57c⟩≡

```
  int polarization_support;
```

58a ⟨parameter *declarations* 51b⟩+≡

```
integer, parameter :: POLAVG = 1, POLHEL = 2, POLGEN = 3
```

58b ⟨*Well known constants (C)* 58b⟩≡

```
#define POLAVG 1
#define POLHEL 2
#define POLGEN 3
```

A linear search for a matching channel should suffice, because the number if channels `nc` will always be a small number. The most popular channels should be first in the list, anyway.

58c ⟨*Find* `ic` *for* `p1`, `h1`, `p2` *and* `h2` 58c⟩≡

```
ic = 0
if (((c2p%polspt .eq. POLAVG) .or. (c2p%polspt .eq. POLGEN)) &
     .and. ((h1 .ne. 0) .or. (h2 .ne. 0))) then
   write (*, '(2A)') 'circe2: current beam description ', &
        'supports only polarization averages'
else if ((c2p%polspt .eq. POLHEL) &
     .and. ((h1 .eq. 0) .or. (h2 .eq. 0))) then
   write (*, '(2A)') 'circe2: polarization averages ', &
        'not supported by current beam description'
else
   do i = 1, c2p%nc
      if (       (p1 .eq. c2p%pid1(i)) .and. (h1 .eq. c2p%pol1(i)) &
           .and. (p2 .eq. c2p%pid2(i)) .and. (h2 .eq. c2p%pol2(i))) then
         ic = i
      end if
   end do
end if
```

58d ⟨*Private Procedures (C)* 47c⟩+≡

```
static circe2_channel *
circe2_find_channel (circe2_channels *channels,
                     int p1, int h1, int p2, int h2)
{
  int i;
  if (((channels->polarization_support == POLAVG)
       || (channels->polarization_support == POLGEN))
      && ((h1 != 0) || (h2 != 0))) {
    fprintf (stderr,
             "circe2: current beam description "
             "supports only polarization averages\n");
    return NULL;
  }
  if ((channels->polarization_support == POLHEL)
```

```
            && ((h1 == 0) || (h2 == 0))) {
      fprintf (stderr,
               "circe2: polarization averages not "
               "supported by current beam description\n");
      return NULL;
    }
    for (i = 0; i < channels->n; i++) {
      circe2_channel *ch = channels->ch[i];
      if ((p1 == ch->d1->pid) && (h1 == ch->d1->pol)
          && (p2 == ch->d2->pid) && (h2 == ch->d2->pol))
        return ch;
    }
    return NULL;
  }
```

$-3.4 \cdot 10^{38}$ is a very large negative number that can be represented in typical 4-byte floating point numbers:

59a ⟨*Complain and* `return` *iff* `ic` ≤ 0 59a⟩≡

```
      if (ic .le. 0) then
         write (*, '(A,2I4,A,2I3)') &
              'circe2: no channel for particles', p1, p2, &
              ' and polarizations', h1, h2
         y1 = -3.4E+38
         y2 = -3.4E+38
         return
      end if
```

59b ⟨*Complain and* `return` *iff* `ch` $=$ *NULL (C)* 59b⟩≡

```
  if (ch == NULL) {
    fprintf (stderr,
             "circe2: no channel for particles (%d, %d) "
             "and polarizations (%d, %d)\n", p1, p2, h1, h2);
    *y1 = -3.4e+38;
    *y2 = -3.4e+38;
    return;
  }
```

The number of bins is typically *much* larger and we must use a binary search to get a reasonable performance.

59c ⟨*Do a binary search for* `wgt(i - 1)` $\leq$ `u` $<$ `wgt(i)` 59c⟩≡

```
        ibot = 0
        itop = c2p%nb1(ic) * c2p%nb2(ic)
        do
          if (itop .le. (ibot + 1)) then
             i = ibot + 1
```

```
            exit
          else
            i = (ibot + itop) / 2
            if (u .lt. c2p%wgt(i,ic)) then
              itop = i
            else
              ibot = i
            end if
          end if
        end do
```

60a ⟨*Private Procedures (C)* 47c⟩+≡

```
static int
binary_search (double *x, int bot, int top, double u)
{
  int low = bot;
  int high = top;
  while (1) {
    if (high <= (low + 1)) {
      return (low + 1);
      break;
    } else {
      int i = (low + high) / 2;
      if (u < x[i])
        high = i;
      else
        low = i;
    }
  }
}
```

60b ⟨x1 ∈ [xb1(i1 − 1), xb1(i1)] 60b⟩≡

```
⟨RNG: generate u 55d⟩
x1 = c2p%xb1(i1,ic)*u + c2p%xb1(i1-1,ic)*(1-u)
```

60c ⟨x2 ∈ [xb2(i2 − 1), xb2(i2)] 60c⟩≡

```
⟨RNG: generate u 55d⟩
x2 = c2p%xb2(i2,ic)*u + c2p%xb2(i2-1,ic)*(1-u)
```

60d ⟨x1 ∈ [ch− > d1− > x[i1 − 1] , ch− > d1− > x[i1]] *(C)* 60d⟩≡

```
rng (&u);
x1 = ch->d1->x[i1]*u + ch->d1->x[i1-1]*(1-u);
```

60e ⟨x2 ∈ [ch− > d2− > x[i2 − 1] , ch− > d2− > x[i2]] *(C)* 60e⟩≡

```
rng (&u);
x2 = ch->d2->x[i2]*u + ch->d2->x[i2-1]*(1-u);
```

61a ⟨y1 ← x1 61a⟩≡

```
if (c2p%map1(i1,ic) .eq. 0) then
   y1 = x1
else if (c2p%map1(i1,ic) .eq. 1) then
   y1 = (c2p%a1(i1,ic)*(x1-c2p%xi1(i1,ic)))**c2p%alpha1(i1,ic) / c2p%b1(i1,ic) &
          + c2p%eta1(i1,ic)
else if (c2p%map1(i1,ic) .eq. 2) then
   y1 = c2p%a1(i1,ic) * tan(c2p%a1(i1,ic)*(x1-c2p%xi1(i1,ic))/c2p%b1(i1,ic)**2) &
          + c2p%eta1(i1,ic)
else
   write (*, '(A,I3)') &
        'circe2: internal error: invalid map: ', c2p%map1(i1,ic)
end if
```

61b ⟨y2 ← x2 61b⟩≡

```
if (c2p%map2(i2,ic) .eq. 0) then
   y2 = x2
else if (c2p%map2(i2,ic) .eq. 1) then
   y2 = (c2p%a2(i2,ic)*(x2-c2p%xi2(i2,ic)))**c2p%alpha2(i2,ic) / c2p%b2(i2,ic) &
          + c2p%eta2(i2,ic)
else if (c2p%map2(i2,ic) .eq. 2) then
   y2 = c2p%a2(i2,ic) * tan(c2p%a2(i2,ic)*(x2-c2p%xi2(i2,ic))/c2p%b2(i2,ic)**2) &
          + c2p%eta2(i2,ic)
else
   write (*, '(A,I3)') &
        'circe2: internal error: invalid map: ', c2p%map2(i2,ic)
end if
```

61c ⟨y1 ← x1 *(C)* 61c⟩≡

```
switch (ch->d1->map[i1]) {
case 0:
  *y1 = x1;
case 1:
  *y1 = pow (ch->d1->a[i1] * (x1 - ch->d1->xi[i1]), ch->d1->alpha[i1])
         / ch->d1->b[i1]
         + ch->d1->eta[i1];
case 2:
  *y1 = ch->d1->a[i1] * tan (ch->d1->a[i1] * (x1 - ch->d1->xi[i1])
                             / (ch->d1->b[i1] * ch->d1->b[i1]))
         + ch->d1->eta[i1];
default:
  fprintf (stderr, "circe2: internal error: invalid map: %d\n", ch->d1->map[i1]);
}
```

61d ⟨y2 ← x2 *(C)* 61d⟩≡

```
switch (ch->d2->map[i2]) {
case 0:
  *y2 = x2;
case 1:
  *y2 = pow (ch->d2->a[i2] * (x2 - ch->d2->xi[i2]), ch->d2->alpha[i2])
         / ch->d2->b[i2]
         + ch->d2->eta[i2];
case 2:
  *y2 = ch->d2->a[i2] * tan (ch->d2->a[i2] * (x2 - ch->d2->xi[i2])
                             / (ch->d2->b[i2] * ch->d2->b[i2]))
         + ch->d2->eta[i2];
default:
  fprintf (stderr, "circe2: internal error: invalid map: %d\n", ch->d2->map[i2]);
}
```

There's still something wrong with *unweighted* events for the case that there is a triangle map *together* with a non-trivial `x2` → `y2` map. *Fix this!!!*

62a ⟨*Inverse triangle map* 62a⟩≡

```
if (c2p%triang(ic)) then
   y2 = y1 * y2
   ⟨Swap y1 and y2 in 50% of the cases 62c⟩
end if
```

62b ⟨*Inverse triangle map (C)* 62b⟩≡

```
if (ch->triangle) {
  *y2 = *y1 * *y2;
  ⟨Swap y1 and y2 in 50% of the cases (C) 62d⟩
}
```

62c ⟨*Swap* `y1` *and* `y2` *in 50% of the cases* 62c⟩≡

```
⟨RNG: generate u 55d⟩
if (2*u .ge. 1) then
   tmp = y1
   y1 = y2
   y2 = tmp
end if
```

62d ⟨*Swap* `y1` *and* `y2` *in 50% of the cases (C)* 62d⟩≡

```
rng (&u);
if (2*u >= 1) {
  double tmp;
  tmp = *y1;
  *y1 = *y2;
```

```
    *y2 = tmp;
}
```

14 Channel selection

We could call `cir2gn` immediately, but then `cir2gn` and `cir2ch` would have the same calling conventions and might have caused a lot of confusion.

63a ⟨*Public procedures* 48d⟩+≡

```
public :: cir2ch
```

63b ⟨*Procedures* 55e⟩+≡

```
subroutine cir2ch (p1, h1, p2, h2, ⟨RNG dummy arguments 55b⟩)
  integer :: p1, h1, p2, h2
  ⟨RNG dummy declarations 55c⟩
  integer :: ic, ibot, itop
  real(kind=double) :: u
  ⟨RNG: generate u 55d⟩
  ibot = 0
  itop = c2p%nc
  do
    if (itop .le. (ibot + 1)) then
       ic = ibot + 1
       p1 = c2p%pid1(ic)
       h1 = c2p%pol1(ic)
       p2 = c2p%pid2(ic)
       h2 = c2p%pol2(ic)
       exit
    else
       ic = (ibot + itop) / 2
       if (u .lt. c2p%cwgt(ic)) then
          itop = ic
       else
          ibot = ic
       end if
    end if
  end do
  write (*, '(A)') 'circe2: internal error'
  stop
end subroutine cir2ch
```

63c ⟨*Public Procedures (C)* 57a⟩+≡

```
void
```

```
circe2_random_channel (circe2_channels *channels,
                       int *p1, int *h1, int *p2, int *h2,
                       void (*rng) (double *))
{
  circe2_channel *ch;
  int ic, ibot, itop;
  double u;
  POINTER_PANIC(circe2_random_channel, channels, "channels");
  rng (&u);
  ibot = 0;
  itop = channels->n;
  while (ibot + 1 < itop) {
    ic = (ibot + itop) / 2;
    if (u < channels->ch[ic]->channel_weight)
      itop = ic;
    else
      ibot = ic;
  }
  ch = channels->ch[ibot + 1];
  POINTER_PANIC(circe2_random_channel, ch, "selected channel");
  *p1 = ch->d1->pid;
  *h1 = ch->d1->pol;
  *p2 = ch->d2->pid;
  *h2 = ch->d2->pol;
}
```

64a ⟨*Macros (C)* 64a⟩≡

```
#define POINTER_PANIC(fct,ptr,name) \
  if (ptr == NULL) { \
    fprintf (stderr, "%s: PANIC: %s not initialized!\n", #fct, name); \
    exit (-1); \
  }
```

Below, we will always have `h1` = `h2` = 0. but we don't have to check this explicitely, because `cir2dm` will do it anyway. The procedure could be made more efficient, since most of `cir2dm` is undoing parts of `cir2gn`.

64b ⟨*Public procedures* 48d⟩+≡

```
  public :: cir2gp
```

64c ⟨*Procedures* 55e⟩+≡

```
  subroutine cir2gp (p1, p2, x1, x2, pol, ⟨RNG dummy arguments 55b⟩)
    integer :: p1, p2
    real(kind=double) :: x1, x2
    real(kind=double), dimension(0:3,0:3) :: pol
    ⟨RNG dummy declarations 55c⟩
```

```
  integer :: h1, h2, i1, i2
  real(kind=double) :: pol00
  call cir2ch (p1, h1, p2, h2, ⟨RNG dummy arguments 55b⟩)
  call cir2gn (p1, h1, p2, h2, x1, x2, ⟨RNG dummy arguments 55b⟩)
  call cir2dm (p1, p2, x1, x2, pol)
  pol00 = pol(0,0)
  do i1 = 0, 4
     do i2 = 0, 4
        pol(i1,i2) = pol(i1,i2) / pol00
      end do
  end do
end subroutine cir2gp
```

15 Luminosity

65a ⟨*Public procedures* 48d⟩+≡

```
  public :: cir2lm
```

65b ⟨*Procedures* 55e⟩+≡

```
  function cir2lm (p1, h1, p2, h2)
    integer :: p1, h1, p2, h2
    integer :: ic
    real(kind=double) :: cir2lm
    cir2lm = 0
    do ic = 1, c2p%nc
       if (       ((p1 .eq. c2p%pid1(ic)) .or. (p1 .eq. 0)) &
            .and. ((h1 .eq. c2p%pol1(ic)) .or. (h1 .eq. 0)) &
            .and. ((p2 .eq. c2p%pid2(ic)) .or. (p2 .eq. 0)) &
            .and. ((h2 .eq. c2p%pol2(ic)) .or. (h2 .eq. 0))) then
          cir2lm = cir2lm + c2p%lumi(ic)
       end if
    end do
  end function cir2lm
```

65c ⟨*Public Procedures (C)* 57a⟩+≡

```
  double
  circe2_lumi (circe2_channels *channels, int p1, int h1, int p2, int h2)
  {
    int i;
    double lumi;
    POINTER_PANIC(circe2_random_channel, channels, "channels");
```

```
  lumi = 0;
  for (i = 0; i < channels->n; i++) {
    circe2_channel *c = channels->ch[i];
    if (((p1 == c->d1->pid) || (p1 == 0))
        && ((h1 == c->d1->pol) || (h1 == 0))
        && ((p2 == c->d2->pid) || (p2 == 0))
        && ((h2 == c->d2->pol) || (h2 == 0)))
      lumi += c->lumi;
  }
  return lumi;
}
```

16 2D-Distribution

66a ⟨*Public procedures* 48d⟩+≡

```
public :: cir2dn
```

66b ⟨*Procedures* 55e⟩+≡

```
function cir2dn (p1, h1, p2, h2, yy1, yy2)
  integer :: p1, h1, p2, h2
  real(kind=double) :: yy1, yy2
  real(kind=double) :: y1, y2
  real(kind=double) :: cir2dn
  integer :: i, ic, i1, i2, ibot, itop
  ⟨Find ic for p1, h1, p2 and h2 58c⟩
  if (ic .le. 0) then
     cir2dn = 0
     return
  end if
  ⟨(y1, y2) ← (yy1, yy2) 69a⟩
  if (     (y1 .lt. c2p%yb1(0,ic)) .or. (y1 .gt. c2p%yb1(c2p%nb1(ic),ic)) &
      .or. (y2 .lt. c2p%yb2(0,ic)) .or. (y2 .gt. c2p%yb2(c2p%nb2(ic),ic))) then
     cir2dn = 0
     return
  end if
  ⟨Do a binary search for yb1(i1 − 1) ≤ y1 < yb1(i1) 69e⟩
  ⟨Do a binary search for yb2(i2 − 1) ≤ y2 < yb2(i2) 70a⟩
  cir2dn = c2p%val(i1,i2,ic)
  ⟨Apply Jacobian for y1 map 67b⟩
  ⟨Apply Jacobian for y2 map 68a⟩
  ⟨Apply Jacobian for triangle map 69c⟩
end function cir2dn
```

67a ⟨*Public Procedures (C)* 57a⟩+≡

```
double
circe2_distribution (circe2_channels *channels,
                     int p1, int h1, int p2, int h2,
                     double yy1, double yy2)
{
  circe2_channel *ch;
  int i, i1, i2;
  double y1, y2, d;
  POINTER_PANIC(circe2_random_channel, channels, "channels");
  ch = circe2_find_channel (channels, p1, h1, p2, h2);
  if (ch == NULL)
    return 0.0;
  ⟨(y1,y2) ← (yy1,yy2) (C) 69b⟩
  if ((y1 < ch->d1->y[0]) || (y1 > ch->d1->y[ch->d1->n])
      || (y2 < ch->d2->y[0]) || (y2 > ch->d2->y[ch->d2->n]))
    return 0.0;
  i1 = binary_search (ch->d1->y, 0, ch->d1->n, y1);
  i2 = binary_search (ch->d2->y, 0, ch->d2->n, y2);
  d = ch->value[i1][i2];
  ⟨Apply Jacobian for y1 map (C) 67c⟩
  ⟨Apply Jacobian for y2 map (C) 68b⟩
  ⟨Apply Jacobian for triangle map (C) 69d⟩
  return d;
}
```

cf. (53)

67b ⟨*Apply Jacobian for* y_1 *map* 67b⟩≡

```
      if (c2p%map1(i1,ic) .eq. 0) then
      else if (c2p%map1(i1,ic) .eq. 1) then
         cir2dn = cir2dn * c2p%b1(i1,ic) / (c2p%a1(i1,ic)*c2p%alpha1(i1,ic)) &
              * (c2p%b1(i1,ic)*(y1-c2p%eta1(i1,ic)))**(1/c2p%alpha1(i1,ic)-1)
      else if (c2p%map1(i1,ic) .eq. 2) then
         cir2dn = cir2dn * c2p%b1(i1,ic)**2 &
              / ((y1-c2p%eta1(i1,ic))**2 + c2p%a1(i1,ic)**2)
      else
         write (*, '(A,I3)') &
              'circe2: internal error: invalid map: ', c2p%map1(i1,ic)
         stop
      end if
```

67c ⟨*Apply Jacobian for* y_1 *map (C)* 67c⟩≡

```
  switch (ch->d1->map[i1]) {
  case 0:
```

```
  /* identity */
case 1:
  d = d * ch->d1->b[i1] / (ch->d1->a[i1] * ch->d1->alpha[i1])
        * pow (ch->d1->b[i1] * (y1 - ch->d1->eta[i1]), 1/ch->d1->alpha[i1] - 1);
case 2:
  d = d * ch->d1->b[i1] * ch->d1->b[i1]
        / ((y1 - ch->d1->eta[i1]) * (y1 - ch->d1->eta[i1])
             + ch->d1->a[i1] * ch->d1->a[i1]);
default:
  fprintf (stderr, "circe2: internal error: invalid map: %d\n", ch->d1->map[i1]);
  exit (-1);
}
```

68a ⟨*Apply Jacobian for* y_2 *map* 68a⟩≡

```
   if (c2p%map2(i2,ic) .eq. 0) then
   else if (c2p%map2(i2,ic) .eq. 1) then
      cir2dn = cir2dn * c2p%b2(i2,ic) / (c2p%a2(i2,ic)*c2p%alpha2(i2,ic)) &
           * (c2p%b2(i2,ic)*(y2-c2p%eta2(i2,ic)))**(1/c2p%alpha2(i2,ic)-1)
   else if (c2p%map2(i2,ic) .eq. 2) then
      cir2dn = cir2dn * c2p%b2(i2,ic)**2 &
           / ((y2-c2p%eta2(i2,ic))**2 + c2p%a2(i2,ic)**2)
   else
      write (*, '(A,I3)') &
           'circe2: internal error: invalid map: ', c2p%map2(i2,ic)
      stop
   end if
```

68b ⟨*Apply Jacobian for* y_2 *map (C)* 68b⟩≡

```
switch (ch->d2->map[i2]) {
case 0:
  /* identity */
case 1:
  d = d * ch->d2->b[i2] / (ch->d2->a[i2] * ch->d2->alpha[i2])
        * pow (ch->d2->b[i2] * (y2 - ch->d2->eta[i2]), 1/ch->d2->alpha[i2] - 1);
case 2:
  d = d * ch->d2->b[i2] * ch->d2->b[i2]
        / ((y2 - ch->d2->eta[i2]) * (y2 - ch->d2->eta[i2])
             + ch->d2->a[i2] * ch->d2->a[i2]);
default:
  fprintf (stderr, "circe2: internal error: invalid map: %d\n", ch->d2->map[i2]);
  exit (-1);
}
```

The triangle map

$$\begin{aligned}\tau : \{(x_1, x_2) \in [0,1] \times [0,1] : x_2 \le x_1\} &\to [0,1] \times [0,1] \\ (x_1, x_2) &\mapsto (y_1, y_2) = (x_1, x_1 x_2)\end{aligned} \tag{54}$$

and its inverse

$$\begin{aligned}\tau^{-1} : [0,1] \times [0,1] &\to \{(x_1, x_2) \in [0,1] \times [0,1] : x_2 \le x_1\} \\ (y_1, y_2) &\mapsto (x_1, x_2) = (y_1, y_2/y_1)\end{aligned} \tag{55}$$

69a ⟨(y1, y2) ← (yy1, yy2) 69a⟩≡

```
if (c2p%triang(ic)) then
   y1 = max (yy1, yy2)
   y2 = min (yy1, yy2) / y1
else
   y1 = yy1
   y2 = yy2
end if
```

69b ⟨(y1, y2) ← (yy1, yy2) *(C)* 69b⟩≡

```
if (ch->triangle) {
  y1 = max (yy1, yy2);
  y2 = min (yy1, yy2) / y1;
} else {
  y1 = yy1;
  y2 = yy2;
}
```

with the jacobian $J^*(y_1, y_2) = 1/y_2$ from

$$\mathrm{d}x_1 \wedge \mathrm{d}x_2 = \frac{1}{y_2} \cdot \mathrm{d}y_1 \wedge \mathrm{d}y_2 \tag{56}$$

69c ⟨*Apply Jacobian for triangle map* 69c⟩≡

```
if (c2p%triang(ic)) then
   cir2dn = cir2dn / y1
end if
```

69d ⟨*Apply Jacobian for triangle map (C)* 69d⟩≡

```
if (ch->triangle)
  d = d / y1;
```

We avoid name space polution and speed up things at the same time by explicit inlining:

69e ⟨*Do a binary search for* yb1(i1 − 1) ≤ y1 < yb1(i1) 69e⟩≡

```
ibot = 0
```

```
itop = c2p%nb1(ic)
do
  if (itop .le. (ibot + 1)) then
     i1 = ibot + 1
     exit
  else
     i1 = (ibot + itop) / 2
     if (y1 .lt. c2p%yb1(i1,ic)) then
        itop = i1
     else
        ibot = i1
     end if
  end if
end do
```

70a ⟨*Do a binary search for* yb2(i2 − 1) ≤ y2 < yb2(i2) 70a⟩≡

```
ibot = 0
itop = c2p%nb2(ic)
do
  if (itop .le. (ibot + 1)) then
     i2 = ibot + 1
  else
     i2 = (ibot + itop) / 2
     if (y2 .lt. c2p%yb2(i2,ic)) then
        itop = i2
     else
        ibot = i2
     end if
  end if
end do
```

70b ⟨*Public procedures* 48d⟩+≡

```
public :: cir2dm
```

70c ⟨*Procedures* 55e⟩+≡

```
subroutine cir2dm (p1, p2, x1, x2, pol)
  integer :: p1, p2
  real(kind=double) :: x1, x2
  real(kind=double) :: pol(0:3,0:3)
  ⟨Test support for density matrices 70d⟩
  print *, 'circe2: cir2dm not implemented yet!'
end subroutine cir2dm
```

70d ⟨*Test support for density matrices* 70d⟩≡

```
if (c2p%polspt .ne. POLGEN) then
```

```
      write (*, '(2A)') 'circe2: current beam ', &
           'description supports no density matrices'
      return
    end if
```

17 Reading Files

71a ⟨*Error codes for* `cir2ld` 71a⟩≡

```
integer, parameter :: EOK = 0
integer, parameter :: EFILE = -1
integer, parameter :: EMATCH = -2
integer, parameter :: EFORMT = -3
integer, parameter :: ESIZE = -4
```

71b ⟨*Well known constants (C)* 58b⟩+≡

```
#define CIRCE2_EOK     0
#define CIRCE2_EFILE  -1
#define CIRCE2_EMATCH -2
#define CIRCE2_EFORMT -3
#define CIRCE2_ESIZE  -4
```

71c ⟨*Public procedures* 48d⟩+≡

```
public :: cir2ld
```

71d ⟨*Procedures* 55e⟩+≡

```
subroutine cir2ld (file, design, roots, ierror)
  character(len=*) :: file, design
  real(kind=double) :: roots
  integer :: ierror
  character(len=72) :: buffer
  character(len=72) :: fdesgn
  character(len=72) :: fpolsp
  real(kind=double) :: froots
  integer :: lun, loaded, prefix
  logical :: match
  ⟨Local variables in cir2ld 74c⟩
  ⟨Error codes for cir2ld 71a⟩
  ⟨Find free logical unit for lun 79e⟩
  loaded = 0
  ⟨Open name for reading on lun 77c⟩
  if (ierror .gt. 0) then
     write (*, '(2A)') 'cir2ld: $Id: circe2.nw,v 1.56 2002/10/14 10:12:06 ohl Exp $
  end if
  prefix = index (design, '*') - 1
```

```
100 continue
      ⟨Skip comments until CIRCE2 78b⟩
      if (buffer(8:15) .eq. 'FORMAT#1') then
         read (lun, *)
         read (lun, *) fdesgn, froots
         ⟨Check if design and fdesgn do match 73⟩
         if (match .and. (abs (froots - roots) .le. 1d0)) then
            ⟨Load histograms 74a⟩
            loaded = loaded + 1
         else
            ⟨Skip data until ECRIC2 79a⟩
            goto 100
         end if
      else
         write (*, '(2A)') 'cir2ld: invalid format: ', buffer(8:72)
         ierror = EFORMT
         return
      end if
      ⟨Check for ECRIC2 79c⟩
      goto 100
  end subroutine cir2ld
```

72 ⟨Public Procedures (C) 57a⟩+≡

```
void
circe2_load (const char *file, const char *design, double roots, int *error)
{
  char buffer[73];
  char file_design[73];
  char file_polarization_support[73];
  double file_roots;
  int loaded;
  FILE *f;
  if (*error > 0)
    printf ("circe2: %s\n",
            "$Id: circe2.nw,v 1.56 2002/10/14 10:12:06 ohl Exp $");
  loaded = 0;
  f = fopen (file, "r");
  if (f == NULL) {
    fprintf (stderr, "circe2_load: can't open %s: %s\n", file, sys_errlist[errno]);
    *error = CIRCE2_EFILE;
    return;
  }
  while (loaded == 0) {
```

```
    ⟨Skip comments until CIRCE2 (C) 78c⟩
    if (strncmp ("FORMAT#1", buffer, 8) == 0) {
    }
    ⟨Check for ECRIC2 (C) 79d⟩
  }
}
/*
      integer prefix
      logical match
      prefix = index (design, '*') - 1
 100  continue
         ⟨Skip comments until CIRCE2 78b⟩
         if (buffer(8:15) .eq. 'FORMAT#1') then
            read (lun, *)
            read (lun, *) fdesgn, froots
            ⟨Check if design and fdesgn do match 73⟩
            if (match .and. (abs (froots - roots) .le. 1d0)) then
               ⟨Load histograms 74a⟩
               loaded = loaded + 1
            else
               ⟨Skip data until ECRIC2 79a⟩
               goto 100
            end if
         else
            write (*, '(2A)') 'cir2ld: invalid format: ', buffer(8:72)
            ierror = EFORMT
            return
         end if
         ⟨Check for ECRIC2 79c⟩
         goto 100
      end
*/
```

73 ⟨Check if design and fdesgn do match 73⟩≡

```
    match = .false.
    if (fdesgn .eq. design) then
       match = .true.
    else if (prefix .eq. 0) then
       match = .true.
    else if (prefix .gt. 0) then
       if (fdesgn(1:min(prefix,len(fdesgn))) &
            .eq. design(1:min(prefix,len(design)))) then
          match = .true.
       end if
```

```
   end if
```

74a ⟨*Load histograms* 74a⟩≡

```
   read (lun, *)
   read (lun, *) c2p%nc, fpolsp
   if (c2p%nc .gt. NCMAX) then
      write (*, '(A)') 'cir2ld: too many channels'
      ierror = ESIZE
      return
   end if
   ⟨Decode polarization support 74b⟩
   c2p%cwgt(0) = 0
   do ic = 1, c2p%nc
      ⟨Load channel ic 74d⟩
      ⟨Load division xb1 75c⟩
      ⟨Load division xb2 76a⟩
      ⟨Calculate yb1 76b⟩
      ⟨Calculate yb2 76c⟩
      ⟨Load weights wgt and val 77a⟩
   end do
   do ic = 1, c2p%nc
      c2p%cwgt(ic) = c2p%cwgt(ic) / c2p%cwgt(c2p%nc)
   end do
```

74b ⟨*Decode polarization support* 74b⟩≡

```
   if (      (fpolsp(1:1).eq.'a') &
        .or. (fpolsp(1:1).eq.'A')) then
      c2p%polspt = POLAVG
   else if (      (fpolsp(1:1).eq.'h') &
              .or. (fpolsp(1:1).eq.'H')) then
      c2p%polspt = POLHEL
   else if (      (fpolsp(1:1).eq.'d') &
              .or. (fpolsp(1:1).eq.'D')) then
      c2p%polspt = POLGEN
   else
      write (*, '(A,I5)') &
           'cir2ld: invalid polarization support: ', fpolsp
      ierror = EFORMT
      return
   end if
```

74c ⟨*Local variables in* cir2ld 74c⟩≡

```
   integer :: i, ic
```

74d ⟨*Load channel* ic 74d⟩≡

```
   read (lun, *)
```

```
read (lun, *) c2p%pid1(ic), c2p%pol1(ic), &
      c2p%pid2(ic), c2p%pol2(ic), c2p%lumi(ic)
c2p%cwgt(ic) = c2p%cwgt(ic-1) + c2p%lumi(ic)
⟨Check polarization support 75a⟩
```

75a ⟨*Check polarization support* 75a⟩≡

```
if (c2p%polspt .eq. POLAVG &
     .and. (      (c2p%pol1(ic) .ne. 0)  &
              .or. (c2p%pol2(ic) .ne. 0))) then
   write (*, '(A)') 'cir2ld: expecting averaged polarization'
   ierror = EFORMT
   return
else if (c2p%polspt .eq. POLHEL &
     .and. (      (c2p%pol1(ic) .eq. 0)       &
              .or. (c2p%pol2(ic) .eq. 0))) then
   write (*, '(A)') 'cir2ld: expecting helicities'
   ierror = EFORMT
   return
else if (c2p%polspt .eq. POLGEN) then
   write (*, '(A)') 'cir2ld: general polarizations not supported yet'
   ierror = EFORMT
   return
else if (c2p%polspt .eq. POLGEN &
     .and. (      (c2p%pol1(ic) .ne. 0)       &
              .or. (c2p%pol2(ic) .ne. 0))) then
   write (*, '(A)') 'cir2ld: expecting pol = 0'
   ierror = EFORMT
   return
end if
```

75b ⟨*Load channel* ic 74d⟩+≡

```
read (lun, *)
read (lun, *) c2p%nb1(ic), c2p%nb2(ic), c2p%triang(ic)
if ((c2p%nb1(ic) .gt. NBMAX) .or. (c2p%nb2(ic) .gt. NBMAX)) then
   write (*, '(A)') 'cir2ld: too many bins'
   ierror = ESIZE
   return
end if
```

75c ⟨*Load division* xb1 75c⟩≡

```
read (lun, *)
read (lun, *) c2p%xb1(0,ic)
do i1 = 1, c2p%nb1(ic)
   read (lun, *) c2p%xb1(i1,ic), c2p%map1(i1,ic), c2p%alpha1(i1,ic), &
       c2p%xi1(i1,ic), c2p%eta1(i1,ic), c2p%a1(i1,ic), c2p%b1(i1,ic)
```

```
end do
```

76a ⟨*Load division* xb2 76a⟩≡

```
read (lun, *)
read (lun, *) c2p%xb2(0,ic)
do i2 = 1, c2p%nb2(ic)
   read (lun, *) c2p%xb2(i2,ic), c2p%map2(i2,ic), c2p%alpha2(i2,ic), &
       c2p%xi2(i2,ic), c2p%eta2(i2,ic), c2p%a2(i2,ic), c2p%b2(i2,ic)
end do
```

The boundaries are guaranteed to be fixed points of the maps only if the boundaries are not allowed to float. This doesn't affect the unweighted events, because they never see the codomain grid, but distribution would be distorted significantly. In the following sums `i1` and `i2` run over the maps, while `i` runs over the boundaries.

An alternative would be to introduce sentinels `alpha1(0,:)`, `xi1(0,:)`, etc.

76b ⟨*Calculate* yb1 76b⟩≡

```
do i = 0, c2p%nb1(ic)
   i1 = max (i, 1)
   if (c2p%map1(i1,ic) .eq. 0) then
      c2p%yb1(i,ic) = c2p%xb1(i,ic)
   else if (c2p%map1(i1,ic) .eq. 1) then
      c2p%yb1(i,ic) = &
           (c2p%a1(i1,ic) &
             * (c2p%xb1(i,ic)-c2p%xi1(i1,ic)))**c2p%alpha1(i1,ic) &
                / c2p%b1(i1,ic) + c2p%eta1(i1,ic)
   else if (c2p%map1(i1,ic) .eq. 2) then
      c2p%yb1(i,ic) = c2p%a1(i1,ic) &
           * tan(c2p%a1(i1,ic)/c2p%b1(i1,ic)**2 &
                 * (c2p%xb1(i,ic)-c2p%xi1(i1,ic))) &
             + c2p%eta1(i1,ic)
   else
      write (*, '(A,I3)') 'cir2ld: invalid map: ', c2p%map1(i1,ic)
      ierror = EFORMT
      return
   end if
end do
```

76c ⟨*Calculate* yb2 76c⟩≡

```
do i = 0, c2p%nb2(ic)
   i2 = max (i, 1)
   if (c2p%map2(i2,ic) .eq. 0) then
```

```
            c2p%yb2(i,ic) = c2p%xb2(i,ic)
         else if (c2p%map2(i2,ic) .eq. 1) then
            c2p%yb2(i,ic) &
                 = (c2p%a2(i2,ic) &
                     * (c2p%xb2(i,ic)-c2p%xi2(i2,ic)))**c2p%alpha2(i2,ic) &
                        / c2p%b2(i2,ic) + c2p%eta2(i2,ic)
         else if (c2p%map2(i2,ic) .eq. 2) then
            c2p%yb2(i,ic) = c2p%a2(i2,ic) &
                 * tan(c2p%a2(i2,ic)/c2p%b2(i2,ic)**2 &
                        * (c2p%xb2(i,ic)-c2p%xi2(i2,ic))) &
                    + c2p%eta2(i2,ic)
         else
            write (*, '(A,I3)') 'cir2ld: invalid map: ', c2p%map2(i2,ic)
            ierror = EFORMT
            return
         end if
      end do
```

cf. (53)

77a ⟨*Load weights* `wgt` *and* `val` 77a⟩≡

```
      read (lun, *)
      c2p%wgt(0,ic) = 0
      do i = 1, c2p%nb1(ic)*c2p%nb2(ic)
         read (lun, *) w
         c2p%wgt(i,ic) = c2p%wgt(i-1,ic) + w
         ⟨(i1, i2) ← i 51g⟩
         c2p%val(i1,i2,ic) = w &
              / (  (c2p%xb1(i1,ic) - c2p%xb1(i1-1,ic)) &
                 * (c2p%xb2(i2,ic) - c2p%xb2(i2-1,ic)))
      end do
      c2p%wgt(c2p%nb1(ic)*c2p%nb2(ic),ic) = 1
```

77b ⟨*Local variables in* `cir2ld` 74c⟩+≡

```
      integer :: i1, i2
      real(kind=double) :: w
```

17.1 Auxiliary Code For Reading Files

77c ⟨*Open* `name` *for reading on* `lun` 77c⟩≡

```
        open (unit = lun, file = file, status = 'old', iostat = status)
        if (status .ne. 0) then
           write (*, '(2A)') 'cir2ld: can''t open ', file
           ierror = EFILE
           return
```

```
      end if
```

78a ⟨*Local variables in* cir2ld 74c⟩+≡

```
      integer :: status
```

78b ⟨*Skip comments until* CIRCE2 78b⟩≡

```
   20 continue
         read (lun, '(A)', end = 29) buffer
         if (buffer(1:6) .eq. 'CIRCE2') then
            goto 21
         else if (buffer(1:1) .eq. '!') then
            if (ierror .gt. 0) then
               write (*, '(A)') buffer
            end if
            goto 20
         end if
         write (*, '(A)') 'cir2ld: invalid file'
         ierror = EFORMT
         return
   29 continue
      if (loaded .gt. 0) then
         close (unit = lun)
         ierror = EOK
      else
         ierror = EMATCH
      end if
      return
   21 continue
```

78c ⟨*Skip comments until* CIRCE2 *(C)* 78c⟩≡

```
  while (1) {
    fgets (buffer, 72, f);
    if (strncmp ("ECRIC2", buffer, 6) == 0) {
      fclose (f);
      if (loaded)
        *error = CIRCE2_EOK;
      else {
        fprintf (stderr, "circe2_load: no match in %s\n", file);
        *error = CIRCE2_EMATCH;
      }
      return;
    } else if ((buffer[0] == '!') && (*error > 0))
      printf ("circe2: %s\n", buffer);
  }
  fprintf (stderr, "circe2_load: invalid format %s\n", file);
```

```
*error = CIRCE2_EFORMT;
return;
```

79a ⟨*Skip data until* ECRIC2 79a⟩≡

```
 101  continue
      read (lun, *) buffer
      if (buffer(1:6) .ne. 'ECRIC2') then
         goto 101
      end if
```

79b ⟨*Skip data until* ECRIC2 *(C)* 79b⟩≡

```
while (1) {
  fgets (buffer, 72, f);
  if (strncmp ("ECRIC2", buffer, 6) == 0)
    break;
}
```

79c ⟨*Check for* ECRIC2 79c⟩≡

```
      read (lun, '(A)') buffer
      if (buffer(1:6) .ne. 'ECRIC2') then
         write (*, '(A)') 'cir2ld: invalid file'
         ierror = EFORMT
         return
      end if
```

79d ⟨*Check for* ECRIC2 *(C)* 79d⟩≡

```
fgets (buffer, 72, f);
if (strncmp ("ECRIC2", buffer, 6) != 0) {
  fprintf (stderr, "circe2_load: invalid format %s\n", file);
  *error = CIRCE2_EFORMT;
  return;
}
```

79e ⟨*Find free logical unit for* lun 79e⟩≡

```
     do lun = 10, 99
        inquire (unit = lun, exist = exists, &
                opened = isopen, iostat = status)
        if ((status .eq. 0) .and. exists .and. .not.isopen) then
          goto 11
        end if
     end do
     write (*, '(A)') 'cir2ld: no free unit'
     ierror = ESIZE
     stop
11 continue
```

80a ⟨*Local variables in* cir2ld 74c⟩+≡
```
logical :: exists, isopen
```

18 Tests and Examples

80b ⟨circe2_sample.f90 80b⟩≡
```
! circe2_sample.f90 -- testing circe2
```
⟨*Copyleft notice* 45d⟩
```
  module circe2_sample_routines
    use kinds
    use circe2
    use tao_rng

    implicit none
    private
```
⟨*Sample procedures: public* 83b⟩
⟨write3_ascii*: public* 82c⟩
```
  contains
```
⟨*Sample procedures* 83c⟩
⟨write3_ascii 82d⟩
```
  end module circe2_sample_routines

  program circe2_sample
    use kinds
    use circe2
    use circe2_sample_routines
```
⟨implicit none 45a⟩
```
    integer :: i, p1, h1, p2, h2, n, ierror
    character(len=256) :: file, design, mode
    real(kind=double) :: roots, x1, x2, w
    read *, file, design, roots, p1, h1, p2, h2, mode, n
    ierror = 0
    call cir2ld (file, design, roots, ierror)
    if (ierror .ne. 0) then
       print *, 'sample: cir2ld failed!'
       stop
```

```
    end if
    if ((mode(1:1) .eq. 'w') .or. (mode(1:1) .eq. 'W')) then
       ⟨Generate n weighted events 82b⟩
    else
       ⟨Generate n unweighted events 82a⟩
    end if
  end program circe2_sample
```

81 ⟨circe2_sample_binary.f90 81⟩≡

```
! circe2_sample_binary.f90 -- testing circe2
⟨Copyleft notice 45d⟩
  module circe2_sample_routines_binary
    use kinds
    use circe2
    use tao_rng

    implicit none
    private

⟨Sample procedures: public 83b⟩

contains

⟨Sample procedures 83c⟩

  end module circe2_sample_routines_binary

  program circe2_sample_binary
    use kinds
    use circe2
    use circe2_sample_routines_binary

⟨implicit none 45a⟩

    integer :: i, p1, h1, p2, h2, n, ierror
    character(len=256) :: file, design, mode
    real(kind=double) :: roots, x1, x2, w
    read *, file, design, roots, p1, h1, p2, h2, mode, n
    ierror = 0
    call cir2ld (file, design, roots, ierror)
    if (ierror .ne. 0) then
       print *, 'sample: cir2ld failed!'
       stop
```

```
    end if
    if ((mode(1:1) .eq. 'w') .or. (mode(1:1) .eq. 'W')) then
      ⟨Generate n weighted events 82b⟩
    else
      ⟨Generate n unweighted events 82a⟩
    end if
  end program circe2_sample_binary
```

Generation of unweighted events is `Circe`'s home turf

82a ⟨*Generate* `n` *unweighted events* 82a⟩≡

```
    do i = 1, n
      call cir2gn (p1, h1, p2, h2, x1, x2, random)
      call write3 (x1, x2, 1d0)
    end do
```

while generation of weighted events without any importance sampling is slightly abusive and only useful for checking `cir2dn`.

82b ⟨*Generate* `n` *weighted events* 82b⟩≡

```
    do i = 1, n
      call random (x1)
      call random (x2)
      w = cir2dn (p1, h1, p2, h2, x1, x2)
      call write3 (x1, x2, w)
    end do
```

We could have written `print *, x, y, w` immediately, but the seperate `write3` allows us to create ASCII and binary versions.

82c ⟨`write3_ascii`*: public* 82c⟩≡

```
    public :: write3
```

82d ⟨`write3_ascii` 82d⟩≡

```
  subroutine write3 (x, y, w)
    real(kind=double), intent(in) :: x, y, w
    print *, x, y, w
  end subroutine write3
```

This is not necessarily portable, but the only way to reliably write binary files without *any* markers uses C. Binary I/O is useful because—on my laptop—event generation takes about 10 % of the time used by formatted writing of three floating point numbers.

82e ⟨`write3_binary.c` 82e⟩≡

```
#include <stdio.h>
void write3_ (double *x, double *y, double *w)
{
```

```
    double buf[3]; buf[0] = *x; buf[1] = *y; buf[2] = *w;
    if (fwrite (buf, sizeof (double), 3, stdout) != 3) {
      fprintf (stdin, "write3: fwrite failed!\n");
      exit (1);
    }
  }
```

The following bare bones random number generator produces some correlations that have been observed in testing `Circe2`

83a ⟨*Unused sample procedures* 83a⟩≡

```
    subroutine random (r)
      real(kind=double) :: r
      integer, parameter :: M = 259200, A = 7141, C = 54773
      integer, save :: n = 0
      n = mod (n*A + C, M)
      r = real (n, kind=double) / real (M, kind=double)
    end subroutine random
```

therfore, it makes sense to call a more sophisticated one:

83b ⟨*Sample procedures: public* 83b⟩≡

```
      public :: random
```

83c ⟨*Sample procedures* 83c⟩≡

```
    subroutine random (u)
      real(kind=double), intent(out) :: u
      call taornu (u)
    end subroutine random
```

19 Listing File Contents

Here's a small utility program for listing the contents of `Circe2` data files. It performs *no* verification and assumes that the file is in the correct format (cf. table 2).

83d ⟨`circe2ls.f90` 83d⟩≡

```
  ! circe2ls.f90 -- beam spectra for linear colliders and photon colliders
  ⟨Copyleft notice 45d⟩
    program circe2ls
      use kinds

      ⟨implicit none 45a⟩
```

```
    integer :: lun
    character(len=72) :: buffer
    character(len=72) :: file
    character(len=60) :: design, polspt
    integer :: pid1, hel1, pid2, hel2, nc
    real(kind=double) :: roots, lumi
    integer :: status
    logical :: exists, isopen
    integer :: ierror
    ⟨Error codes for cir2ld 71a⟩
    ⟨Find free logical unit for lun 79e⟩
    write (*, '(A)') 'enter name of Circe2 data file:'
    read (*, '(A)') buffer
    file = buffer
    open (unit = lun, file = file, status = 'old', iostat = status)
    if (status .ne. 0) then
       write (*, '(2A)') 'circe2: can''t open ', file
       stop
    end if
    write (*, '(A,1X,A)') 'file:', file
30  continue
    read (lun, '(A)', end = 39) buffer
    if (buffer(1:7) .eq. 'design,') then
       read (lun, *) design, roots
       read (lun, *)
       read (lun, *) nc, polspt
       ⟨Write design/beam data 84a⟩
       ⟨Write channel header 84b⟩
    else if (buffer(1:5) .eq. 'pid1,') then
       read (lun, *) pid1, hel1, pid2, hel2, lumi
       ⟨Write channel data 85a⟩
    end if
    goto 30
39  continue
  end program circe2ls
```

84a ⟨*Write design/beam data* 84a⟩≡

```
      write (*, '(2A)')     '        design: ', design
      write (*, '(A,F7.1)') '       sqrt(s): ', roots
      write (*, '(A,I3)')   '     #channels: ', nc
      write (*, '(2A)')     '  polarization: ', polspt
```

84b ⟨*Write channel header* 84b⟩≡

```
      write (*, '(4X,4(A5,2X),A)') &
```

```
        'pid#1', 'hel#1', 'pid#2', 'hel#2', &
        'luminosity / (10^32cm^-2sec^-1)'
```

85a ⟨*Write channel data* 85a⟩≡

```
    write (*, '(4X,4(I5,2X),F10.2,1X)') pid1, hel1, pid2, hel2, lumi
```

20 Static Data Sets

For those that despise reading files and prefer big ugly `block datas`:

85b ⟨circe2d.f90 85b⟩≡

```
! circe2d.f90 -- beam spectra for linear colliders and photon colliders
```

⟨*Copyleft notice* 45d⟩

⟨subroutine cir2lb 85c⟩

⟨block data cir2bd *template* 87b⟩

85c ⟨subroutine cir2lb 85c⟩≡

```
  subroutine cir2lb (design, roots, ierror)
   ⟨implicit none 45a⟩
    integer, parameter :: single = &
      & selected_real_kind (precision(1.), range(1.))
    integer, parameter :: double = &
      & selected_real_kind (precision(1._single) + 1, range(1._single) + 1)
    character(len=*), intent(in) :: design
    real(kind=double) :: roots
    integer :: ierror
    ⟨parameter declarations 51b⟩
    ⟨/cir2cd/ 86a⟩
    external cir2bd
    integer :: ib, ic, i1, i2
    integer :: loaded
    loaded = 0
    do ib = 1, bdnb
       if (design .eq. bddsgn(ib)) then
          print *, 'circe2: CIR2LB not available yet!'
          loaded = loaded + 1
       end if
    end do
    if (loaded .gt. 0) then
       ierror = 0
    else
```

```
        write (*, '(A)') 'circe2: no matching design'
        ierror = -1
      end if
    end subroutine cir2lb
```

86a ⟨/cir2cd/ 86a⟩≡

```
⟨8-byte aligned part of /cir2cd/ 86e⟩
⟨4-byte aligned part of /cir2cd/ 86c⟩
⟨1-byte aligned part of /cir2cd/ 86d⟩
```

⟨*Separator* 45c⟩

Three designs with two energies each would be `NBMAX = 6`, but let's be reasonable here while we're playing:

86b ⟨parameter *declarations* 51b⟩+≡

```
integer, parameter :: NBMMAX = 1
```

The actual number of parametersets:

86c ⟨*4-byte aligned part of* /cir2cd/ 86c⟩≡

```
integer :: bdnb
common /cir2cd/ bdnb
```

86d ⟨*1-byte aligned part of* /cir2cd/ 86d⟩≡

```
character(len=6), dimension(NBMMAX) :: bddsgn
common /cir2cd/ bddsgn
```

86e ⟨*8-byte aligned part of* /cir2cd/ 86e⟩≡

```
real(kind=double), dimension(0:NBMAX*NBMAX,NCMAX,NBMMAX) :: bdwgt
common /cir2cd/ bdwgt
real(kind=double), dimension(NBMAX,NBMAX,NCMAX,NBMMAX) :: bdval
common /cir2cd/ bdval
real(kind=double), dimension(0:NBMAX,NCMAX,NBMMAX) :: bdxb1
real(kind=double), dimension(0:NBMAX,NCMAX,NBMMAX) :: bdxb2
common /cir2cd/ bdxb1, bdxb2
real(kind=double), dimension(NCMAX,NBMMAX) :: bdlumi
common /cir2cd/ bdlumi
real(kind=double), dimension(0:NCMAX,NBMMAX) :: bdcwgt
common /cir2cd/ bdcwgt
real(kind=double), dimension(0:NBMAX,NCMAX,NBMMAX) :: bdyb1
real(kind=double), dimension(0:NBMAX,NCMAX,NBMMAX) ::  bdyb2
common /cir2cd/ bdyb1, bdyb2
real(kind=double), dimension(NBMAX,NCMAX,NBMMAX) :: bdalf1
real(kind=double), dimension(NBMAX,NCMAX,NBMMAX) :: bdalf2
real(kind=double), dimension(NBMAX,NCMAX,NBMMAX) :: bdxi1
real(kind=double), dimension(NBMAX,NCMAX,NBMMAX) :: bdxi2
real(kind=double), dimension(NBMAX,NCMAX,NBMMAX) :: bdeta1
```

```
real(kind=double), dimension(NBMAX,NCMAX,NBMMAX) :: bdeta2
real(kind=double), dimension(NBMAX,NCMAX,NBMMAX) :: bda1
real(kind=double), dimension(NBMAX,NCMAX,NBMMAX) :: bda2
real(kind=double), dimension(NBMAX,NCMAX,NBMMAX) :: bdb1
real(kind=double), dimension(NBMAX,NCMAX,NBMMAX) :: bdb2
common /cir2cd/ bdalf1, bdxi1, bdeta1, bda1, bdb1
common /cir2cd/ bdalf2, bdxi2, bdeta2, bda2, bdb2
```

87a ⟨*4-byte aligned part of* /cir2cd/ 86c⟩+≡

```
integer, dimension(NCMAX,NBMMAX) :: bdnb1, bdnb2
common /cir2cd/ bdnb1, bdnb2
logical, dimension(NCMAX,NBMMAX) :: bdtria
common /cir2cd/ bdtria
integer, dimension(NBMMAX) :: bdnc
common /cir2cd/ bdnc
integer, dimension(NCMAX,NBMMAX) :: bdpid1, bdpid2
integer, dimension(NCMAX,NBMMAX) :: bdpol1, bdpol2
common /cir2cd/ bdpid1, bdpol1, bdpid2, bdpol2
integer, dimension(NBMAX,NCMAX,NBMMAX) :: bdmap1, bdmap2
common /cir2cd/ bdmap1, bdmap2
```

In real life, this will be written by `circe2_tool`:

87b ⟨block data cir2bd *template* 87b⟩≡

```
block data cir2bd
  ⟨implicit none 45a⟩
  integer, parameter :: single = &
    & selected_real_kind (precision(1.), range(1.))
  integer, parameter :: double = &
    & selected_real_kind (precision(1._single) + 1, range(1._single) + 1)
  ⟨/cir2cm/ 87c⟩
  ⟨/cir2cd/ 86a⟩
  data bdnb /0/
  data bddsgn(1) /'TESLA '/
end block data cir2bd
```

87c ⟨/cir2cm/ 87c⟩≡

```
  ⟨parameter part of /cir2cm/ 88⟩
  ⟨8-byte aligned part of circe2 parameters 50b⟩
  common /cir2cm/ wgt
  common /cir2cm/ val
  common /cir2cm/ xb1, xb2
  common /cir2cm/ lumi
  common /cir2cm/ cwgt
  common /cir2cm/ yb1, yb2
  common /cir2cm/ alpha1, xi1, eta1, a1, b1
```

```
common /cir2cm/ alpha2, xi2, eta2, a2, b2
⟨4-byte aligned part of circe2 parameters 51a⟩
common /cir2cm/ nb1, nb2
common /cir2cm/ triang
common /cir2cm/ nc
common /cir2cm/ pid1, pol1, pid2, pol2
common /cir2cm/ map1, map2
common /cir2cm/ polspt
save /cir2cm/
```

88 ⟨parameter *part of* /cir2cm/ 88⟩≡

```
integer, parameter :: NBMAX = 100, NCMAX = 36
integer, parameter :: NBMMAX = 1
integer, parameter :: POLAVG = 1, POLHEL = 2, POLGEN = 3
```

A Making Grids

A.1 Interface of *Float*

```
module type T =
  sig
    type t
    (* Difference between 1.0 and the minimum float greater than 1.0 *)
    val epsilon : t
    val to_string : t → string
    val input_binary_float : in_channel → float
    val input_binary_floats : in_channel → float array → unit
  end
module Double : T with type t = float
```

A.2 Implementation of *Float*

```
open Printf
module type T =
  sig
    type t
    val epsilon : t
    val to_string : t → string
    val input_binary_float : in_channel → float
    val input_binary_floats : in_channel → float array → unit
  end
```

```
module Double =
  struct

    type t = float
```

Difference between 1.0 and the minimum float greater than 1.0

This is the hard coded value for double precision on Linux/Intel. We should determine this *machine dependent* value during configuration.

```
    let epsilon = 2.2204460492503131 · 10^-16
    let little_endian = true

    let to_string x =
      let s = sprintf "%.17E" x in
      for i = 0 to String.length s - 1 do
        let c = s.[i] in
        if c = 'e' ∨ c = 'E' then
          s.[i] ← 'D'
      done;
      s
```

Identity floatingpoint numbers that are indistinguishable from integers for more concise printing.

```
    type int_or_float =
      | Int of int
      | Float of float

    let float_min_int = float min_int
    let float_max_int = float max_int

    let soft_truncate x =
      let eps = 2.0 *. abs_float x *. epsilon in
      if x ≥ 0.0 then begin
        if x > float_max_int then
          Float x
        else if x -. floor x ≤ eps then
          Int (int_of_float x)
        else if ceil x -. x ≤ eps then
          Int (int_of_float x + 1)
        else
          Float x
      end else begin
        if x < float_min_int then
```

```
      Float x
    else if ceil x -. x ≤ eps then
      Int (int_of_float x)
    else if x -. floor x ≤ eps then
      Int (int_of_float x - 1)
    else
      Float x
  end

let to_short_string x =
  match soft_truncate x with
  | Int i → string_of_int i ^ "D0"
  | Float x → to_string x
```

Remark JRR: The function *float_of_string* was part of the `C` code of `O'Caml` from version 3.01 to version 3.07. In the transition from version 3.06 to 3.07 it was decided to be obsolete and superseded by the routines from the module *Int64*. There I make ThO's functions available and I comment out the external *C* function.

The following code uses the external *C* function:

```
let float_to_bytes x =
  let bytes = String.create 8 in
  let bits = Int64.bits_of_float x in
  let copy i j =
    String.unsafe_set bytes i
      (Char.chr (FF16 land (Int64.to_int (Int64.shift_right_logical bits (8×
j))))) in
  copy 7 0;
  copy 6 1;
  copy 5 2;
  copy 4 3;
  copy 3 4;
  copy 2 5;
  copy 1 6;
  copy 0 7;
  bytes
```

The following three functions make only use of the *Int64* module.

```
let float_of_bytes bytes =
  let copy i j =
    Int64.shift_left (Int64.of_int (Char.code (String.unsafe_get bytes j))) (8×
i) in
```

```
  Int64.float_of_bits
    (Int64.logor (copy 7 0)
       (Int64.logor (copy 6 1)
          (Int64.logor (copy 5 2)
             (Int64.logor (copy 4 3)
                (Int64.logor (copy 3 4)
                   (Int64.logor (copy 2 5)
                      (Int64.logor (copy 1 6)
                         (copy 0 7)))))))))
let input_binary_float ic =
  let buf = String.create 8 in
  really_input ic buf 0 8;
  float_of_bytes buf
let input_binary_floats ic array =
  let n = Array.length array in
  let bytes = 8 * n in
  let buf = String.create bytes in
  really_input ic buf 0 bytes;
  for i = 0 to n - 1 do
    let s = String.sub buf (8 * i) 8 in
    array.(i) <- float_of_bytes s
  done
```

Suggested by Xavier Leroy:

```
let output_float_big_endian oc f =
  let n = ref (Int64.bits_of_float f) in
  for i = 0 to 7 do
    output_byte oc (Int64.to_int (Int64.shift_right_logical !n 56));
    n := Int64.shift_left !n 8
  done
let output_float_little_endian oc f =
  let n = ref (Int64.bits_of_float f) in
  for i = 0 to 7 do
    output_byte oc (Int64.to_int !n);
    n := Int64.shift_right_logical !n 8
  done
let input_float_big_endian oc =
  let n = ref Int64.zero in
  for i = 0 to 7 do
    let b = input_byte oc in
```

```
      n := Int64.logor (Int64.shift_left !n 8) (Int64.of_int b)
    done;
    Int64.float_of_bits !n

  let input_float_little_endian oc =
    let n = ref Int64.zero in
    for i = 0 to 7 do
      let b = input_byte oc in
      n := Int64.logor !n (Int64.shift_left (Int64.of_int b) (i × 8))
    done;
    Int64.float_of_bits !n

end
```

A.3 Interface of *Diffmap*

```
module type T =
  sig
    type t
```

An invertible differentiable map is characterized by its domain $[x_{\min}, x_{\max}]$

```
    type domain
    val x_min : t → domain
    val x_max : t → domain
```

and codomain $[y_{\min}, y_{\max}]$

```
    type codomain
    val y_min : t → codomain
    val y_max : t → codomain
```

the map proper

$$\begin{aligned} \phi : [x_{\min}, x_{\max}] &\to [y_{\min}, y_{\max}] \\ x &\mapsto y = \phi(x) \end{aligned} \tag{57}$$

```
    val phi : t → domain → codomain
```

the inverse map

$$\begin{aligned} \phi^{-1} : [y_{\min}, y_{\max}] &\to [x_{\min}, x_{\max}] \\ y &\mapsto x = \phi^{-1}(y) \end{aligned} \tag{58}$$

```
    val ihp : t → codomain → domain
```

the jacobian of the map

$$\begin{aligned} J : [x_{\min}, x_{\max}] &\to \mathbf{R} \\ x &\mapsto J(x) = \frac{\mathrm{d}\phi}{\mathrm{d}x}(x) \end{aligned} \tag{59}$$

val *jac* : *t* → *domain* → *float*

and finally the jacobian of the inverse map

$$\begin{aligned} J^* : [y_{\min}, y_{\max}] &\to \mathbf{R} \\ y &\mapsto J^*(y) = \frac{\mathrm{d}\phi^{-1}}{\mathrm{d}y}(y) = \left(\frac{\mathrm{d}\phi}{\mathrm{d}x}(\phi^{-1}(y))\right)^{-1} \end{aligned} \tag{60}$$

val *caj* : *t* → *codomain* → *float*

with_domain map x_min x_max takes the map *map* and returns the 'same' map with the new domain $[x_{\min}, x_{\max}]$

val *with_domain* : *t* → *x_min* : *domain* → *x_max* : *domain* → *t*

There is also a convention for encoding the map so that it can be read by `Circe2`:

val *encode* : *t* → *string*
val *as_block_data_to_channel* : *t* →
out_channel → *string* → *int* → *int* → *int* → *unit*

end

For the application in `Circe2`, it suffices to consider real maps. Introducing *domain* and *codomain* does not make any difference for the typechecker as long as we only use *Diffmap.Real*, but it provides documentation and keeps the door for extensions open.

module type *Real* = *T* with type *domain* = *float* and type *codomain* = *float*

A.4 Testing Real Maps

module type *Test* =
sig
module *M* : *Real*
val *domain* : *M.t* → *unit*
val *inverse* : *M.t* → *unit*
val *jacobian* : *M.t* → *unit*
val *all* : *M.t* → *unit*
end

module *Make_Test* (*M* : *Real*) : *Test* with module *M* = *M*

A.5 Specific Real Maps

module *Id* :
  sig
    include *Real*

create x_min x_max y_min y_max creates an identity map $[x_{\min}, x_{\max}] \to [y_{\min}, y_{\max}]$.

$$\begin{aligned} \iota : [x_{\min}, x_{\max}] &\to [x_{\min}, x_{\max}] \\ x &\mapsto \iota(x) = x \end{aligned} \tag{61}$$

Default values for *x_min* and *x_max* are *y_min* and *y_max*, respectively. Indeed, they are the only possible values and other values raise an exception.

    val *create* :
      ?x_min : *domain* → *?x_max* : *domain* → *codomain* → *codomain* → *t*
  end

module *Linear* :
  sig
    include *Real*

create x_min x_max y_min y_max creates a linear map $[x_{\min}, x_{\max}] \to [y_{\min}, y_{\max}]$. The parameters a and b are determined from domain and codomain.

$$\begin{aligned} \lambda_{a,b} : [x_{\min}, x_{\max}] &\to [y_{\min}, y_{\max}] \\ x &\mapsto \lambda_{a,b}(x) = ax + b \end{aligned} \tag{62}$$

Default values for *x_min* and *x_max* are *y_min* and *y_max*, respectively.

    val *create* :
      ?x_min : *domain* → *?x_max* : *domain* → *codomain* → *codomain* → *t*
  end

module *Power* :
  sig
    include *Real*

create alpha eta x_min x_max y_min y_max creates a power map $[x_{\min}, x_{\max}] \to [y_{\min}, y_{\max}]$. The parameters ξ, a and b are determined from α, η, domain and codomain.

$$\begin{aligned} \psi_{a,b}^{\alpha,\xi,\eta} : [x_{\min}, x_{\max}] &\to [y_{\min}, y_{\max}] \\ x &\mapsto \psi_{a,b}^{\alpha,\xi,\eta}(x) = \frac{1}{b}(a(x-\xi))^{\alpha} + \eta \end{aligned} \tag{63}$$

Default values for *x_min* and *x_max* are *y_min* and *y_max*, respectively.

```
    val create : alpha :float → eta :float →
      ?x_min : domain → ?x_max : domain → codomain →
codomain → t
  end

module Resonance :
  sig
    include Real
```

create eta a x_min x_max y_min y_max creates a resonance map $[x_{\min}, x_{\max}] \to [y_{\min}, y_{\max}]$.

$$\begin{aligned} \rho_{a,b}^{\xi,\eta} : [x_{\min}, x_{\max}] &\to [y_{\min}, y_{\max}] \\ x &\mapsto \rho_{a,b}^{\xi,\eta}(x) = a \tan\left(\frac{a}{b^2}(x-\xi)\right) + \eta \end{aligned} \tag{64}$$

The parameters ξ and b are determined from η, a, domain and codomain. Default values for *x_min* and *x_max* are *y_min* and *y_max*, respectively.

```
    val create : eta :float → a :float →
      ?x_min : domain → ?x_max : domain → codomain →
codomain → t
  end
```

A.6 Implementation of *Diffmap*

```
open Printf

module type T =
  sig
```

```
    type t
    type domain
    val x_min : t → domain
    val x_max : t → domain
    type codomain
    val y_min : t → codomain
    val y_max : t → codomain
    val phi : t → domain → codomain
    val ihp : t → codomain → domain
    val jac : t → domain → float
    val caj : t → codomain → float
    val with_domain : t → x_min : domain → x_max : domain → t
    val encode : t → string
    val as_block_data_to_channel : t →
      out_channel → string → int → int → int → unit
  end
module type Real = T with type domain = float and type codomain = float
```

A.7 Testing Real Maps

```
module type Test =
  sig
    module M : Real
    val domain : M.t → unit
    val inverse : M.t → unit
    val jacobian : M.t → unit
    val all : M.t → unit
  end
module Make_Test (M : Real) =
  struct
    module M = M
    let steps = 1000
    let epsilon = 1.0 · 10^-6
```

```
let diff ?(tolerance = 1.0 · 10^-13) x1 x2 =
  let d = (x1 -. x2) in
  if abs_float d < (abs_float x1 +. abs_float x2) *. tolerance then
    0.0
  else
    d

let derive x_min x_max f x =
  let xp = min x_max (x +. epsilon)
  and xm = max x_min (x -. epsilon) in
  (f xp -. f xm) /. (xp -. xm)

let domain m =
  let x_min = M.x_min m
  and x_max = M.x_max m
  and y_min = M.y_min m
  and y_max = M.y_max m in
  let x_min' = M.ihp m y_min
  and x_max' = M.ihp m y_max
  and y_min' = M.phi m x_min
  and y_max' = M.phi m x_max in
  printf "␣␣␣f:␣[%g,%g]␣->␣[%g,%g]␣([%g,%g])\n"
    x_min x_max y_min' y_max' (diff y_min' y_min) (diff y_max' y_max);
  printf "f^-1:␣[%g,%g]␣->␣[%g,%g]␣([%g,%g])\n"
    y_min y_max x_min' x_max' (diff x_min' x_min) (diff x_max' x_max)

let inverse m =
  let x_min = M.x_min m
  and x_max = M.x_max m
  and y_min = M.y_min m
  and y_max = M.y_max m in
  for i = 1 to steps do
    let x = x_min +. Random.float (x_max -. x_min)
    and y = y_min +. Random.float (y_max -. y_min) in
    let x' = M.ihp m y
    and y' = M.phi m x in
    let x'' = M.ihp m y'
    and y'' = M.phi m x' in
    let dx = diff x'' x
    and dy = diff y'' y in
    if dx ≠ 0.0 then
      printf "f^-1␣o␣f␣␣␣:␣%g␣->␣%g␣->␣%g␣(%g)\n" x y' x'' dx;
    if dy ≠ 0.0 then
```

```
      printf "␣␣␣f␣o␣f^-1:␣%g␣->␣%g␣->␣%g␣(%g)\n" y x' y'' dy
    done

  let jacobian m =
    let x_min = M.x_min m
    and x_max = M.x_max m
    and y_min = M.y_min m
    and y_max = M.y_max m in
    for i = 1 to steps do
      let x = x_min +. Random.float (x_max -. x_min)
      and y = y_min +. Random.float (y_max -. y_min) in
      let jac_x' = derive x_min x_max (M.phi m) x
      and jac_x = M.jac m x
      and inv_jac_y' = derive y_min y_max (M.ihp m) y
      and inv_jac_y = M.caj m y in
      let dj = diff ~tolerance:1.0e-9 jac_x' jac_x
      and dij = diff ~tolerance:1.0e-9 inv_jac_y' inv_jac_y in
      if dj <> 0.0 then
        printf "dy/dx:␣%g␣->␣%g␣(%g)\n" x jac_x' dj;
      if dij <> 0.0 then
        printf "dx/dy:␣%g␣->␣%g␣(%g)\n" y inv_jac_y' dij
    done

  let all m =
    printf "phi(domain)␣=␣codomain␣and␣phi(codomain)␣=␣domain";
    domain m;
    printf "ihp␣o␣phi␣=␣id␣(domain)␣and␣phi␣o␣ihp␣=␣id(codomain)";
    inverse m;
    printf "jacobian";
    jacobian m

end
```

A.8 Specific Real Maps

```
module Id =
  struct

    type domain = float
    type codomain = float
```

```
type t =
    { x_min : domain;
      x_max : domain;
      y_min : codomain;
      y_max : codomain;
      phi : float → float;
      ihp : float → float;
      jac : float → float;
      caj : float → float }

let encode m = "0␣1␣0␣0␣1␣1"

let as_block_data_to_channel m oc tag i n_ch n_beam =
  fprintf oc "␣␣␣␣␣␣data␣bdmap%s(%d,%d,%d)␣/0/\n" tag i n_ch n_beam;
  fprintf oc "␣␣␣␣␣␣data␣bdalf%s(%d,%d,%d)␣/1D0/\n" tag i n_ch n_beam;
  fprintf oc "␣␣␣␣␣␣data␣bdxi%s(%d,%d,%d)␣/0D0/\n" tag i n_ch n_beam;
  fprintf oc "␣␣␣␣␣␣data␣bdeta%s(%d,%d,%d)␣/0D0/\n" tag i n_ch n_beam;
  fprintf oc "␣␣␣␣␣␣data␣bda%s(%d,%d,%d)␣/1D0/\n" tag i n_ch n_beam;
  fprintf oc "␣␣␣␣␣␣data␣bdb%s(%d,%d,%d)␣/1D0/\n" tag i n_ch n_beam

let closure ~x_min ~x_max ~y_min ~y_max =
  let phi x = x
  and ihp y = y
  and jac x = 1.0
  and caj y = 1.0 in
  { x_min = x_min;
    x_max = x_max;
    y_min = y_min;
    y_max = y_max;
    phi = phi;
    ihp = ihp;
    jac = jac;
    caj = caj }

let idmap ~x_min ~x_max ~y_min ~y_max =
  if x_min ≠ y_min ∧ x_max ≠ y_max then
    invalid_arg "Diffmap.Id.idmap"
  else
    closure ~x_min ~x_max ~y_min ~y_max

let with_domain m ~x_min ~x_max =
  idmap ~x_min ~x_max ~y_min : m.y_min ~y_max : m.y_max
```

```
    let create ?x_min ?x_max y_min y_max =
      idmap
        ~x_min : (match x_min with Some x -> x | None -> y_min)
        ~x_max : (match x_max with Some x -> x | None -> y_max)
        ~y_min ~y_max

    let x_min m = m.x_min
    let x_max m = m.x_max
    let y_min m = m.y_min
    let y_max m = m.y_max

    let phi m = m.phi
    let ihp m = m.ihp
    let jac m = m.jac
    let caj m = m.caj

  end

module Linear =
  struct

    type domain = float
    type codomain = float

    type t =
        { x_min : domain;
          x_max : domain;
          y_min : codomain;
          y_max : codomain;
          a : float;
          b : float;
          phi : domain -> codomain;
          ihp : codomain -> domain;
          jac : domain -> float;
          caj : codomain -> float }

    let encode m = failwith "Diffmap.Linear: not used in Circe2"
    let as_block_data_to_channel m oc tag i n_ch n_beam =
      failwith "Diffmap.Linear: not used in Circe2"

    let closure ~x_min ~x_max ~y_min ~y_max ~a ~b =
```

$$x \mapsto \lambda_{a,b}(x) = ax + b \tag{65}$$

```
      let phi x = a *. x +. b
```

$$y \mapsto (\lambda_{a,b})^{-1}(y) = \frac{y-b}{a} \tag{66}$$

```
  and ihp y = (y -. b) /. a

  and jac x = a
  and caj y = 1.0 /. a in

  { x_min = x_min;
    x_max = x_max;
    y_min = y_min;
    y_max = y_max;
    a = a;
    b = b;
    phi = phi;
    ihp = ihp;
    jac = jac;
    caj = caj }

let linearmap ~x_min ~x_max ~y_min ~y_max =
  let delta_x = x_max -. x_min
  and delta_y = y_max -. y_min in
  let a = delta_y /. delta_x
  and b = (y_min *. x_max -. y_max *. x_min) /. delta_x in
  closure ~x_min ~x_max ~y_min ~y_max ~a ~b

let with_domain m ~x_min ~x_max =
  linearmap ~x_min ~x_max ~y_min : m.y_min ~y_max : m.y_max

let create ?x_min ?x_max y_min y_max =
  linearmap
    ~x_min : (match x_min with Some x -> x | None -> y_min)
    ~x_max : (match x_max with Some x -> x | None -> y_max)
    ~y_min ~y_max

let x_min m = m.x_min
let x_max m = m.x_max
let y_min m = m.y_min
let y_max m = m.y_max

let phi m = m.phi
let ihp m = m.ihp
let jac m = m.jac
let caj m = m.caj
```

```
  end

module Power =
  struct

    type domain = float
    type codomain = float

    type t =
        { x_min : domain;
          x_max : domain;
          y_min : codomain;
          y_max : codomain;
          alpha : float;
          xi : float;
          eta : float;
          a : float;
          b : float;
          phi : domain -> codomain;
          ihp : codomain -> domain;
          jac : domain -> float;
          caj : codomain -> float }

    let encode m =
      sprintf "1 %s %s %s %s %s"
        (Float.Double.to_string m.alpha)
        (Float.Double.to_string m.xi)
        (Float.Double.to_string m.eta)
        (Float.Double.to_string m.a)
        (Float.Double.to_string m.b)

    let as_block_data_to_channel m oc tag i n_ch n_beam =
      fprintf oc "       data bdmap%s(%d,%d,%d) /1/\n"
        tag i n_ch n_beam;
      fprintf oc "       data bdalf%s(%d,%d,%d) /%s/\n"
        tag i n_ch n_beam (Float.Double.to_string m.alpha);
      fprintf oc "       data bdxi%s(%d,%d,%d) /%s/\n"
        tag i n_ch n_beam (Float.Double.to_string m.xi);
      fprintf oc "       data bdeta%s(%d,%d,%d) /%s/\n"
        tag i n_ch n_beam (Float.Double.to_string m.eta);
      fprintf oc "       data bda%s(%d,%d,%d) /%s/\n"
        tag i n_ch n_beam (Float.Double.to_string m.a);
      fprintf oc "       data bdb%s(%d,%d,%d) /%s/\n"
        tag i n_ch n_beam (Float.Double.to_string m.b)
```

```
let closure ~x_min ~x_max ~y_min ~y_max ~alpha ~xi ~eta ~a ~b =
```

$$x \mapsto \psi_{a,b}^{\alpha,\xi,\eta}(x) = \frac{1}{b}(a(x-\xi))^{\alpha} + \eta \tag{67}$$

```
  let phi x =
    (a *. (x -. xi)) ** alpha /. b +. eta
```

$$y \mapsto (\psi_{a,b}^{\alpha,\xi,\eta})^{-1}(y) = \frac{1}{a}(b(y-\eta))^{1/\alpha} + \xi \tag{68}$$

```
  and ihp y =
    (b *. (y -. eta)) ** (1.0 /. alpha) /. a +. xi
```

$$\frac{\mathrm{d}y}{\mathrm{d}x}(x) = \frac{a\alpha}{b}(a(x-\xi))^{\alpha-1} \tag{69}$$

```
  and jac x =
    a *. alpha *. (a *. (x -. xi)) ** (alpha -. 1.0) /. b
```

$$\frac{\mathrm{d}x}{\mathrm{d}y}(y) = \frac{b}{a\alpha}(b(y-\eta))^{1/\alpha-1} \tag{70}$$

```
  and caj y =
    b *. (b *. (y -. eta)) ** (1.0 /. alpha -. 1.0) /. (a *. alpha) in
{ x_min = x_min;
  x_max = x_max;
  y_min = y_min;
  y_max = y_max;
  alpha = alpha;
  xi = xi;
  eta = eta;
  a = a;
  b = b;
  phi = phi;
  ihp = ihp;
  jac = jac;
  caj = caj }
```

$$a_i = \frac{(b_i(y_i-\eta_i))^{1/\alpha_i} - (b_i(y_{i-1}-\eta_i))^{1/\alpha_i}}{x_i - x_{i-1}} \tag{71a}$$

$$\xi_i = \frac{x_{i-1}|y_i - \eta_i|^{1/\alpha_i} - x_i|y_{i-1} - \eta_i|^{1/\alpha_i}}{|y_i - \eta_i|^{1/\alpha_i} - |y_{i-1} - \eta_i|^{1/\alpha_i}} \tag{71b}$$

The degeneracy (39) can finally be resolved by demanding $|b| = 1$ in (47a).

```
    let powermap ~x_min ~x_max ~y_min ~y_max ~alpha ~eta =
      let b =
        if eta ≤ y_min then
          1.
        else if eta ≥ y_max then
          -1.
        else
          invalid_arg "singular" in
      let pow y = (b *. (y -. eta)) ** (1. /. alpha) in
      let delta_pow = pow y_max -. pow y_min
      and delta_x = x_max -. x_min in
      let a = delta_pow /. delta_x
      and xi = (x_min *. pow y_max -. x_max *. pow y_min) /. delta_pow in
      closure ~x_min ~x_max ~y_min ~y_max ~alpha ~xi ~eta ~a ~b

    let with_domain m ~x_min ~x_max =
      powermap ~x_min ~x_max ~y_min : m.y_min ~y_max : m.y_max
        ~alpha : m.alpha ~eta : m.eta

    let create ~alpha ~eta ?x_min ?x_max y_min y_max =
      powermap
        ~x_min : (match x_min with Some x → x | None → y_min)
        ~x_max : (match x_max with Some x → x | None → y_max)
        ~y_min ~y_max ~alpha ~eta

    let x_min m = m.x_min
    let x_max m = m.x_max
    let y_min m = m.y_min
    let y_max m = m.y_max

    let phi m = m.phi
    let ihp m = m.ihp
    let jac m = m.jac
    let caj m = m.caj

  end

module Resonance =
  struct

    type domain = float
    type codomain = float
```

```
type t =
    { x_min : domain;
      x_max : domain;
      y_min : codomain;
      y_max : codomain;
      xi : float;
      eta : float;
      a : float;
      b : float;
      phi : domain -> codomain;
      ihp : codomain -> domain;
      jac : domain -> float;
      caj : codomain -> float }
let encode m =
  sprintf "2 0 %s %s %s %s"
    (Float.Double.to_string m.xi)
    (Float.Double.to_string m.eta)
    (Float.Double.to_string m.a)
    (Float.Double.to_string m.b)
let as_block_data_to_channel m oc tag i n_ch n_beam =
  fprintf oc "       data bdmap%s(%d,%d,%d) /2/\n"
    tag i n_ch n_beam;
  fprintf oc "       data bdalf%s(%d,%d,%d) /0D0/\n"
    tag i n_ch n_beam;
  fprintf oc "       data bdxi%s(%d,%d,%d) /%s/\n"
    tag i n_ch n_beam (Float.Double.to_string m.xi);
  fprintf oc "       data bdeta%s(%d,%d,%d) /%s/\n"
    tag i n_ch n_beam (Float.Double.to_string m.eta);
  fprintf oc "       data bda%s(%d,%d,%d) /%s/\n"
    tag i n_ch n_beam (Float.Double.to_string m.a);
  fprintf oc "       data bdb%s(%d,%d,%d) /%s/\n"
    tag i n_ch n_beam (Float.Double.to_string m.b)
let closure ~x_min ~x_max ~y_min ~y_max ~xi ~eta ~a ~b =
```

$$x \mapsto \rho_{a,b}^{\xi,\eta}(x) = a \tan\left(\frac{a}{b^2}(x-\xi)\right) + \eta \tag{72}$$

```
  let phi x = a *. tan (a *. (x -. xi) /. (b *. b)) +. eta
```

$$y \mapsto (\rho_{a,b}^{\xi,\eta})^{-1}(y) = \frac{b^2}{a}\mathrm{atan}\left(\frac{y-\eta}{a}\right) + \xi \tag{73}$$

```
and ihp y = b *. b *. (atan2 (y -. eta) a) /. a +. xi
```

$$\frac{\mathrm{d}y}{\mathrm{d}x}(x(y)) = \frac{1}{\dfrac{\mathrm{d}x}{\mathrm{d}y}(y)} = \left(\frac{b^2}{(y-\eta)^2 + a^2}\right)^{-1} \tag{74}$$

```
and caj y = b *. b /. ((y -. eta) ** 2.0 +. a *. a) in
let jac x = 1.0 /. caj (phi x) in
{ x_min = x_min;
  x_max = x_max;
  y_min = y_min;
  y_max = y_max;
  xi = xi;
  eta = eta;
  a = a;
  b = b;
  phi = phi;
  ihp = ihp;
  jac = jac;
  caj = caj }
```

$$b_i = \sqrt{a_i \frac{x_i - x_{i-1}}{\operatorname{atan}\left(\frac{y_i - \eta_i}{a_i}\right) - \operatorname{atan}\left(\frac{y_{i-1} - \eta_i}{a_i}\right)}} \tag{75a}$$

$$\xi_i = \frac{x_{i-1}\operatorname{atan}\left(\frac{y_i - \eta_i}{a_i}\right) - x_i \operatorname{atan}\left(\frac{y_{i-1} - \eta_i}{a_i}\right)}{x_i - x_{i-1}} \tag{75b}$$

```
let resonancemap ~x_min ~x_max ~y_min ~y_max ~eta ~a =
  let arc y = atan2 (y -. eta) a in
  let delta_arc = arc y_max -. arc y_min
  and delta_x = x_max -. x_min in
  let b = sqrt (a *. delta_x /. delta_arc)
  and xi = (x_min *. arc y_max -. x_max *. arc y_min) /. delta_arc in
  closure ~x_min ~x_max ~y_min ~y_max ~xi ~eta ~a ~b

let with_domain m ~x_min ~x_max =
  resonancemap ~x_min ~x_max ~y_min : m.y_min ~y_max : m.y_max
    ~eta : m.eta ~a : m.a
```

```
    let create ~eta ~a ?x_min ?x_max y_min y_max =
      resonancemap
        ~x_min : (match x_min with Some x -> x | None -> y_min)
        ~x_max : (match x_max with Some x -> x | None -> y_max)
        ~y_min ~y_max ~eta ~a

    let x_min m = m.x_min
    let x_max m = m.x_max
    let y_min m = m.y_min
    let y_max m = m.y_max

    let phi m = m.phi
    let ihp m = m.ihp
    let jac m = m.jac
    let caj m = m.caj

  end
```

A.9 Interface of *Diffmaps*

A.10 Combined Differentiable Maps

```
module type T =
  sig
    include Diffmap.T
    val id : ?x_min : domain -> ?x_max : domain -> codomain ->
codomain -> t
  end

module type Real = T with type domain = float and type codomain = float

module type Default =
  sig

    include Real

    val power : alpha :float -> eta :float ->
      ?x_min : domain -> ?x_max : domain -> codomain -> codomain ->
t
    val resonance : eta :float -> a :float ->
      ?x_min : domain -> ?x_max : domain -> codomain -> codomain ->
t

  end

module Default : Default
```

A.11 Implementation of *Diffmaps*

```
module type T =
  sig
    include Diffmap.T
    val id : ?x_min : domain → ?x_max : domain → codomain →
codomain → t
  end

module type Real = T with type domain = float and type codomain = float

module type Default =
  sig

    include Real

    val power : alpha :float → eta :float →
      ?x_min : domain → ?x_max : domain → codomain → codomain →
t
    val resonance : eta :float → a :float →
      ?x_min : domain → ?x_max : domain → codomain → codomain →
t

  end

module Default =
  struct

    type domain = float
    type codomain = float

    type t =
        { encode : string;
          as_block_data_to_channel :
            out_channel → string → int → int → int → unit;
          with_domain : x_min : domain → x_max : domain → t;
          x_min : domain;
          x_max : domain;
          y_min : codomain;
          y_max : codomain;
          phi : domain → codomain;
          ihp : codomain → domain;
          jac : domain → float;
          caj : codomain → float }
```

```
let encode m = m.encode
let as_block_data_to_channel m = m.as_block_data_to_channel
let with_domain m = m.with_domain

let x_min m = m.x_min
let x_max m = m.x_max
let y_min m = m.y_min
let y_max m = m.y_max

let phi m = m.phi
let ihp m = m.ihp
let jac m = m.jac
let caj m = m.caj

let rec id ?x_min ?x_max y_min y_max =
  let m = Diffmap.Id.create ?x_min ?x_max y_min y_max in
  let with_domain ~x_min ~x_max =
    id ~x_min ~x_max y_min y_max in
  { encode = Diffmap.Id.encode m;
    as_block_data_to_channel = Diffmap.Id.as_block_data_to_channel m;
    with_domain = with_domain;
    x_min = Diffmap.Id.x_min m;
    x_max = Diffmap.Id.x_max m;
    y_min = Diffmap.Id.y_min m;
    y_max = Diffmap.Id.y_max m;
    phi = Diffmap.Id.phi m;
    ihp = Diffmap.Id.ihp m;
    jac = Diffmap.Id.jac m;
    caj = Diffmap.Id.caj m }

let rec power ~alpha ~eta ?x_min ?x_max y_min y_max =
  let m = Diffmap.Power.create ~alpha ~eta ?x_min ?x_max y_min y_max in
  let with_domain ~x_min ~x_max =
    power ~alpha ~eta ~x_min ~x_max y_min y_max in
  { encode = Diffmap.Power.encode m;
    as_block_data_to_channel = Diffmap.Power.as_block_data_to_channel m;
    with_domain = with_domain;
    x_min = Diffmap.Power.x_min m;
    x_max = Diffmap.Power.x_max m;
    y_min = Diffmap.Power.y_min m;
    y_max = Diffmap.Power.y_max m;
    phi = Diffmap.Power.phi m;
    ihp = Diffmap.Power.ihp m;
```

```
      jac = Diffmap.Power.jac m;
      caj = Diffmap.Power.caj m }
  let rec resonance ~eta ~a ?x_min ?x_max y_min y_max =
    let m = Diffmap.Resonance.create ~eta ~a ?x_min ?x_max y_min y_max in
    let with_domain ~x_min ~x_max =
      resonance ~eta ~a ~x_min ~x_max y_min y_max in
    { encode = Diffmap.Resonance.encode m;
      as_block_data_to_channel = Diffmap.Resonance.as_block_data_to_channel m;
      with_domain = with_domain;
      x_min = Diffmap.Resonance.x_min m;
      x_max = Diffmap.Resonance.x_max m;
      y_min = Diffmap.Resonance.y_min m;
      y_max = Diffmap.Resonance.y_max m;
      phi = Diffmap.Resonance.phi m;
      ihp = Diffmap.Resonance.ihp m;
      jac = Diffmap.Resonance.jac m;
      caj = Diffmap.Resonance.caj m }
end
```

A.12 Interface of *Division*

We have divisions (*Mono*) and divisions of divisions (*Poly*). Except for creation, they share the same interface (T), which can be used as a signature for functor arguments. In particular, both kinds of divisions can be used with the *Grid.Make* functor.

```
module type T =
  sig
    type t
```

Copy a division, allocating fresh arrays with identical contents.

```
    val copy : t → t
```

Using $\{x_0, x_1, \ldots, x_n\}$, find i, such that $x_i \leq x < x_{i+1}$. We need to export this, if we want to maintain additional histograms in user modules.

```
    val find : t → float → int
```

record d x f records the value f at coordinate x. NB: this function modifies d.

```
    val record : t → float → float → unit
```

VEGAS style rebinning. The default values for *power* and both *fixed_min*, *fixed_max* are 1.5 and false respectively.

val *rebin* : ?*power* :*float* → ?*fixed_min* :*bool* → ?*fixed_max* :*bool* → *t* → *t*

$J^*(y)$

Should this include the $1/\Delta y$?

val *caj* : *t* → *float* → *float*

val *n_bins* : *t* → *int*
val *bins* : *t* → *float array*
val *to_channel* : *out_channel* → *t* → *unit*
val *as_block_data_to_channel* : *out_channel* →
string → *int* → *int* → *t* → *unit*

end

exception *Out_of_range* of *float* × (*float* × *float*)
exception *Rebinning_failure* of *string*

A.12.1 Primary Divisions

module type *Mono* =
sig
include *T*

create bias n x_min x_max creates a division with n equidistant bins spanning $[x_{\min}, x_{\max}]$. The *bias* is a function that is multiplied with the weights for VEGAS/VAMP rebinning. It can be used to highlight the regions of phasespace that are expected to be most relevant in applications. The default is fun $x \to 1.0$, of course.

val *create* : ?*bias* : (*float* → *float*) → *int* → *float* → *float* → *t*

end

module *Mono* : *Mono*

A.12.2 Polydivisions

```
module type Poly =
  sig
    module M : Diffmaps.Real
    include T
```

create n x_min x_max intervals creates a polydivision of the interval from *x_min* to *x_max* described by the list of *intervals*, filling the gaps among intervals and between the intervals and the outer borders with an unmapped divisions with n bins each.

```
    val create : ?bias : (float → float) →
      (int × M.t) list → int → float → float → t
  end
module Make_Poly (M : Diffmaps.Real) : Poly with module M = M
module Poly : Poly
```

A.13 Implementation of *Division*

```
open Printf
let epsilon_100 = 100.0 *. Float.Double.epsilon
let equidistant n x_min x_max =
  if n ≤ 0 then
    invalid_arg "Division.equidistant: n <= 0"
  else
    let delta = (x_max -. x_min) /. (float n) in
    Array.init (n + 1) (fun i → x_min +. delta *. float i)
exception Out_of_range of float × (float × float)
exception Rebinning_failure of string
```

```
let find_raw d x =
  let n_max = Array.length d - 1 in
  let eps = epsilon_100 *. (d.(n_max) -. d.(0)) in
  let rec find' a b =
    if b <= a + 1 then
      a
    else
      let m = (a + b) / 2 in
      if x < d.(m) then
        find' a m
      else
        find' m b in
  if x < d.(0) -. eps || x > d.(n_max) +. eps then
    raise (Out_of_range (x, (d.(0), d.(n_max))))
  else if x <= d.(0) then
    0
  else if x >= d.(n_max) then
    n_max - 1
  else
    find' 0 n_max
module type T =
sig
    type t
    val copy : t -> t
    val find : t -> float -> int
    val record : t -> float -> float -> unit
    val rebin : ?power :float -> ?fixed_min :bool -> ?fixed_max :bool ->
 t -> t
    val caj : t -> float -> float
    val n_bins : t -> int
    val bins : t -> float array
    val to_channel : out_channel -> t -> unit
    val as_block_data_to_channel : out_channel ->
      string -> int -> int -> t -> unit
  end
```

A.13.1 Primary Divisions

```
module type Mono =
```

```
  sig
    include T
    val create : ?bias : (float -> float) -> int -> float -> float -> t
  end
module Mono (* : T *) =
  struct
    type t =
        { x : float array;
          mutable x_min : float;
          mutable x_max : float;
          n : int array;
          w : float array;
          w2 : float array;
          bias : float -> float }
    let copy d =
      { x = Array.copy d.x;
        x_min = d.x_min;
        x_max = d.x_max;
        n = Array.copy d.n;
        w = Array.copy d.w;
        w2 = Array.copy d.w2;
        bias = d.bias }
    let create ?(bias = fun x -> 1.0) n x_min x_max =
      { x = equidistant n x_min x_max;
        x_min = x_max;
        x_max = x_min;
        n = Array.create n 0;
        w = Array.create n 0.0;
        w2 = Array.create n 0.0;
        bias = bias }
    let bins d = d.x
    let n_bins d = Array.length d.x - 1
    let find d = find_raw d.x
    let normal_float x =
      match classify_float x with
      | FP_normal | FP_subnormal | FP_zero -> true
      | FP_infinite | FP_nan -> false
```

```
let report_denormal x f b what =
  eprintf
    "circe2: Division.record: ignoring %s (x=%g, f=%g, b=%g)\n"
    what x f b;
  flush stderr

let caj d x = 1.0

let record d x f =
  if x < d.x_min then
    d.x_min ← x;
  if x > d.x_max then
    d.x_max ← x;
  let i = find d x in
  d.n.(i) ← succ d.n.(i);
  let b = d.bias x in
  let w = f *. b in
  match classify_float w with
  | FP_normal | FP_subnormal | FP_zero →
      d.w.(i) ← d.w.(i) +. w;
      let w2 = f *. w in
      begin match classify_float w2 with
      | FP_normal | FP_subnormal | FP_zero →
          d.w2.(i) ← d.w2.(i) +. w2
      | FP_infinite → report_denormal x f b "w2 = [inf]"
      | FP_nan → report_denormal x f b "w2 = [nan]"
      end
  | FP_infinite → report_denormal x f b "w2 = [inf]"
  | FP_nan → report_denormal x f b "w2 = [nan]"
```

$$
\begin{aligned}
d_1 &\to \frac{1}{2}(d_1 + d_2) \\
d_2 &\to \frac{1}{3}(d_1 + d_2 + d_3) \\
&\dots \\
d_{n-1} &\to \frac{1}{3}(d_{n-2} + d_{n-1} + d_n) \\
d_n &\to \frac{1}{2}(d_{n-1} + d_n)
\end{aligned}
\tag{76}
$$

```
let smooth3 f =
  match Array.length f with
  | 0 → f
  | 1 → Array.copy f
  | 2 → Array.create 2 ((f.(0) +. f.(1)) /. 2.0)
  | n →
      let f' = Array.create n 0.0 in
      f'.(0) ← (f.(0) +. f.(1)) /. 2.0;
      for i = 1 to n − 2 do
        f'.(i) ← (f.(i − 1) +. f.(i) +. f.(i + 1)) /. 3.0
      done;
      f'.(n − 1) ← (f.(n − 2) +. f.(n − 1)) /. 2.0;
      f'
```

$$m_i = \left(\frac{\frac{\bar{f}_i \Delta x_i}{\sum_j \bar{f}_j \Delta x_j} - 1}{\ln\left(\frac{\bar{f}_i \Delta x_i}{\sum_j \bar{f}_j \Delta x_j}\right)} \right)^{\alpha} \tag{77}$$

```
let rebinning_weights' power fs =
  let sum_f = Array.fold_left (+.) 0.0 fs in
  if sum_f ≤ 0.0 then
    Array.create (Array.length fs) 1.0
  else
    Array.map (fun f →
      let f' = f /. sum_f in
      if f' < 1.0 · 10^−12 then
        0.
      else
        ((f' −. 1.0) /. (log f')) ** power) fs
```

The nested loops can be turned into recursions, of course. But arrays aren't purely functional anyway ...

```
let rebin' m x =
  let n = Array.length x − 1 in
  let x' = Array.create (n + 1) 0.0 in
  let sum_m = Array.fold_left (+.) 0.0 m in
  if sum_m ≤ 0.0 then
    Array.copy x
  else begin
    let step = sum_m /. (float n) in
    let k = ref 0
```

```
    and delta = ref 0.0 in
    x'.(0) ← x.(0);
    for i = 1 to n − 1 do
```

We increment k until another Δ (a. k. a. *step*) of the integral has been accumulated (cf. figure **??**).

```
      while !delta < step do
        incr k;
        delta := !delta +. m.(!k − 1)
      done;
```

Correct the mismatch.

```
      delta := !delta −. step;
```

Linearly interpolate the next bin boundary.

```
      x'.(i) ← x.(!k) −. (x.(!k) −. x.(!k − 1)) ∗. !delta /. m.(!k − 1);
      if x'.(i) < x'.(i − 1) then
        raise (Rebinning_failure
                 (sprintf "x(%d)=%g␣<␣x(%d)=%g" i x'.(i) (i−1) x'.(i−
1)))
    done;
    x'.(n) ← x.(n);
    x'
  end
```

Check that *x_min* and *x_max* are implemented correctly!!!!

One known problem is that the second outermost bins hinder the outermost bins from moving.

```
    let rebin ?(power = 1.5) ?(fixed_min = false) ?(fixed_max = false) d =
      let n = Array.length d.w in
      let x = rebin' (rebinning_weights' power (smooth3 d.w2)) d.x in
      if ¬ fixed_min then
        x.(0) ← (x.(0) +. min d.x_min x.(1)) /. 2.;
      if ¬ fixed_max then
        x.(n) ← (x.(n) +. max d.x_max x.(n − 1)) /. 2.;
      { x = x;
        x_min = d.x_min;
        x_max = d.x_max;
        n = Array.create n 0;
        w = Array.create n 0.0;
        w2 = Array.create n 0.0;
        bias = d.bias }

    let to_channel oc d =
      Array.iter (fun x →
        fprintf oc "␣%s␣0␣1␣0␣0␣1␣1\n" (Float.Double.to_string x)) d.x

    let as_block_data_to_channel oc name n_ch n_beam d =
      for i = 0 to Array.length d.x − 1 do
        fprintf oc "␣␣␣␣␣␣data␣%s(%d,%d,%d)␣/%s/\n"
          name i n_ch n_beam (Float.Double.to_string d.x.(i))
      done

  end
```

A.13.2 Polydivisions

```
module type Poly =
  sig

    module M : Diffmaps.Real
    include T
    val create : ?bias : (float → float) →
      (int × M.t) list → int → float → float → t

  end

module Make_Poly (M : Diffmaps.Real) (∗ : Poly ∗) =
  struct

    module M = M
```

```
type t =
    { x : float array;
      d : Mono.t array;
      n_bins : int;
      ofs : int array;
      maps : M.t array;
      n : int array;
      w : float array;
      w2 : float array }

let copy pd =
  { x = Array.copy pd.x;
    d = Array.map Mono.copy pd.d;
    n_bins = pd.n_bins;
    ofs = Array.copy pd.ofs;
    maps = Array.copy pd.maps;
    n = Array.copy pd.n;
    w = Array.copy pd.w;
    w2 = Array.copy pd.w2 }

let n_bins pd = pd.n_bins

let find pd y =
  let i = find_raw pd.x y in
  let x = M.ihp pd.maps.(i) y in
  pd.ofs.(i) + Mono.find pd.d.(i) x

let bins pd =
  let a = Array.create (pd.n_bins + 1) 0.0 in
  let bins0 = Mono.bins pd.d.(0) in
  let len = Array.length bins0 in
  Array.blit bins0 0 a 0 len;
  let ofs = ref len in
  for i = 1 to Array.length pd.d - 1 do
    let len = Mono.n_bins pd.d.(i) in
    Array.blit (Mono.bins pd.d.(i)) 1 a !ofs len;
    ofs := !ofs + len
  done;
  a

type interval =
    { nbin : int;
      x_min : float;
      x_max : float;
```

```
    map : M.t }

let interval nbin map =
  { nbin = nbin;
    x_min = M.x_min map;
    x_max = M.x_max map;
    map = map }

let id_map n y_min y_max =
  interval n (M.id ~x_min : y_min ~x_max : y_max y_min y_max)

let sort_intervals intervals =
  List.sort (fun i1 i2 -> compare i1.x_min i2.x_min) intervals
```

Fill the gaps between adjacent intervals, using `val default : int -> float -> float -> interval` to construct intermediate intervals.

```
let fill_gaps default n x_min x_max intervals =
  let rec fill_gaps' prev_x_max acc = function
    | i :: rest ->
        if i.x_min = prev_x_max then
          fill_gaps' i.x_max (i :: acc) rest
        else if i.x_min > prev_x_max then
          fill_gaps' i.x_max
            (i :: (default n prev_x_max i.x_min) :: acc) rest
        else
          invalid_arg "Polydivision.fill_gaps: overlapping"
    | [] ->
        if x_max = prev_x_max then
          List.rev acc
        else if x_max > prev_x_max then
          List.rev (default n prev_x_max x_max :: acc)
        else
          invalid_arg "Polydivision.fill_gaps: sticking out" in
  match intervals with
  | i :: rest ->
      if i.x_min = x_min then
        fill_gaps' i.x_max [i] rest
      else if i.x_min > x_min then
        fill_gaps' i.x_max (i :: [default n x_min i.x_min]) rest
      else
        invalid_arg "Polydivision.fill_gaps: sticking out"
  | [] -> [default n x_min x_max]
```

```
let create ?bias intervals n x_min x_max =
  let intervals = List.map (fun (n, m) -> interval n m) intervals in
  match fill_gaps id_map n x_min x_max (sort_intervals intervals) with
  | [] -> failwith "Division.Poly.create: impossible"
  | interval :: _ as intervals ->
      let ndiv = List.length intervals in
      let x = Array.of_list (interval.x_min ::
                               List.map (fun i -> i.x_max) intervals) in
      let d = Array.of_list
          (List.map (fun i ->
             Mono.create ?bias i.nbin i.x_min i.x_max) intervals) in
      let ofs = Array.create ndiv 0 in
      for i = 1 to ndiv - 1 do
        ofs.(i) <- ofs.(i - 1) + Mono.n_bins d.(i - 1)
      done;
      let n_bins = ofs.(ndiv - 1) + Mono.n_bins d.(ndiv - 1) in
      { x = x;
        d = d;
        n_bins = n_bins;
        ofs = ofs;
        maps = Array.of_list (List.map (fun i -> i.map) intervals);
        n = Array.create ndiv 0;
        w = Array.create ndiv 0.0;
        w2 = Array.create ndiv 0.0 }
```

We can safely assume that *find_raw pd.x y = find_raw pd.x x*.

$$w = \frac{f}{\frac{\mathrm{d}x}{\mathrm{d}y}} = f \cdot \frac{\mathrm{d}y}{\mathrm{d}x} \tag{78}$$

Here, the jacobian make no difference for the final result, but it steers VEGAS/VAMP into the right direction.

```
let caj pd y =
  let i = find_raw pd.x y in
  let m = pd.maps.(i)
  and d = pd.d.(i) in
  let x = M.ihp m y in
  M.caj m y *. Mono.caj d x
```

```
let record pd y f =
  let i = find_raw pd.x y in
  let m = pd.maps.(i) in
  let x = M.ihp m y in
  let w = M.jac m x *. f in
  Mono.record pd.d.(i) x w;
  pd.n.(i) ← succ pd.n.(i);
  pd.w.(i) ← pd.w.(i) +. w;
  pd.w2.(i) ← pd.w2.(i) +. w *. w
```

Rebin the divisions, enforcing fixed boundaries for the inner intervals.

```
let rebin ?(power = 1.5) ?(fixed_min = false) ?(fixed_max = false) pd =
  let ndiv = Array.length pd.d in
  let rebin_mono i d =
    if ndiv ≤ 1 then
      Mono.rebin ~power ~fixed_min ~fixed_max d
    else if i = 0 then
      Mono.rebin ~power ~fixed_min ~fixed_max :true d
    else if i = ndiv − 1 then
      Mono.rebin ~power ~fixed_min :true ~fixed_max d
    else
      Mono.rebin ~power ~fixed_min :true ~fixed_max :true d in
  { x = Array.copy pd.x;
    d = Array.init ndiv (fun i → rebin_mono i pd.d.(i));
    n_bins = pd.n_bins;
    ofs = pd.ofs;
    maps = Array.copy pd.maps;
    n = Array.create ndiv 0;
    w = Array.create ndiv 0.0;
    w2 = Array.create ndiv 0.0 }

let to_channel oc pd =
  for i = 0 to Array.length pd.d − 1 do
    let map = M.encode pd.maps.(i)
    and bins = Mono.bins pd.d.(i)
    and j0 = if i = 0 then 0 else 1 in
    for j = j0 to Array.length bins − 1 do
      fprintf oc "␣%s␣%s\n" (Float.Double.to_string bins.(j)) map;
    done
  done
```

```
    let as_block_data_to_channel oc tag n_ch n_beam pd =
      let k = ref 0 in
      for i = 0 to Array.length pd.d - 1 do
        let bins = Mono.bins pd.d.(i)
        and j0 = if i = 0 then 0 else 1 in
        for j = j0 to Array.length bins - 1 do
          fprintf oc "␣␣␣␣␣␣data␣bdx%s(%d,%d,%d)␣/%s/\n"
            tag !k n_ch n_beam (Float.Double.to_string bins.(j));
          M.as_block_data_to_channel pd.maps.(i) oc tag !k n_ch n_beam;
          incr k
        done
      done

  end
module Poly = Make_Poly (Diffmaps.Default)
```

A.14 Interface of *Grid*

```
module type T =
  sig
    module D : Division.T

    type t
    val copy : t → t
```

Create an initial grid.

```
    val create : ?triangle :bool → D.t → D.t → t
```

record grid x1 x2 w records the value *w* in the bin corresponding to coordinates *x1* and *x2*.

```
    val record : t → float → float → float → unit
```

VEGAS style rebinning.

```
    val rebin : ?power :float →
      ?fixed_x1_min :bool → ?fixed_x1_max :bool →
        ?fixed_x2_min :bool → ?fixed_x2_max :bool → t → t
```

The sum of all the weights shall be one.

```
    val normalize : t → t
```

Adapt an initial grid to data. The *power* controls speed vs. stability of adaption and is passed on to *Division.rebin*. *iterations* provides a hard cutoff for the number of iterations (default: 1000), while *margin* and *cutoff*

control the soft cutoff of the adaption. If the variance grows to the best value multiplied by *margin* of if there are no improvements for *cutoff* steps, the adaption is stopped (defaults: 1.5 and 20). The remaining options control if the boundaries are fixed or allowed to move towards the limits of the dataset. The defaults are all false, meaning that the boundaries are allowed to move.

```
val of_bigarray : ?verbose :bool → ?power :float →
  ?iterations :int → ?margin :float → ?cutoff :int →
    ?fixed_x1_min :bool → ?fixed_x1_max :bool →
      ?fixed_x2_min :bool → ?fixed_x2_max :bool →
        (float, Bigarray.float64_elt,
          Bigarray.fortran_layout) Bigarray.Array2.t → t → t
```

Write output that `Circe2` can read:

```
type channel =
    { pid1 : int;
      pol1 : int;
      pid2 : int;
      pol2 : int;
      lumi : float;
      g : t }

val to_channel : out_channel → channel → unit

type design =
    { name : string;
      roots : float;
      channels : channel list;
      comments : string list }

val design_to_channel : out_channel → design → unit
val designs_to_channel : out_channel →
  ?comments :string list→ design list → unit
val designs_to_file : string →
  ?comments :string list → design list → unit

val design_as_block_data_to_channel :
    out_channel → int × design → unit
val designs_as_block_data_to_channel :
    out_channel → ?comments :string list→ (int × design) list →
unit
val designs_as_block_data_to_file :
    string → ?comments :string list → (int × design) list → unit

val variance : t → float
```

```
  end
module Make (D : Division.T) : T with module D = D
```

A.15 Implementation of *Grid*

```
open Printf
module type T =
  sig
    module D : Division.T
    type t
    val copy : t → t
    val create : ?triangle :bool → D.t → D.t → t
    val record : t → float → float → float → unit
    val rebin : ?power :float →
      ?fixed_x1_min :bool → ?fixed_x1_max :bool →
        ?fixed_x2_min :bool → ?fixed_x2_max :bool → t → t
    val normalize : t → t
    val of_bigarray : ?verbose :bool → ?power :float →
      ?iterations :int → ?margin :float → ?cutoff :int →
        ?fixed_x1_min :bool → ?fixed_x1_max :bool →
          ?fixed_x2_min :bool → ?fixed_x2_max :bool →
            (float, Bigarray.float64_elt,
              Bigarray.fortran_layout) Bigarray.Array2.t → t → t
    type channel =
        { pid1 : int;
          pol1 : int;
          pid2 : int;
          pol2 : int;
          lumi : float;
          g : t }
    val to_channel : out_channel → channel → unit
    type design =
        { name : string;
          roots : float;
          channels : channel list;
          comments : string list }
```

```
    val design_to_channel : out_channel → design → unit
    val designs_to_channel : out_channel →
      ?comments :string list→ design list → unit
    val designs_to_file : string →
      ?comments :string list → design list → unit
    val design_as_block_data_to_channel :
        out_channel → int × design → unit
    val designs_as_block_data_to_channel :
        out_channel → ?comments :string list→ (int × design) list →
 unit
    val designs_as_block_data_to_file :
        string → ?comments :string list → (int × design) list → unit
    val variance : t → float
  end
module Make (D : Division.T) =
  struct
    module D = D
    type t =
        { d1 : D.t;
          d2 : D.t;
          w : float array;
          var : float array;
          triangle : bool }
    let copy grid =
      { d1 = D.copy grid.d1;
        d2 = D.copy grid.d2;
        w = Array.copy grid.w;
        var = Array.copy grid.var;
        triangle = grid.triangle }
    let create ?(triangle = false) d1 d2 =
      let n = D.n_bins d1 × D.n_bins d2 in
      { d1 = d1;
        d2 = d2;
        w = Array.create n 0.0;
        var = Array.create n 0.0;
        triangle = triangle }
```

Here's the offset-0 variant of the offset-1 Fortran code:

```
    let find grid x y =
      D.find grid.d1 x + D.n_bins grid.d1 × D.find grid.d2 y
```

```
let project_triangle triangle x y =
  if triangle then begin
    if x ≥ y then begin
      (x, y /. x)
    end else begin
      (y, x /. y)
    end
  end else
    (x, y)
```

Note that there is *no* jacobian here. It is applied by Fortran program interpreting the grid as distribution. It is not needed for the event generator anyway.

```
let record grid x y f =
  let x', y' = project_triangle grid.triangle x y in
  D.record grid.d1 x' f;
  D.record grid.d2 y' f;
  let n = find grid x' y' in
  grid.w.(n) ← grid.w.(n) +. f;
  grid.var.(n) ← grid.var.(n) +. f /. D.caj grid.d1 x' /. D.caj grid.d2 y'

let rebin ?power ?fixed_x1_min ?fixed_x1_max
    ?fixed_x2_min ?fixed_x2_max grid =
  let n = D.n_bins grid.d1 × D.n_bins grid.d2 in
  { d1 = D.rebin ?power
      ?fixed_min : fixed_x1_min ?fixed_max : fixed_x1_max grid.d1;
    d2 = D.rebin ?power
      ?fixed_min : fixed_x2_min ?fixed_max : fixed_x2_max grid.d2;
    w = Array.create n 0.0;
    var = Array.create n 0.0;
    triangle = grid.triangle }

let normalize grid =
  let sum_w = Array.fold_left (+.) 0.0 grid.w in
  { d1 = D.copy grid.d1;
    d2 = D.copy grid.d2;
    w = Array.map (fun w → w /. sum_w) grid.w;
    var = Array.copy grid.var;
    triangle = grid.triangle }
```

Monitoring the variance in each cell is *not* a good idea for approximating distributions of unweighted events: it always vanishes for unweighted events,

even if they are distributed very unevenly. Therefore, we monitor the *global* variance instead:

```
let variance grid =
  let n = float (Array.length grid.w) in
  let w = Array.fold_left (+.) 0.0 grid.w /. n
  and w2 = Array.fold_left (fun acc w → acc +. w *. w) 0.0 grid.w /. n in
  w2 -. w *. w

let variance grid =
  let n = float (Array.length grid.var) in
  let w = Array.fold_left (+.) 0.0 grid.var /. n
  and w2 = Array.fold_left (fun acc w → acc +. w *. w) 0.0 grid.var /. n in
  w2 -. w *. w
```

Find the grid with the lowest variance. Allow local fluctuations and stop only after moving to twice the lowest value.

```
let start_progress_report verbose var =
  if verbose then begin
    eprintf "adapting␣variance:␣%g" var;
    flush stderr
  end

let progress_report verbose soft_limit best_var var =
  if verbose then begin
    if var < best_var then begin
      eprintf ",␣%g" var;
      flush stderr
    end else begin
      eprintf "␣[%d]" soft_limit;
      flush stderr
    end
  end

let stop_progress_report verbose =
  if verbose then begin
    eprintf "␣done.\n";
    flush stderr
  end
```

The main routine constructing an adapted grid.

```
let of_bigarray ?(verbose = false)
    ?power ?(iterations = 1000) ?(margin = 1.5) ?(cutoff = 10)
    ?fixed_x1_min ?fixed_x1_max ?fixed_x2_min ?fixed_x2_max data initial =
```

```
let record_data grid =
  for i2 = 1 to Bigarray.Array2.dim2 data do
    let x = Bigarray.Array2.get data 1 i2
    and y = Bigarray.Array2.get data 2 i2
    and w = Bigarray.Array2.get data 3 i2 in
    try
      record grid x y w
    with
    | Division.Out_of_range (x, (x_min, x_max)) →
        eprintf "internal error: %g not in [%g,%g]\n" x x_min x_max
  done in

let rebinner grid =
  rebin ?power
    ?fixed_x1_min ?fixed_x1_max ?fixed_x2_min ?fixed_x2_max grid in

let rec improve_bigarray hard_limit soft_limit best_var best_grid grid =
  if soft_limit ≤ 0 ∨ hard_limit ≤ 0 then
    normalize best_grid
  else begin
    record_data grid;
    let var = variance grid in
    progress_report verbose soft_limit best_var var;
    if var ≥ margin *. best_var then
      normalize best_grid
    else
      let best_var, best_grid, soft_limit =
        if var < best_var then
          (var, grid, cutoff)
        else
          (best_var, best_grid, pred soft_limit) in
```

Continuation passing makes recursion with exception handling tail recursive. This is not really needed, because the data structures are not to big and recursion is not expected to be too deep. It doesn't hurt either, since the idiom is sufficiently transparent.

```
      let continue =
        try
          let grid' = rebinner grid in
          fun () → improve_bigarray
                (pred hard_limit) soft_limit best_var best_grid grid'
        with
```

```
          | Division.Rebinning_failure msg →
              eprintf "circe2: rebinning failed: %s!\n" msg;
              fun () → best_grid in
        continue ()
    end in
  record_data initial;
  let var = variance initial in
  start_progress_report verbose var;
  let result =
    improve_bigarray iterations cutoff var initial (rebinner initial) in
  stop_progress_report verbose;
  result
type channel =
    { pid1 : int;
      pol1 : int;
      pid2 : int;
      pol2 : int;
      lumi : float;
      g : t }
let to_channel oc ch =
  fprintf oc "pid1, pol1, pid2, pol2, lumi\n";
  fprintf oc " %d %d %d %d %G\n"
    ch.pid1 ch.pol1 ch.pid2 ch.pol2 ch.lumi;
  fprintf oc "#bins1, #bins2, triangle?\n";
  fprintf oc " %d %d %s\n"
    (D.n_bins ch.g.d1) (D.n_bins ch.g.d2)
    (if ch.g.triangle then "T" else "F");
  fprintf oc "x1, map1, alpha1, xi1, eta1, a1, b1\n";
  D.to_channel oc ch.g.d1;
  fprintf oc "x2, map2, alpha2, xi2, eta2, a2, b2\n";
  D.to_channel oc ch.g.d2;
  fprintf oc "weights\n";
  Array.iter (fun x →
    fprintf oc " %s\n" (Float.Double.to_string x)) ch.g.w
let as_block_data_to_channel oc n_beam n_ch ch =
  let print_integer name value =
    fprintf oc "      data bd%s(%d,%d) /%d/\n" name n_ch n_beam value
  and print_float name value =
    fprintf oc "      data bd%s(%d,%d) /%s/\n"
```

```
      name n_ch n_beam (Float.Double.to_string value)
  and print_string name value =
    fprintf oc "␣␣␣␣␣␣data␣bd%s(%d,%d)␣/'%s'/\n" name n_ch n_beam value
  and print_logical name value =
    fprintf oc "␣␣␣␣␣␣data␣bd%s(%d,%d)␣/%s/\n"
      name n_ch n_beam (if value then ".true." else ".false.") in
  print_integer "pid1" ch.pid1;
  print_integer "pol1" ch.pol1;
  print_integer "pid2" ch.pid2;
  print_integer "pol2" ch.pol2;
  print_float "lumi" ch.lumi;
  print_integer "nb1" (D.n_bins ch.g.d1);
  print_integer "nb2" (D.n_bins ch.g.d2);
  print_logical "tria" ch.g.triangle;
  D.as_block_data_to_channel oc "1" n_ch n_beam ch.g.d1;
  D.as_block_data_to_channel oc "2" n_ch n_beam ch.g.d2;
  ()

type design =
    { name : string;
      roots : float;
      channels : channel list;
      comments : string list }

type polarization_support =
  | Averaged
  | Helicities
  | Density_Matrices

let polarization_support design =
  if List.for_all (fun ch → ch.pol1 = 0 ∧ ch.pol2 = 0)
      design.channels then
    Averaged
  else if List.for_all (fun ch → ch.pol1 ≠ 0 ∧ ch.pol2 ≠ 0)
      design.channels then
    Helicities
  else
    invalid_arg
      "Grid.polarization_support:␣mixed␣polarization␣support!"

let format_polarization_support = function
  | Averaged → "averaged"
  | Helicities → "helicities"
```

```
  | Density_Matrices → "density␣matrices"

let getlogin () =
  (Unix.getpwuid (Unix.getuid ())).Unix.pw_name

let design_to_channel oc design =
  let utc = Unix.gmtime (Unix.time ()) in
  List.iter (fun s → fprintf oc "!␣%s\n" s) design.comments;
  fprintf oc "!␣generated␣with␣%s␣by␣%s@%s,␣"
    (Sys.argv.(0)) (getlogin ()) (Unix.gethostname ());
  fprintf oc "%4.4d/%2.2d/%2.2d␣%2.2d:%2.2d:%2.2d␣GMT\n"
    (utc.Unix.tm_year + 1900) (utc.Unix.tm_mon + 1) utc.Unix.tm_mday
    utc.Unix.tm_hour utc.Unix.tm_min utc.Unix.tm_sec;
  fprintf oc "CIRCE2␣FORMAT#1\n";
  fprintf oc "design,␣roots\n";
  fprintf oc "␣'%s'␣%G\n" design.name design.roots;
  fprintf oc "#channels,␣pol.support\n";
  fprintf oc "␣%d␣'%s'\n"
    (List.length design.channels)
    (format_polarization_support (polarization_support design));
  List.iter (to_channel oc) design.channels;
  fprintf oc "ECRIC2\n"

let design_as_block_data_to_channel oc (n_design, design) =
  let utc = Unix.gmtime (Unix.time ()) in
  List.iter (fun s → fprintf oc "c␣%s\n" s) design.comments;
  fprintf oc "c␣generated␣with␣%s␣by␣%s@%s,␣"
    (Sys.argv.(0)) (Unix.getlogin ()) (Unix.gethostname ());
  fprintf oc "%4.4d/%2.2d/%2.2d␣%2.2d:%2.2d:%2.2d␣GMT\n"
    (utc.Unix.tm_year + 1900) (utc.Unix.tm_mon + 1) utc.Unix.tm_mday
    utc.Unix.tm_hour utc.Unix.tm_min utc.Unix.tm_sec;
  fprintf oc "␣␣␣␣␣␣data␣bddsgn(%d)␣␣/'%s'/\n" n_design design.name;
  fprintf oc "␣␣␣␣␣␣data␣bdbrs(%d)␣␣/%G/\n" n_design design.roots;
  fprintf oc "␣␣␣␣␣␣data␣bdnc(%d)␣␣␣/%d/\n"
    n_design (List.length design.channels);
  let _ =
    List.fold_left (fun n_ch ch →
      as_block_data_to_channel oc n_design n_ch ch;
      succ n_ch) 1 design.channels in
  ()

let designs_to_channel oc ?(comments = []) designs =
  List.iter (fun c → fprintf oc "!␣%s\n" c) comments;
```

```
    List.iter (design_to_channel oc) designs

  let designs_as_block_data_to_channel oc ?(comments = []) designs =
    List.iter (fun c -> fprintf oc "! %s\n" c) comments;
    List.iter (design_as_block_data_to_channel oc) designs

  let designs_to_file name ?comments designs =
    let oc = open_out name in
    designs_to_channel oc ?comments designs;
    close_out oc

  let designs_as_block_data_to_file name ?comments designs =
    let oc = open_out name in
    designs_as_block_data_to_channel oc ?comments designs;
    close_out oc

end
```

A.16 Interface of *Events*

We're dealing with Fortran style `DOUBLE PRECISION` arrays exclusively.

```
type t =
    (float, Bigarray.float64_elt, Bigarray.fortran_layout) Bigarray.Array2.t
```

Read an ASCII representation of a big array from a channel or a file. The array is read in pieces of *chunk* columns each; the default value for *chunk* is 100000. The number of rows is given by the integer argument, while the number of columns is determined by the number of lines in the file. If the *file* argument is present the resulting bigarray is mapped to a file.

```
val of_ascii_channel : ?file :string -> ?chunk :int -> int -> in_channel ->
  t
val of_ascii_file : ?file :string -> ?chunk :int -> int -> string -> t
```

Map a file containing a binary representation of a big array. The number of rows is again given by the argument and the number of columns is determined by the size of the file. The first version does a read-only (or rather copy-on-write) map, while the second version allows modifications.

```
val of_binary_file : int -> string -> t
val shared_map_binary_file : int -> string -> t
```

Selfexplaining, hopefully ...

```
val to_ascii_channel : out_channel -> t -> unit
val to_ascii_file : string -> t -> unit
val to_binary_file : string -> t -> unit
```

Utilities for reading ASCII representations.

```
val lexer : char Stream.t → Genlex.token Stream.t
val next_float : Genlex.token Stream.t → float
```

A.17 Implementation of *Events*

A.17.1 Reading Bigarrays

Reading big arrays efficiently is not trivial, if we don't know the size of the arrays beforehand. Here we use the brute force approach of reading a list of not-so-big arrays and blitting them into the resulting array later. This avoids a second reading of the file, but temporarily needs twice the memory.

```
open Bigarray
open Printf

type t = (float, float64_elt, fortran_layout) Array2.t

exception Incomplete of int × t
```

Read lines from a channel into the columns of a bigarray. If the file turns out to be short, the exception *Incomplete* (*i2*, *array*) is raised with the number of columns actually read.

```
let read_lines ic reader array i2_first i2_last =
  let i2 = ref i2_first in
  try
    while !i2 ≤ i2_last do
      reader array !i2 (input_line ic);
      incr i2
    done
  with
  | End_of_file → raise (Incomplete (pred !i2, array))
```

Decode a line of floating point numbers into a column of a bigarray.
Fortran allows `'d'` and `'D'` as exponent starter, but O'Caml's *Genlex* doesn't accept it.

```
let normalize_ascii_floats orig =
  let normalized = String.copy orig in
  for i = 0 to String.length normalized − 1 do
    let c = normalized.[i] in
    if c = 'd' ∨ c = 'D' then
      normalized.[i] ← 'E'
  done;
  normalized
```

```
let lexer = Genlex.make_lexer []

let next_float s =
  match Stream.next s with
  | Genlex.Int n → float n
  | Genlex.Float x → x
  | _ → invalid_arg "Events.int_as_float"

let read_floats array i2 line =
  let tokens = lexer (Stream.of_string (normalize_ascii_floats line)) in
  for i1 = 1 to Array2.dim1 array do
    Array2.set array i1 i2 (next_float tokens)
  done
```

Try to read the columns of a bigarray from a channel. If the file turns out to be short, the exception *Incomplete* (*dim2*, *array*) is raised with the number of columns actually read.

```
let try_of_ascii_channel dim1 dim2 ic =
  let array = Array2.create float64 fortran_layout dim1 dim2 in
  read_lines ic read_floats array 1 dim2;
  (dim2, array)
```

Read a *dim1* floating point numbers per line into the columns of a reverted list of bigarrays, each with a maximum of *chunk* columns.

```
let rev_list_of_ascii_channel chunk dim1 ic =
  let rec rev_list_of_ascii_channel' acc =
    let continue =
      try
        let acc' = try_of_ascii_channel dim1 chunk ic :: acc in
        fun () → rev_list_of_ascii_channel' acc'
      with
      | Incomplete (len, a) → fun () → (len, a) :: acc in
    continue () in
  rev_list_of_ascii_channel' []
```

Concatenate a list of bigarrays $[(l_n, a_n); \ldots; (l_2, a_2); (l_1, a_1)]$ in reverse order $a_1 a_2 \ldots a_n$. Of each array a_i, only the first l_i columns are used. If the optional *file* name is present, map the corresponding file to the bigarray. We can close the file descriptor immediately, since `close(2)` does *not* `munmap(2)`.

```
let create_array ?file dim1 dim2 =
  match file with
  | None → Array2.create float64 fortran_layout dim1 dim2
  | Some name →
      let fd =
        Unix.openfile name
          [Unix.O_RDWR; Unix.O_CREAT; Unix.O_TRUNC] 644₈ in
      let a = Array2.map_file fd float64 fortran_layout true dim1 dim2 in
      Unix.close fd;
      a

let rev_concat ?file arrays =
  let sum_dim2 =
    List.fold_left (fun sum (dim2, _) → sum + dim2) 0 arrays in
  if sum_dim2 ≤ 0 then
    invalid_arg "Events.rev_concat";
  let dim1 = Array2.dim1 (snd (List.hd arrays)) in
  let array = create_array ?file dim1 sum_dim2 in
  let _ = List.fold_right
      (fun (dim2, a) ofs →
        Array2.blit
          (Array2.sub_right a 1 dim2) (Array2.sub_right array ofs dim2);
        ofs + dim2)
      arrays 1 in
  array

let of_ascii_channel ?file ?(chunk = 100000) dim1 ic =
  rev_concat ?file (rev_list_of_ascii_channel chunk dim1 ic)

let of_ascii_file ?file ?chunk dim1 name =
  let ic = open_in name in
  let a = of_ascii_channel ?file ?chunk dim1 ic in
  close_in ic;
  a
```

We can close the file descriptor immediately, since `close(2)` does *not* `munmap(2)`.

```
let of_binary_file dim1 file =
  let fd = Unix.openfile file [Unix.O_RDONLY] 644₈ in
  let a = Array2.map_file fd float64 fortran_layout false dim1 (−1) in
  Unix.close fd;
  a
```

```
let shared_map_binary_file dim1 file =
  let fd = Unix.openfile file [Unix.O_RDWR] 644_8 in
  let a = Array2.map_file fd float64 fortran_layout true dim1 (-1) in
  Unix.close fd;
  a

let to_ascii_channel oc a =
  let dim1 = Array2.dim1 a
  and dim2 = Array2.dim2 a in
  for i2 = 1 to dim2 do
    for i1 = 1 to dim1 do
      fprintf oc "␣%.17E" (Array2.get a i1 i2)
    done;
    fprintf oc "\n"
  done

let to_ascii_file name a =
  let oc = open_out name in
  to_ascii_channel oc a;
  close_out oc

let to_binary_file file a =
  let fd =
    Unix.openfile file
      [Unix.O_RDWR; Unix.O_CREAT; Unix.O_TRUNC] 644_8 in
  let a' =
    Array2.map_file fd float64 fortran_layout true
      (Array2.dim1 a) (Array2.dim2 a) in
  Unix.close fd;
  Array2.blit a a'
```

A.18 Interface of *Commands*

An example for a command file:

```
{ file = "tesla.circe"
   { design = "TESLA" roots = 500
     { pid/1 = electron pid/2 = positron
       events = "tesla_500.electron_positron" }
     { pid = photon
       events = "tesla_500.gamma_gamma" }
     { pid/1 = photon pid/2 = positron
       events = "tesla_500.gamma_positron" }
```

```
    { pid/1 = electron pid/2 = photon
      events = "tesla_500.electron_gamma" } }
  { design = "TESLA" roots = 800
    { pid/1 = electron pid/2 = positron
      events = "tesla_800.electron_positron" } }
  { design = "TESLA" roots = 500
    { pid = photon
      events = "tesla_gg_500.gamma_gamma" } }
  { design = "TESLA" roots = 500
    { pid = electron
      events = "tesla_ee_500.electron_electron" } } }
```

type t
val $parse\_file$: $string \rightarrow t$
val $parse\_string$: $string \rightarrow t$
val $execute$: $t \rightarrow unit$

exception $Parse\_Error$ of $string$

A.19 Implementation of *Commands*

open $Printf$

module $Maps = Diffmaps.Default$
module $Div = Division.Make\_Poly\ (Maps)$
module $Grid = Grid.Make\ (Div)$

A.19.1 Abstract Syntax and Default Values

A channel is uniquely specified by PDG particle ids and polarizations $\{-1, 0, +1\}$, which must match the 'events' in the given file; as should the luminosity. The options are for tuning the grid.

type $channel =$
  $\{$ $pid1$: int;
    $pol1$: int;
    $pid2$: int;
    $pol2$: int;
    $lumi$: $float$;
    $bins1$: int;
    $x1\_min$: $float$;

```
    x1_max : float;
    fixed_x1_min : bool;
    fixed_x1_max : bool;
    intervals1 : (int × Maps.t) list;
    bins2 : int;
    x2_min : float;
    x2_max : float;
    fixed_x2_min : bool;
    fixed_x2_max : bool;
    intervals2 : (int × Maps.t) list;
    triangle : bool;
    iterations : int;
    events : string;
    binary : bool;
    columns : int }
let default_channel =
  { pid1 = 11 (* e⁻ *);
    pol1 = 0;
    pid2 = −11 (* e⁺ *);
    pol2 = 0;
    lumi = 0.0;
    bins1 = 20;
    x1_min = 0.0;
    x1_max = 1.0;
    fixed_x1_min = false;
    fixed_x1_max = false;
    intervals1 = [];
    bins2 = 20;
    x2_min = 0.0;
    x2_max = 1.0;
    fixed_x2_min = false;
    fixed_x2_max = false;
    intervals2 = [];
    triangle = false;
    iterations = 1000;
    events = "circe2.events";
    binary = false;
    columns = 3 }
```

A parameter set is uniquely specified by PDG particle ids (*par abus de langage*), polarizations (now a floating point number for the effective polariza-

tion of the beam), and center of mass energy. This must match the 'events' in the files given for the channels. The other options are for tuning the grid.

```
type design =
    { design : string;
      roots : float;
      design_bins1 : int;
      design_bins2 : int;
      channels : channel list;
      comments : string list }

let default_design =
    { design = "TESLA";
      roots = 500.0;
      design_bins1 = default_channel.bins1;
      design_bins2 = default_channel.bins2;
      channels = [];
      comments = [] }
```

One file can hold more than one grid.

```
type file =
    { name : string; block_data : string option; designs : design list }

let default_file =
  { name = "circe2_tool.out"; block_data = None; designs = [] }

type t = file list
```

A.19.2 Processing

```
let report msg =
  prerr_string msg;
  flush stderr

let process_channel ch =
  report ("reading: " ^ ch.events ^ " ...");
  let data =
    if ch.binary then
      Events.of_binary_file ch.columns ch.events
    else
      Events.of_ascii_file 3 ch.events in
  report " done.\n";
  let initial_grid =
```

```
    Grid.create ~triangle : ch.triangle
      (Div.create ch.intervals1 ch.bins1 ch.x1_min ch.x1_max)
      (Div.create ch.intervals2 ch.bins2 ch.x2_min ch.x2_max) in
  let grid =
    Grid.of_bigarray ~verbose :true ~iterations : ch.iterations
      ~fixed_x1_min : ch.fixed_x1_min ~fixed_x1_max : ch.fixed_x1_max
      ~fixed_x2_min : ch.fixed_x2_min ~fixed_x2_max : ch.fixed_x2_max
      data initial_grid in
  { Grid.pid1 = ch.pid1;
    Grid.pol1 = ch.pol1;
    Grid.pid2 = ch.pid2;
    Grid.pol2 = ch.pol2;
    Grid.lumi = ch.lumi;
    Grid.g = grid }

module S = Set.Make (struct type t = string let compare = compare end)

let channel_prerequisites acc ch =
  S.add ch.events acc

let process_design oc name block_data design =
  let channels = List.rev_map process_channel design.channels
  and comments = List.rev design.comments in
  let acc =
    { Grid.name = design.design;
      Grid.roots = design.roots;
      Grid.channels = channels;
      Grid.comments = comments } in
  report ("writing:␣" ^ name ^ "␣...");
  Grid.design_to_channel oc acc;
  report "␣done.\n";
  match block_data with
  | Some (oc, name, nb) →
      report ("writing:␣" ^ name ^ "␣...");
      Grid.design_as_block_data_to_channel oc (nb, acc);
      report "␣done.\n";
  | None → ()

let design_prerequisites acc design =
  List.fold_left (channel_prerequisites) acc design.channels

let write_file file =
  let oc = open_out file.name in
  begin match file.block_data with
```

```
  | Some name →
      let bd_oc = open_out name in
      let _ =
        List.fold_left (fun nb design →
          process_design oc file.name (Some (bd_oc, name, nb)) design;
          succ nb) 1 file.designs in
      close_out bd_oc
  | None → List.iter (process_design oc file.name None) file.designs
  end;
  close_out oc

let file_prerequisites acc file =
  List.fold_left (design_prerequisites) acc file.designs

let prerequisites files =
  List.fold_left (file_prerequisites) S.empty files

let unreadable name =
  try
    Unix.access name [Unix.R_OK];
    false
  with
  | Unix.Unix_error (_, _, _) → true

let execute files =
  let missing = S.filter unreadable (prerequisites files) in
  if S.is_empty missing then
    List.iter write_file files
  else
    eprintf "circe2_tool: unreadable input files: %s!\n"
      (String.concat ", " (S.elements missing))
```

A.19.3 Concrete syntax.

```
open Genlex

let lexer =
  Genlex.make_lexer
    ["file"; "block_data"; "events"; "binary"; "ascii"; "columns";
     "design"; "roots"; "pid"; "pol"; "unpol"; "lumi";
     "fix"; "free"; "min"; "max";
     "bins"; "triangle"; "notriangle"; "iterations";
```

```
    "electron"; "positron"; "photon"; "gamma";
    "map"; "id"; "power"; "resonance";
    "beta"; "eta"; "width"; "center";
    "comment"; "/"; "["; ","; "]"; "{"; "}"; "="; "*" ]

exception Parse_Error of string

let expecting s =
  raise (Parse_Error ("expecting " ^ s))

let int_as_float = parser
    [< 'Float x >] -> x
  | [< 'Int n >] -> float n

let interval_cmd = parser
  | [< 'Int n;
       'Kwd "["; x_min = int_as_float;
       'Kwd ","; x_max = int_as_float;
       'Kwd "]" >] -> (n, x_min, x_max)
  | [< >] -> expecting "interval: `n_bins [x_min, x_max]'"

let power_cmd (n, x_min, x_max) = parser
  | [< 'Kwd "beta"; 'Kwd "="; beta = int_as_float;
       'Kwd "eta"; 'Kwd "="; eta = int_as_float >] ->
         if beta <= -1.0 then begin
           eprintf "circe2: ignoring invalid beta: %g <= -1\n" beta;
           flush stderr;
           (n, Maps.id x_min x_max)
         end else
           let alpha = 1.0 /. (1.0 +. beta) in
           (n, Maps.power ~alpha ~eta x_min x_max)
  | [< >] -> expecting "power map parameters: `beta = float eta = float'"

let resonance_cmd (n, x_min, x_max) = parser
  | [< 'Kwd "center"; 'Kwd "="; eta = int_as_float;
       'Kwd "width"; 'Kwd "="; a = int_as_float >] ->
         (n, Maps.resonance ~eta ~a x_min x_max)
  | [< >] -> expecting "resonance map parameters: `center = float width = float'"

let map_cmd = parser
  | [< 'Kwd "id";
       'Kwd "{"; (n, x_min, x_max) = interval_cmd; 'Kwd "}" >] ->
         (n, Maps.id x_min x_max)
  | [< 'Kwd "power";
       'Kwd "{"; (n, x_min, x_max) = interval_cmd;
```

```
        m = power_cmd (n, x_min, x_max); 'Kwd "}" ⟩] → m
  | [⟨ 'Kwd "resonance";
        'Kwd "{"; (n, x_min, x_max) = interval_cmd;
        m = resonance_cmd (n, x_min, x_max); 'Kwd "}" ⟩] → m
  | [⟨ ⟩] → expecting "map:␣'id',␣'power'␣or␣'resonance'"

let particle_as_int = parser
  | [⟨ 'Int pid ⟩] → pid
  | [⟨ 'Kwd "electron" ⟩] → 11
  | [⟨ 'Kwd "positron" ⟩] → -11
  | [⟨ 'Kwd "photon" ⟩] → 22
  | [⟨ 'Kwd "gamma" ⟩] → 22

let polarization_as_int = parser
  | [⟨ 'Int pol ⟩] → pol
  | [⟨ 'Kwd "unpol" ⟩] → 0
  | [⟨ ⟩] → expecting "normalized␣helicity:␣integer␣or␣'unpol'"

let polarization_as_float = parser
  | [⟨ pol = int_as_float ⟩] → pol
  | [⟨ 'Kwd "unpol" ⟩] → 0.0
  | [⟨ ⟩] → expecting "polarization:␣float␣or␣'unpol'"

type coord = X1 | X2 | X12
let coord = parser
  | [⟨ 'Kwd "/"; 'Int c ⟩] →
      begin match c with
      | 1 → X1
      | 2 → X2
      | _ →
          eprintf "circe2:␣ignoring␣dimension␣%d␣(not␣1,␣2,␣or␣*)\n" c;
          X12
      end
  | [⟨ ⟩] → X12

type side = Min | Max | Minmax
let side = parser
  | [⟨ 'Kwd "min" ⟩] → Min
  | [⟨ 'Kwd "max" ⟩] → Max
  | [⟨ 'Kwd "*" ⟩] → Minmax
  | [⟨ ⟩] → expecting "'min',␣'max'␣or␣'*'"
```

```
type channel_cmd =
  | Pid of int × coord
  | Pol of int × coord
  | Lumi of float
  | Xmin of float × coord
  | Xmax of float × coord
  | Bins of int × coord
  | Diffmap of (int × Maps.t) × coord
  | Triangle of bool
  | Iterations of int
  | Events of string
  | Binary of bool
  | Columns of int
  | Fix of bool × coord × side

let channel_cmd = parser
  | [⟨ 'Kwd "pid"; c = coord;
       'Kwd "="; n = particle_as_int ⟩] → Pid (n, c)
  | [⟨ 'Kwd "pol"; c = coord;
       'Kwd "="; p = polarization_as_int ⟩] → Pol (p, c)
  | [⟨ 'Kwd "fix"; c = coord; 'Kwd "="; s = side ⟩] → Fix (true, c, s)
  | [⟨ 'Kwd "free"; c = coord; 'Kwd "="; s = side ⟩] → Fix (false, c, s)
  | [⟨ 'Kwd "bins"; c = coord; 'Kwd "="; 'Int n ⟩] → Bins (n, c)
  | [⟨ 'Kwd "min"; c = coord; 'Kwd "="; x = int_as_float ⟩] →
Xmin (x, c)
  | [⟨ 'Kwd "max"; c = coord; 'Kwd "="; x = int_as_float ⟩] →
Xmax (x, c)
  | [⟨ 'Kwd "map"; c = coord; 'Kwd "="; m = map_cmd ⟩] →
Diffmap (m, c)
  | [⟨ 'Kwd "lumi"; 'Kwd "="; l = int_as_float ⟩] → Lumi l
  | [⟨ 'Kwd "columns"; 'Kwd "="; 'Int n ⟩] → Columns n
  | [⟨ 'Kwd "triangle" ⟩] → Triangle true
  | [⟨ 'Kwd "notriangle" ⟩] → Triangle false
  | [⟨ 'Kwd "iterations"; 'Kwd "="; 'Int i ⟩] → Iterations i
  | [⟨ 'Kwd "events"; 'Kwd "="; 'String s ⟩] → Events s
  | [⟨ 'Kwd "binary" ⟩] → Binary true
  | [⟨ 'Kwd "ascii" ⟩] → Binary false

let rec channel_cmd_list = parser
  | [⟨ cmd = channel_cmd; cmd_list = channel_cmd_list ⟩] →
      cmd :: cmd_list
  | [⟨ ⟩] → []
```

```
type design_cmd =
  | Design of string
  | Roots of float
  | Design_Bins of int × coord
  | Channels of channel_cmd list
  | Comment of string

let design_cmd = parser
  | [⟨ 'Kwd "bins"; c = coord; 'Kwd "="; 'Int n ⟩] → Design_Bins (n, c)
  | [⟨ 'Kwd "design"; 'Kwd "="; 'String s ⟩] → Design s
  | [⟨ 'Kwd "roots"; 'Kwd "="; x = int_as_float ⟩] → Roots x
  | [⟨ 'Kwd "{"; cmds = channel_cmd_list; 'Kwd "}" ⟩] → Channels cmds
  | [⟨ 'Kwd "comment"; 'Kwd "="; 'String c ⟩] → Comment c

let rec design_cmd_list = parser
  | [⟨ cmd = design_cmd; cmd_list = design_cmd_list ⟩] →
      cmd :: cmd_list
  | [⟨ ⟩] → []

type file_cmd =
  | File of string
  | Block_data of string
  | Designs of design_cmd list

let file_cmd = parser
  | [⟨ 'Kwd "file"; 'Kwd "="; 'String s ⟩] → File s
  | [⟨ 'Kwd "block_data"; 'Kwd "="; 'String s ⟩] → Block_data s
  | [⟨ 'Kwd "{"; cmds = design_cmd_list; 'Kwd "}" ⟩] → Designs cmds

let rec file_cmd_list = parser
  | [⟨ cmd = file_cmd; cmd_list = file_cmd_list ⟩] →
      cmd :: cmd_list
  | [⟨ ⟩] → []

let rec file_cmds = parser
  | [⟨ 'Kwd "{"; cmd = file_cmd_list; 'Kwd "}";
       cmd_list = file_cmds ⟩] →
      cmd :: cmd_list
  | [⟨ ⟩] → []

type file_cmds = file_cmd list
```

A.19.4 Translate.

```
let rec update_fix acc = function
  | b, X12, s → update_fix (update_fix acc (b, X2, s)) (b, X1, s)
  | b, c, Minmax → update_fix (update_fix acc (b, c, Max)) (b, c, Min)
  | b, X1, Min → { acc with fixed_x1_min = b }
  | b, X1, Max → { acc with fixed_x1_max = b }
  | b, X2, Min → { acc with fixed_x2_min = b }
  | b, X2, Max → { acc with fixed_x2_max = b }

let rec update_pid acc = function
  | n, X12 → update_pid (update_pid acc (n, X2)) (n, X1)
  | n, X1 → { acc with pid1 = n }
  | n, X2 → { acc with pid2 = n }

let rec update_pol acc = function
  | n, X12 → update_pol (update_pol acc (n, X2)) (n, X1)
  | n, X1 → { acc with pol1 = n }
  | n, X2 → { acc with pol2 = n }

let rec update_bins acc = function
  | n, X12 → update_bins (update_bins acc (n, X2)) (n, X1)
  | n, X1 → { acc with bins1 = n }
  | n, X2 → { acc with bins2 = n }

let rec update_x_min acc = function
  | x, X12 → update_x_min (update_x_min acc (x, X2)) (x, X1)
  | x, X1 → { acc with x1_min = x }
  | x, X2 → { acc with x2_min = x }

let rec update_x_max acc = function
  | x, X12 → update_x_max (update_x_max acc (x, X2)) (x, X1)
  | x, X1 → { acc with x1_max = x }
  | x, X2 → { acc with x2_max = x }

let rec update_map acc = function
  | m, X12 → update_map (update_map acc (m, X2)) (m, X1)
  | m, X1 → { acc with intervals1 = m :: acc.intervals1 }
  | m, X2 → { acc with intervals2 = m :: acc.intervals2 }

let channel design cmds =
  List.fold_left
    (fun acc → function
      | Pid (n, c) → update_pid acc (n, c)
      | Pol (p, c) → update_pol acc (p, c)
```

```
      | Lumi l -> { acc with lumi = l }
      | Diffmap (m, c) -> update_map acc (m, c)
      | Bins (n, c) -> update_bins acc (n, c)
      | Xmin (x, c) -> update_x_min acc (x, c)
      | Xmax (x, c) -> update_x_max acc (x, c)
      | Fix (b, c, s) -> update_fix acc (b, c, s)
      | Triangle b -> { acc with triangle = b }
      | Iterations i -> { acc with iterations = i }
      | Events s -> { acc with events = s }
      | Columns n ->
          if n < 3 then
            invalid_arg "#columns < 3"
          else
            { acc with columns = n }
      | Binary b -> { acc with binary = b })
    default_channel cmds

let rec update_design_bins acc = function
  | n, X12 -> update_design_bins (update_design_bins acc (n, X2)) (n, X1)
  | n, X1 -> { acc with design_bins1 = n }
  | n, X2 -> { acc with design_bins2 = n }

let design cmds =
  List.fold_left
    (fun acc -> function
      | Design s -> { acc with design = s }
      | Roots r -> { acc with roots = r }
      | Design_Bins (n, c) -> update_design_bins acc (n, c)
      | Channels cmds ->
          { acc with channels = channel acc cmds :: acc.channels }
      | Comment c -> { acc with comments = c :: acc.comments })
    default_design cmds

let file cmds =
  List.fold_right
    (fun cmd acc ->
      match cmd with
      | File s -> { acc with name = s }
      | Block_data s -> { acc with block_data = Some s }
      | Designs cmds -> { acc with designs = design cmds :: acc.designs })
    cmds default_file
```

A.19.5 API

```
let parse_file name =
  let ic = open_in name in
  let tokens = lexer (Stream.of_channel ic) in
  let cmds = List.map file (file_cmds tokens) in
  close_in ic;
  cmds

let parse_string s =
  let tokens = lexer (Stream.of_string s) in
  List.map file (file_cmds tokens)
```

A.20 Interface of *Histogram*

```
type t
val create : int → float → float → t
val record : t → float → float → unit
val normalize : t → t
val to_channel : out_channel → t → unit
val to_file : string → t → unit
val as_bins_to_channel : out_channel → t → unit
val as_bins_to_file : string → t → unit

val regression : t → (float → bool) →
  (float → float) → (float → float) → float × float
```

A.21 Implementation of *Histogram*

```
open Printf

type t =
    { n_bins : int;
      n_bins_float : float;
      x_min : float;
      x_max : float;
      x_min_eps : float;
      x_max_eps : float;
      mutable n_underflow : int;
      mutable underflow : float;
      mutable underflow2 : float;
      mutable n_overflow : int;
```

```
    mutable overflow : float;
    mutable overflow2 : float;
    n : int array;
    w : float array;
    w2 : float array }
let create n_bins x_min x_max =
  let eps = 100. *. Float.Double.epsilon *. abs_float (x_max -. x_min) in
  { n_bins = n_bins;
    n_bins_float = float n_bins;
    x_min = x_min;
    x_max = x_max;
    x_min_eps = x_min -. eps;
    x_max_eps = x_max +. eps;
    n_underflow = 0;
    underflow = 0.0;
    underflow2 = 0.0;
    n_overflow = 0;
    overflow = 0.0;
    overflow2 = 0.0;
    n = Array.create n_bins 0;
    w = Array.create n_bins 0.0;
    w2 = Array.create n_bins 0.0 }
let record h x f =
  let i =
    truncate
      (floor (h.n_bins_float *. (x -. h.x_min) /. (h.x_max -. h.x_min))) in
  let i =
    if i < 0 ∧ x > h.x_min_eps then
      0
    else if i ≥ h.n_bins - 1 ∧ x < h.x_max_eps then
      h.n_bins - 1
    else
      i in
  if i < 0 then begin
    h.n_underflow ← h.n_underflow + 1;
    h.underflow ← h.underflow +. f;
    h.underflow2 ← h.underflow2 +. f *. f
  end else if i ≥ h.n_bins then begin
    h.n_overflow ← h.n_overflow + 1;
    h.overflow ← h.overflow +. f;
```

```
    h.overflow2 ← h.overflow2 +. f *. f
  end else begin
    h.n.(i) ← h.n.(i) + 1;
    h.w.(i) ← h.w.(i) +. f;
    h.w2.(i) ← h.w2.(i) +. f *. f
  end

let normalize h =
  let sum_w = Array.fold_left (+.) (h.underflow +. h.overflow) h.w in
  let sum_w2 = sum_w *. sum_w in
  { n_bins = h.n_bins;
    n_bins_float = h.n_bins_float;
    x_min = h.x_min;
    x_max = h.x_max;
    x_min_eps = h.x_min_eps;
    x_max_eps = h.x_max_eps;
    n_underflow = h.n_underflow;
    underflow = h.underflow /. sum_w;
    underflow2 = h.underflow2 /. sum_w2;
    n_overflow = h.n_overflow;
    overflow = h.overflow /. sum_w;
    overflow2 = h.overflow2 /. sum_w2;
    n = Array.copy h.n;
    w = Array.map (fun w' → w' /. sum_w) h.w;
    w2 = Array.map (fun w2' → w2' /. sum_w2) h.w2 }

let to_channel oc h =
  for i = 0 to h.n_bins - 1 do
    let x_mid = h.x_min
        +. (h.x_max -. h.x_min) *. (float i +. 0.5) /. h.n_bins_float in
    if h.n.(i) > 1 then
      let n = float h.n.(i) in
      let var1 = (h.w2.(i) /. n -. (h.w.(i) /. n) ** 2.0) /. (n -. 1.0)
      and var2 = h.w.(i) ** 2.0 /. (n *. (n -. 1.0)) in
      let var = var2 in
      fprintf oc "␣%.17E␣%.17E␣%.17E\n" x_mid h.w.(i) (sqrt var)
    else if h.n.(i) = 1 then
      fprintf oc "␣%.17E␣%.17E␣%.17E\n" x_mid h.w.(i) h.w.(i)
    else
      fprintf oc "␣%.17E␣%.17E\n" x_mid h.w.(i)
  done
```

```
let as_bins_to_channel oc h =
  for i = 0 to h.n_bins - 1 do
    let x_min = h.x_min
        +. (h.x_max -. h.x_min) *. (float i) /. h.n_bins_float
    and x_max = h.x_min
        +. (h.x_max -. h.x_min) *. (float i +. 1.0) /. h.n_bins_float in
    fprintf oc "␣%.17e␣%.17e\n" x_min h.w.(i);
    fprintf oc "␣%.17e␣%.17e\n" x_max h.w.(i)
  done

let to_file name h =
  let oc = open_out name in
  to_channel oc h;
  close_out oc

let as_bins_to_file name h =
  let oc = open_out name in
  as_bins_to_channel oc h;
  close_out oc
```

A.22 Naive Linear Regression

```
type regression_moments =
    { mutable n : int;
      mutable x : float;
      mutable y : float;
      mutable xx : float;
      mutable xy : float }

let init_regression_moments =
    { n = 0;
      x = 0.0;
      y = 0.0;
      xx = 0.0;
      xy = 0.0 }

let record_regression m x y =
  m.n <- m.n + 1;
  m.x <- m.x +. x;
  m.y <- m.y +. y;
  m.xx <- m.xx +. x *. x;
  m.xy <- m.xy +. x *. y
```

Minimize

$$f(a,b) = \sum_i w_i(ax_i + b - y_i)^2 = \langle (ax + b - y)^2 \rangle \tag{79}$$

i. e.

$$\frac{1}{2}\frac{\partial f}{\partial a}(a,b) = \langle x(ax + b - y) \rangle = a\langle x^2 \rangle + b\langle x \rangle - \langle xy \rangle = 0 \tag{80a}$$

$$\frac{1}{2}\frac{\partial f}{\partial b}(a,b) = \langle ax + b - y \rangle = a\langle x \rangle + b - \langle y \rangle = 0 \tag{80b}$$

and

$$a = \frac{\langle xy \rangle - \langle x \rangle \langle y \rangle}{\langle x^2 \rangle - \langle x \rangle^2} \tag{81a}$$

$$b = \langle y \rangle - a\langle x \rangle \tag{81b}$$

```
let linear_regression m =
  let n = float m.n in
  let x = m.x /. n
  and y = m.y /. n
  and xx = m.xx /. n
  and xy = m.xy /. n in
  let a = (xy -. x *. y) /. (xx -. x *. x) in
  let b = y -. a *. x in
  (a, b)
let regression h chi fx fy =
  let m = init_regression_moments in
  for i = 0 to h.n_bins - 1 do
    let x_mid = h.x_min
        +. (h.x_max -. h.x_min) *. (float i +. 0.5) /. h.n_bins_float in
    if chi x_mid then
      record_regression m (fx x_mid) (fy h.w.(i))
  done;
  linear_regression m
```

A.23 Implementation of *Circe2_tool*

A.23.1 Large Numeric File I/O

```
type input_file =
  | ASCII_ic of in_channel
  | ASCII_inf of string
  | Binary_inf of string
```

```
type output_file =
  | ASCII_oc of out_channel
  | ASCII_outf of string
  | Binary_outf of string

let read columns = function
  | ASCII_ic ic -> Events.of_ascii_channel columns ic
  | ASCII_inf inf -> Events.of_ascii_file columns inf
  | Binary_inf inf -> Events.of_binary_file columns inf

let write output array =
  match output with
  | ASCII_oc oc -> Events.to_ascii_channel oc array
  | ASCII_outf outf -> Events.to_ascii_file outf array
  | Binary_outf outf -> Events.to_binary_file outf array
```

The special case of writing a binary file with mapped I/O can be treated most efficiently:

```
let cat columns input output =
  match input, output with
  | ASCII_ic ic, Binary_outf outf ->
      ignore (Events.of_ascii_channel ~file:outf columns ic)
  | _, _ -> write output (read columns input)

let map_xy fx fy columns input output =
  let a = read columns input in
  for i2 = 1 to Bigarray.Array2.dim2 a do
    Bigarray.Array2.set a 1 i2 (fx (Bigarray.Array2.get a 1 i2));
    Bigarray.Array2.set a 2 i2 (fy (Bigarray.Array2.get a 2 i2))
  done;
  write output a

let log10_xy = map_xy log10 log10
let exp10_xy = map_xy (fun x -> 10.0 ** x) (fun y -> 10.0 ** y)
```

A.23.2 Histogramming

```
let scan_string s =
  let tokens = Events.lexer (Stream.of_string s) in
  let t1 = Events.next_float tokens in
  let t2 = Events.next_float tokens in
  let t3 = Events.next_float tokens in
  (t1, t2, t3)
```

```
let histogram_ascii name histograms =
  let ic = open_in name
  and histos =
    List.map (fun (tag, f, n, x_min, x_max) →
      (tag, f, Histogram.create n x_min x_max)) histograms in
  begin try
    while true do
      let x, y, w = scan_string (input_line ic) in
      List.iter (fun (_, f, h) → Histogram.record h (f x y) w) histos
    done
  with
  | End_of_file → ()
  end;
  close_in ic;
  List.map (fun (t, _, h) → (t, h)) histos

let histogram_binary_channel ic histograms =
  let histos =
    List.map (fun (tag, f, n, x_min, x_max) →
      (tag, f, Histogram.create n x_min x_max)) histograms in
  begin try
    while true do
      let x = Float.Double.input_binary_float ic
      and y = Float.Double.input_binary_float ic
      and w = Float.Double.input_binary_float ic in
      List.iter (fun (_, f, h) → Histogram.record h (f x y) w) histos
    done
  with
  | End_of_file → ()
  end;
  List.map (fun (t, _, h) → (t, h)) histos

let histogram_binary name histograms =
  let a = Events.of_binary_file 3 name
  and histos =
    List.map (fun (tag, f, n, x_min, x_max) →
      (tag, f, Histogram.create n x_min x_max)) histograms in
  for i2 = 1 to Bigarray.Array2.dim2 a do
    let x = Bigarray.Array2.get a 1 i2
    and y = Bigarray.Array2.get a 2 i2
    and w = Bigarray.Array2.get a 3 i2 in
    List.iter (fun (_, f, h) → Histogram.record h (f x y) w) histos
```

```
  done;
  List.map (fun (t, _, h) -> (t, h)) histos
let histogram_data to_file n reader suffix =
  let histograms = reader
      [ ("x", (fun x y -> x), n, 0.0, 1.0);
        ("x_low", (fun x y -> x), n, 0.0, 1.0 · 10^-4);
        ("1-x_low", (fun x y -> 1.0 -. x), n, 0.0, 1.0 · 10^-2);
        ("1-x_low2", (fun x y -> 1.0 -. x), n, 1.0 · 10^-10, 1.0 · 10^-2);
        ("y", (fun x y -> y), n, 0.0, 1.0);
        ("y_low", (fun x y -> y), n, 0.0, 1.0 · 10^-4);
        ("1-y_low", (fun x y -> 1.0 -. y), n, 0.0, 1.0 · 10^-2);
        ("1-y_low2", (fun x y -> 1.0 -. y), n, 1.0 · 10^-10, 1.0 · 10^-2);
        ("xy", (fun x y -> x *. y), n, 0.0, 1.0);
        ("xy_low", (fun x y -> x *. y), n, 0.0, 1.0 · 10^-8);
        ("z", (fun x y -> sqrt (x *. y)), n, 0.0, 1.0);
        ("z_low", (fun x y -> sqrt (x *. y)), n, 0.0, 1.0 · 10^-4);
        ("x-y", (fun x y -> x -. y), n, -1.0, 1.0);
        ("x_fine", (fun x y -> x), n, 0.75, 0.85);
        ("y_fine", (fun x y -> y), n, 0.75, 0.85);
        ("xy_fine", (fun x y -> x *. y), n, 0.5, 0.7);
        ("x-y_fine", (fun x y -> x -. y), n, -0.1, 0.1) ] in
  List.iter (fun (tag, h) ->
    to_file (tag ^ suffix) (Histogram.normalize h))
    histograms
```

A.23.3 Moments

```
let moments_ascii name moments =
  let ic = open_in name
  and f = Array.of_list (List.map (fun (tag, f) -> f) moments)
  and m = Array.of_list (List.map (fun (tag, f) -> 0.0) moments)
  and sum_w = ref 0.0 in
  begin try
    while true do
      let x, y, w = scan_string (input_line ic) in
      sum_w := !sum_w +. w;
      for i = 0 to Array.length f - 1 do
        m.(i) <- m.(i) +. w *. (f.(i) x y)
      done
```

```
    done
  with
  | End_of_file -> ()
  end;
  close_in ic;
  List.map2 (fun (tag, f) m -> (tag, m /. !sum_w)) moments (Array.to_list m)

let moments_binary name moments =
  let a = Events.of_binary_file 3 name in
  let f = Array.of_list (List.map (fun (tag, f) -> f) moments)
  and m = Array.of_list (List.map (fun (tag, f) -> 0.0) moments)
  and sum_w = ref 0.0 in
  for i2 = 1 to Bigarray.Array2.dim2 a do
    let x = Bigarray.Array2.get a 1 i2
    and y = Bigarray.Array2.get a 2 i2
    and w = Bigarray.Array2.get a 3 i2 in
    sum_w := !sum_w +. w;
    for i = 0 to Array.length f - 1 do
      m.(i) <- m.(i) +. w *. (f.(i) x y)
    done
  done;
  List.map2 (fun (tag, f) m -> (tag, m /. !sum_w)) moments (Array.to_list m)

let fmt var = function
  | 0 -> ""
  | 1 -> var
  | n -> var ^ "^" ^ string_of_int n

let moment nx ny =
  (fmt "x" nx ^ fmt "y" ny, (fun x y -> x ** (float nx) *. y ** (float ny)))

let diff_moment n =
  (fmt "|x-y|" n, (fun x y -> (abs_float (x -. y)) ** (float n)))

let moments_data reader =
  let moments = reader
      (List.map (moment 0) [1; 2; 3; 4; 5; 6] @
       List.map (moment 1) [0; 1; 2; 3; 4; 5] @
       List.map (moment 2) [0; 1; 2; 3; 4] @
       List.map (moment 3) [0; 1; 2; 3] @
       List.map (moment 4) [0; 1; 2] @
       List.map (moment 5) [0; 1] @
       List.map (moment 6) [0] @
       List.map diff_moment [1; 2; 3; 4; 5; 6]) in
```

```
List.iter (fun (tag, m) → Printf.printf "%s = %g\n" tag m) moments
```

A.23.4 Regression

```
let regression_interval (tag, h) (log_min, log_max) =
  let a, b =
    Histogram.regression h
      (fun x → x ≥ log_min ∧ x ≤ log_max) (fun x → x) (fun x →
log x) in
  Printf.printf "%g<%s<%g: a = %g, b = %g\n" log_min tag log_max a b

let intervals =
  [ (−7.0, −6.0);
    (−6.0, −5.0);
    (−5.0, −4.0);
    (−4.0, −3.0);
    (−3.0, −2.0);
    (−7.0, −5.0);
    (−6.0, −4.0);
    (−5.0, −3.0);
    (−4.0, −2.0);
    (−7.0, −4.0);
    (−6.0, −3.0);
    (−5.0, −2.0);
    (−7.0, −3.0);
    (−6.0, −2.0) ]

let intervals =
  [ (−7.0, −4.0);
    (−6.0, −3.0);
    (−7.0, −3.0);
    (−6.0, −2.0) ]

let regression_data n reader =
  let histograms = reader
      [ ("log(x1)", (fun x1 x2 → log x1), n, −8.0, 0.0);
        ("log(x2)", (fun x1 x2 → log x2), n, −8.0, 0.0) ] in
  List.iter (fun (tag, h) →
    List.iter (regression_interval (tag, h)) intervals) histograms
```

A.23.5 Visually Adapting Powermaps

```
let power_map beta eta =
  Diffmap.Power.create (1.0 /. (1.0 +. beta)) eta 0.0 1.0

let power_data to_file n center resolution reader suffix =
  let histograms = reader
      (List.flatten
         (List.map (fun p →
           let pm = power_map p 0.0 in
           let ihp = Diffmap.Power.ihp pm in
           [((Printf.sprintf "1-x_low.%.2f" p), (fun x1 x2 → ihp (1.0−
.x1)), n, 0.0, ihp 1.0 · 10^−4);
            ((Printf.sprintf "1-y_low.%.2f" p), (fun x1 x2 → ihp (1.0−
.x2)), n, 0.0, ihp 1.0 · 10^−4);
            ((Printf.sprintf "x_low.%.2f" p), (fun x1 x2 → ihp x1), n, 0.0, ihp 1.0·
10^−4);
            ((Printf.sprintf "y_low.%.2f" p), (fun x1 x2 → ihp x2), n, 0.0, ihp 1.0·
10^−4)])
            [center −. 2.0 ∗. resolution;
             center −. resolution; center; center +. resolution;
             center +. 2.0 ∗. resolution])) in
  List.iter (fun (tag, h) →
    to_file (tag ^ suffix) (Histogram.normalize h)) histograms
```

A.23.6 Testing

```
let make_test_data n (x_min, x_max) (y_min, y_max) f =
  let delta_x = x_max −. x_min
  and delta_y = y_max −. y_min in
  let array =
    Bigarray.Array2.create Bigarray.float64 Bigarray.fortran_layout 3 n in
  for i = 1 to n do
    let x = x_min +. Random.float delta_x
    and y = y_min +. Random.float delta_y in
    Bigarray.Array2.set array 1 i x;
    Bigarray.Array2.set array 2 i y;
    Bigarray.Array2.set array 3 i (f x y)
  done;
  array
```

```
module Div = Division.Mono
module Grid = Grid.Make (Div)

let test_design grid =
  let channel =
    { Grid.pid1 = 22; Grid.pol1 = 0;
      Grid.pid2 = 22; Grid.pol2 = 0;
      Grid.lumi = 0.0; Grid.g = grid } in
  { Grid.name = "TEST";
    Grid.roots = 500.0;
    Grid.channels = [ channel ];
    Grid.comments = [ "unphysical test" ]}

let test verbose triangle shrink nbins name f =
  let data = make_test_data 100000 (0.4, 0.9) (0.2, 0.7) f in
  let initial_grid =
    Grid.create ~triangle
      (Div.create nbins 0.0 1.0)
      (Div.create nbins 0.0 1.0) in
  let grid =
    Grid.of_bigarray ~verbose
      ~fixed_x1_min : (¬ shrink) ~fixed_x1_max : (¬ shrink)
      ~fixed_x2_min : (¬ shrink) ~fixed_x2_max : (¬ shrink)
      data initial_grid in
  Grid.designs_to_file name [test_design grid]

let random_interval () =
  let x1 = Random.float 1.0
  and x2 = Random.float 1.0 in
  (min x1 x2, max x1 x2)

module Test_Power = Diffmap.Make_Test (Diffmap.Power)
module Test_Resonance = Diffmap.Make_Test (Diffmap.Resonance)

let test_maps seed =
  Random.init seed;
  let x_min, x_max = random_interval ()
  and y_min, y_max = random_interval () in
  let alpha = 1.0 +. Random.float 4.0
  and eta =
    if Random.float 1.0 > 0.5 then
      y_max +. Random.float 5.0
    else
      y_min -. Random.float 5.0 in
```

```
  Test_Power.all
    (Diffmap.Power.create ~alpha ~eta ~x_min ~x_max y_min y_max);
  let a = Random.float 1.0
  and eta = y_min +. Random.float (y_max -. y_min) in
  Test_Resonance.all
    (Diffmap.Resonance.create ~eta ~a ~x_min ~x_max y_min y_max)
```

A.23.7 Main Program

```
type format = ASCII | Binary

type action =
  | Nothing
  | Command_file of string
  | Commands of string
  | Cat
  | Histo of format × string
  | Moments of format × string
  | Regression of format × string
  | Test of string × (float → float → float)
  | Test_Diffmaps of int
  | Log10
  | Exp10
  | Power of format × string

let _ =
  let usage = "usage: " ^ Sys.argv.(0) ^ " [options]" in
  let nbins = ref 100
  and triangle = ref false
  and shrink = ref true
  and verbose = ref false
  and action = ref Nothing
  and suffix = ref ".histo"
  and input = ref (ASCII_ic stdin)
  and output = ref (ASCII_oc stdout)
  and columns = ref 3
  and block_data = ref None
  and histogram_to_file = ref Histogram.to_file
  and center = ref 0.0
  and resolution = ref 0.01 in
  Arg.parse
```

```
[("-c", Arg.String (fun s → action := Commands s), "commands");
 ("-f", Arg.String (fun f → action := Command_file f), "command␣file");
 ("-ia", Arg.String (fun n → input := ASCII_inf n),
  "ASCII␣input␣file");
 ("-ib", Arg.String (fun n → input := Binary_inf n),
  "Binary␣input␣file");
 ("-oa", Arg.String (fun n → output := ASCII_outf n),
  "ASCII␣output␣file");
 ("-ob", Arg.String (fun n → output := Binary_outf n),
  "Binary␣output␣file");
 ("-cat", Arg.Unit (fun () →
    input := ASCII_ic stdin; output := ASCII_oc stdout;
    action := Cat), "copy␣stdin␣to␣stdout");
 ("-log10", Arg.Unit (fun () →
    input := ASCII_ic stdin; output := ASCII_oc stdout;
    action := Log10), "");
 ("-exp10", Arg.Unit (fun () →
    input := ASCII_ic stdin; output := ASCII_oc stdout;
    action := Exp10), "");
 ("-ha", Arg.String (fun s → action := Histo (ASCII, s)),
  "ASCII␣histogramming␣tests");
 ("-hb", Arg.String (fun s → action := Histo (Binary, s)),
  "binary␣histogramming␣tests");
 ("-ma", Arg.String (fun s → action := Moments (ASCII, s)),
  "ASCII␣moments␣␣tests");
 ("-mb", Arg.String (fun s → action := Moments (Binary, s)),
  "binary␣moments␣tests");
 ("-pa", Arg.String (fun s → action := Power (ASCII, s)), "");
 ("-pb", Arg.String (fun s → action := Power (Binary, s)), "");
 ("-C", Arg.Float (fun c → center := c), "");
 ("-R", Arg.Float (fun r → resolution := r), "");
 ("-Pa", Arg.String (fun s → action := Regression (ASCII, s)), "");
 ("-Pb", Arg.String (fun s → action := Regression (Binary, s)), "");
 ("-p", Arg.String (fun s → suffix := s), "histogram␣name␣suffix");
 ("-h", Arg.Unit (fun () →
    histogram_to_file := Histogram.as_bins_to_file), "");
 ("-bd", Arg.String (fun s → block_data := Some s), "");
 ("-b", Arg.Int (fun n → nbins := n), "#bins");
 ("-s", Arg.Set shrink, "shrinkwrap␣interval␣[default]");
 ("-S", Arg.Clear shrink, "don't␣shrinkwrap␣interval");
 ("-t", Arg.Set triangle,
```

```
    "project␣symmetrical␣distribution␣onto␣triangle");
   ("-v", Arg.Set verbose, "verbose");
   ("-test1", Arg.String (fun s →
     action := Test (s, fun x y → 1.0)), "testing");
   ("-test2", Arg.String (fun s →
     action := Test (s, fun x y → x *. y)), "testing");
   ("-test3", Arg.String (fun s →
     action := Test (s, fun x y → 1.0 /. x +. 1.0 /. y)), "testing");
   ("-testm", Arg.Int (fun seed → action := Test_Diffmaps seed),
    "testing␣maps") ]
  (fun names → prerr_endline usage; exit 2)
  usage;
begin try
  match !action with
  | Nothing → ()
  | Commands name → Commands.execute (Commands.parse_string name)
  | Command_file name → Commands.execute (Commands.parse_file name)
  | Histo (ASCII, name) →
      histogram_data !histogram_to_file !nbins
        (histogram_ascii name) !suffix
  | Histo (Binary, "-") →
      histogram_data !histogram_to_file !nbins
        (histogram_binary_channel stdin) !suffix
  | Histo (Binary, name) →
      histogram_data !histogram_to_file !nbins
        (histogram_binary name) !suffix
  | Moments (ASCII, name) → moments_data (moments_ascii name)
  | Moments (Binary, name) → moments_data (moments_binary name)
  | Power (ASCII, name) →
      power_data !histogram_to_file !nbins !center !resolution
        (histogram_ascii name) !suffix
  | Power (Binary, name) →
      power_data !histogram_to_file !nbins !center !resolution
        (histogram_binary name) !suffix
  | Regression (ASCII, name) → regression_data !nbins (histogram_ascii name)
  | Regression (Binary, name) → regression_data !nbins (histogram_binary name)
  | Cat → cat !columns !input !output
  | Log10 → log10_xy !columns !input !output
  | Exp10 → exp10_xy !columns !input !output
  | Test (name, f) → test !verbose !triangle !shrink !nbins name f
  | Test_Diffmaps seed → test_maps seed
```

```
with
| Commands.Parse_Error msg →
    Printf.eprintf "%s: parse error: %s\n" Sys.argv.(0) msg;
    exit 1
end;
exit 0
```

Identifiers

Refinements

⟨*API documentation* 176a⟩
⟨*Abstract interfaces* 56a⟩
⟨*Abstract types* 56c⟩
⟨*Allocate* `circe2_channel` *components* `*p` 51d⟩
⟨*Allocate* `circe2_division` *components* `*d` 53b⟩
⟨*Apply Jacobian for triangle map* 69c⟩
⟨*Apply Jacobian for triangle map (C)* 69d⟩
⟨*Apply Jacobian for* y_1 *map* 67b⟩
⟨*Apply Jacobian for* y_2 *map* 68a⟩
⟨*Apply Jacobian for* y_1 *map (C)* 67c⟩
⟨*Apply Jacobian for* y_2 *map (C)* 68b⟩
⟨*Bootstrap the* `x` *buffer* 185e⟩
⟨*Buffer* 178d⟩
⟨*Calculate* `yb1` 76b⟩
⟨*Calculate* `yb2` 76c⟩
⟨*Check for* `ECRIC2` 79c⟩
⟨*Check for* `ECRIC2` *(C)* 79d⟩
⟨*Check if* `design` *and* `fdesgn` *do* `match` 73⟩
⟨*Check magic number* 179a⟩
⟨*Check polarization support* 75a⟩
⟨*Complain and* `return` *iff* `ch` $=$ *NULL (C)* 59b⟩
⟨*Complain and* `return` *iff* `ic` ≤ 0 59a⟩
⟨*Copyleft notice* 45d⟩
⟨*Copyleft notice (C)* 47b⟩
⟨*Copyleft notice (C++)* 48c⟩
⟨*Data type declarations (C)* 49a⟩
⟨*Data type declarations (C++)* (never defined)⟩
⟨*Declaration: circe2 parameters* 48e⟩
⟨*Decode polarization support* 74b⟩
⟨*Do a binary search for* $\texttt{wgt(i}-1\texttt{)} \leq \texttt{u} < \texttt{wgt(i)}$ 59c⟩
⟨*Do a binary search for* $\texttt{yb1(i1}-1\texttt{)} \leq \texttt{y1} < \texttt{yb1(i1)}$ 69e⟩
⟨*Do a binary search for* $\texttt{yb2(i2}-1\texttt{)} \leq \texttt{y2} < \texttt{yb2(i2)}$ 70a⟩
⟨*Error codes for* `cir2ld` 71a⟩
⟨*Fill* `ranx` *from* `x` 187c⟩
⟨*Find free logical unit for* `lun` 79e⟩
⟨*Find* `ic` *for* `p1`, `h1`, `p2` *and* `h2` 58c⟩
⟨*Generate* `n` *unweighted events* 82a⟩
⟨*Generate* `n` *weighted events* 82b⟩
⟨*Get another* `chunk` *of integers* 181e⟩

⟨*Get another* `chunk` *of reals* 181a⟩
⟨*Get first* `chunk` *of integers* 181d⟩
⟨*Get first* `chunk` *of reals* 180b⟩
⟨*Inverse triangle map* 62a⟩
⟨*Inverse triangle map (C)* 62b⟩
⟨*Lags* `K`, `L` 182d⟩
⟨*Load* `a` *and refresh* `ranx` 183b⟩
⟨*Load* `a` *and refresh* `ranx` *(Fortran90)* 184a⟩
⟨*Load channel* `ic` 74d⟩
⟨*Load division* `xb1` 75c⟩
⟨*Load division* `xb2` 76a⟩
⟨*Load histograms* 74a⟩
⟨*Load weights* `wgt` *and* `val` 77a⟩
⟨*Local variables* 185d⟩
⟨*Local variables in* `cir2ld` 74c⟩
⟨*Macros (C)* 64a⟩
⟨*Magic number* `MAGIC0` 179b⟩
⟨*Modulus* `M` 178a⟩
⟨*Open* `name` *for reading on* `lun` 77c⟩
⟨*Other parameters* 185a⟩
⟨*Prepare array* `a` *and* `done`, `todo`, `chunk` 180a⟩
⟨*Private Procedures (C)* 47c⟩
⟨*Procedures* 55e⟩
⟨*Procedures (C++)* (never defined)⟩
⟨*Public Procedures (C)* 57a⟩
⟨*Public procedures* 48d⟩
⟨*Public types* 56b⟩
⟨*RNG dummy arguments* 55b⟩
⟨*RNG dummy declarations* 55c⟩
⟨*RNG: generate* `u` 55d⟩
⟨*Reset the buffer* 188a⟩
⟨*Sample procedures* 83c⟩
⟨*Sample procedures: public* 83b⟩
⟨*Separator* 45c⟩
⟨*Separator (C)* 47a⟩
⟨*Separator (C++)* 48b⟩
⟨*Separator2 (C)* 46b⟩
⟨*Separator2 (C++)* 48a⟩
⟨*Set* `n` *from* `luxury` 182c⟩
⟨*Set up* `s` *and* `t` 185g⟩
⟨*Shift* `s` *or* `t` 187b⟩

⟨*Skip comments until* `CIRCE2` 78b⟩
⟨*Skip comments until* `CIRCE2` *(C)* 78c⟩
⟨*Skip data until* `ECRIC2` 79a⟩
⟨*Skip data until* `ECRIC2` *(C)* 79b⟩
⟨*State* 183c⟩
⟨*Step* `i` *and reload* `a` *iff necessary* 178e⟩
⟨*Swap* `y1` *and* `y2` *in 50% of the cases* 62c⟩
⟨*Swap* `y1` *and* `y2` *in 50% of the cases (C)* 62d⟩
⟨*Test support for density matrices* 70d⟩
⟨*Unused sample procedures* 83a⟩
⟨*Update* `i`, `done` *and* `todo` *and set new* `chunk` 180c⟩
⟨*Verify* `seed` 185b⟩
⟨*Well known constants (C)* 58b⟩
⟨*Write channel data* 85a⟩
⟨*Write channel header* 84b⟩
⟨*Write design/beam data* 84a⟩
⟨`block data cir2bd` *template* 87b⟩
⟨*1-byte aligned part of* `/cir2cd/` 86d⟩
⟨*4-byte aligned part of* `/cir2cd/` 86c⟩
⟨*8-byte aligned part of* `/cir2cd/` 86e⟩
⟨*4-byte aligned part of circe2 parameters* 51a⟩
⟨*8-byte aligned part of circe2 parameters* 50b⟩
⟨`/cir2cd/` 86a⟩
⟨`circe2_c.c` 46a⟩
⟨`circe2_channel` *components* 51c⟩
⟨`circe2_channels` *components* 57c⟩
⟨`circe2_cpp.cc` 47d⟩
⟨`circe2d.f90` 85b⟩
⟨`circe2_division` *components* 53a⟩
⟨`circe2.f90` 45b⟩
⟨`circe2ls.f90` 83d⟩
⟨`circe2_sample_binary.f90` 81⟩
⟨`circe2_sample.f90` 80b⟩
⟨`/cir2cm/` 87c⟩
⟨`i` $\leftarrow$ (`i1`, `i2`) 51e⟩
⟨`i` $\leftarrow$ (`i1`, `i2`) *(C)* 51f⟩
⟨(`i1`, `i2`) $\leftarrow$ `i` 51g⟩
⟨`implicit none` 45a⟩
⟨`parameter` *declarations* 51b⟩
⟨`parameter` *part of* `/cir2cm/` 88⟩
⟨$p(z) \to p(z)^2$ (*modulo* 2 *and* $z^{100} + z^{37} + 1$) 186a⟩

⟨$p(z) \rightarrow zp(z)$ (*modulo* 2 *and* $z^{100} + z^{37} + 1$) 187a⟩
⟨sample_taorng.f90 188b⟩
⟨subroutine cir2lb 85c⟩
⟨taorng.f90 177b⟩
⟨taorng.f90*: public* 178b⟩
⟨taorng.f90*: subroutines* 178c⟩
⟨write3_ascii 82d⟩
⟨write3_ascii*: public* 82c⟩
⟨write3_binary.c 82e⟩
⟨x1 ∈ [ch− > d1− > x[i1 − 1] , ch− > d1− > x[i1]] *(C)* 60d⟩
⟨x2 ∈ [ch− > d2− > x[i2 − 1] , ch− > d2− > x[i2]] *(C)* 60e⟩
⟨x1 ∈ [xb1(i1 − 1), xb1(i1)] 60b⟩
⟨x2 ∈ [xb2(i2 − 1), xb2(i2)] 60c⟩
⟨y1 ← x1 61a⟩
⟨y2 ← x2 61b⟩
⟨y1 ← x1 *(C)* 61c⟩
⟨y2 ← x2 *(C)* 61d⟩
⟨(y1, y2) ← (yy1, yy2) 69a⟩
⟨(y1, y2) ← (yy1, yy2) *(C)* 69b⟩

Index

B `taorng`: The Portable Random Number Generator from The Art of Computer Programming

B.1 User's Manual

The second editions of volume two of Donald E. Knuth' *The Art of Computer Programming* [12] had always been celebrated as a prime reference for random number generation. In 1996, Don Knuth made his notes for future editions available on the internet in the form of errata, which have in the meantime have been incorporated in the third edition.

These notes contain a gem of a *portable* random number generator. It generates 30-bit integers with the following desirable properties

- they pass all the tests from George Marsaglia's "diehard" suite of tests for random number generators [13].
- they can be generated with portable signed 32-bit arithmetic (Fortran can't do unsigned arithmetic)
- it is faster than other lagged Fibonacci generators
- it can create at least $2^{30} - 2$ independent sequences

Functions returning single reals and integers:

176a ⟨*API documentation* 176a⟩≡

```
double precision u
integer i
call taornu (u)
call taorni (i)
```

Subroutines filling arrays of reals and integers:

176b ⟨*API documentation* 176a⟩+≡

```
integer i(n), n
integer u(n)
subroutine taornv (i, n)
subroutine taornj (u, n)
```

Subroutine for changing the seed:

176c ⟨*API documentation* 176a⟩+≡

```
integer seed
subroutine taorns (seed)
```

Subroutine for changing the luxury:

177a ⟨*API documentation* 176a⟩+≡

```
integer luxury
subroutine taornl (luxury)
```

B.2 Implementation

177b ⟨taorng.f90 177b⟩≡

```
! $Id: postlude.nw,v 1.24 2001/10/30 11:47:54 ohl Exp $
!   Copyright (C) 1996-2011 by
!       Wolfgang Kilian <kilian@physik.uni-siegen.de>
!       Thorsten Ohl <ohl@physik.uni-wuerzburg.de>
!       Juergen Reuter <juergen.reuter@desy.de>
!       Christian Speckner <cnspeckn@googlemail.com>
!
!   Taorng is free software; you can redistribute it and/or modify it
!   under the terms of the GNU General Public License as published by
!   the Free Software Foundation; either version 2, or (at your option)
!   any later version.
!
!   Taorng is distributed in the hope that it will be useful, but
!   WITHOUT ANY WARRANTY; without even the implied warranty of
!   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
!   GNU General Public License for more details.
!
!   You should have received a copy of the GNU General Public License
!   along with this program; if not, write to the Free Software
!   Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
module tao_rng
  use kinds

  implicit none
  private

⟨taorng.f90: public 178b⟩

contains

⟨taorng.f90: subroutines 178c⟩
end module tao_rng
```

B.2.1 High Level Routines

Single Random Numbers A modulus of 2^{30} is the largest we can handle in *portable* (i.e. *signed*) 32-bit arithmetic

178a ⟨*Modulus* `M` 178a⟩≡
```
integer , parameter :: M = 2**30
```

A random integer r with $0 \le r < 2^{30} = 1073741824$:

178b ⟨`taorng.f90`*: public* 178b⟩≡
```
public :: taorni
```

178c ⟨`taorng.f90`*: subroutines* 178c⟩≡
```
subroutine taorni (r)
  integer :: r
  ⟨Buffer 178d⟩
  ⟨Check magic number 179a⟩
  ⟨Step i and reload a iff necessary 178e⟩
  r = a(i)
end subroutine taorni
```

The low level routine `taorng` will fill an array $a_1, \ldots, a_N$, which will be consumed and refilled like an input buffer. We need at least $N \ge K = 100$ for the call to `taorng`. Higher values don't change the results, but make `taorng` more efficient (about a factor of two, asymptotically). DEK recommends $N \ge 1000$.

178d ⟨*Buffer* 178d⟩≡
```
integer, parameter :: NN = 1000
integer, dimension(NN), save :: a(NN)
integer :: i, n
!!! common /taornb/ a, i, n
!!! save /taornb/
```

Increment the index `i` and reload the array `a`, iff this buffer is exhausted. Throughout these routines, `i` will point to random number that has just been consumed. For the array filling routines below, this is simpler than pointing to the next waiting number.

178e ⟨*Step* `i` *and reload* `a` *iff necessary* 178e⟩≡
```
i = i + 1
if (i .gt. n) then
   call taorng (a, NN)
   i = 1
end if
```

We add an integer to the common block which we can check for initialization by filling it with an unlikely "magic" number.

178f ⟨*Buffer* 178d⟩+≡

⟨*Magic number* `MAGIC0` 179b⟩
```
integer :: magic
!!! common /taornb/ magic
```

Iff the magic number is not found, trigger a buffer refill.

179a ⟨*Check magic number* 179a⟩≡
```
if (magic .ne. MAGIC0) then
   n = NN
   i = n
   magic = MAGIC0
end if
```

Incidentally, the magic number is the date on which DEK wrote down the routines discussed below.

179b ⟨*Magic number* `MAGIC0` 179b⟩≡
```
integer MAGIC0
parameter (MAGIC0 = 19950826)
```

A random real $r \in [0, 1)$. This is almost identical to `taorni`, but we duplicate the code to avoid the function call overhead for speed.

179c ⟨`taorng.f90`*: public* 178b⟩+≡
```
public :: taornu
```

179d ⟨`taorng.f90`*: subroutines* 178c⟩+≡
```
subroutine taornu (r)
  real(kind=double) :: r
  ⟨Modulus M 178a⟩
  real(kind=double), parameter :: INVM = 1D0/M
  ⟨Buffer 178d⟩
  ⟨Check magic number 179a⟩
  ⟨Step i and reload a iff necessary 178e⟩
  r = INVM * a(i)
end subroutine taornu
```

Arrays of Random Numbers Fill the array $v_1, \ldots, v_\nu$ with uniform deviates $v_i \in [0, 1)$. This has to be done such that the underlying array length in `taorng` is transparent to the user. At the same time we want to avoid the overhead of calling `taornu` ν times.

179e ⟨`taorng.f90`*: public* 178b⟩+≡
```
public :: taornv
```

179f ⟨`taorng.f90`*: subroutines* 178c⟩+≡
```
subroutine taornv (v, nu)
  real(kind=double), dimension(:) :: v
```

```
integer :: nu
⟨Modulus M 178a⟩
real(kind=double), parameter :: INVM = 1D0/M
integer :: done, todo, chunk, k
⟨Buffer 178d⟩
⟨Check magic number 179a⟩
⟨Prepare array a and done, todo, chunk 180a⟩
⟨Get first chunk of reals 180b⟩
do
  ⟨Update i, done and todo and set new chunk 180c⟩
  if (chunk .le. 0) then
     exit
  else
     ⟨Get another chunk of reals 181a⟩
  end if
end do
end subroutine taornv
```

`i` is used as an offset into the buffer `a`, as usual. `done` is an offset into the target. We still have to process all `nu` numbers. The first chunk can only use what's left in the buffer.

180a ⟨*Prepare array* `a` *and* `done`, `todo`, `chunk` 180a⟩≡

```
if (i .ge. n) then
   call taorng (a, NN)
   i = 0
endif
done = 0
todo = nu
chunk = min (todo, n - i)
```

180b ⟨*Get first* `chunk` *of reals* 180b⟩≡

```
do k = 1, chunk
   v(k) = INVM * a(i+k)
end do
```

This logic is a bit weird, but after the first chunk, `todo` will either vanish (in which case we're done) or we have consumed all of the buffer and must reload. In any case we can pretend that the next chunk can use the whole buffer.

180c ⟨*Update* `i`, `done` *and* `todo` *and set new* `chunk` 180c⟩≡

```
i = i + chunk
done = done + chunk
todo = todo - chunk
chunk = min (todo, n)
```

As just mentioned, when we get here, we have to reload.

181a ⟨*Get another* `chunk` *of reals* 181a⟩≡

```
call taorng (a, NN)
i = 0
do k = 1, chunk
   v(done+k) = INVM * a(k)
end do
```

Fill the array $j_1, \ldots, j_\nu$ with random integers $0 \le j_i < 2^{30} = 1073741824$. Again, this has to be done such that the underlying array length in `taorng` is transparent to the user and we want to avoid the overhead of calling `taorni` ν times.

181b ⟨`taorng.f90`*: public* 178b⟩+≡

```
public :: taornj
```

181c ⟨`taorng.f90`*: subroutines* 178c⟩+≡

```
subroutine taornj (j, nu)
  integer, dimension(:) :: j
  integer :: nu
  integer :: done, todo, chunk, k
  ⟨Buffer 178d⟩
  ⟨Check magic number 179a⟩
  ⟨Prepare array a and done, todo, chunk 180a⟩
  ⟨Get first chunk of integers 181d⟩
  do
    ⟨Update i, done and todo and set new chunk 180c⟩
    if (chunk .le. 0) then
       exit
    else
       ⟨Get another chunk of integers 181e⟩
    end if
  end do
end subroutine taornj
```

181d ⟨*Get first* `chunk` *of integers* 181d⟩≡

```
do k = 1, chunk
   j(k) = a(i+k)
end do
```

181e ⟨*Get another* `chunk` *of integers* 181e⟩≡

```
call taorng (a, NN)
i = 0
do k = 1, chunk
   j(done+k) = a(k)
end do
```

182a ⟨taorng.f90: *public* 178b⟩+≡
```
public :: taornl
```

182b ⟨taorng.f90: *subroutines* 178c⟩+≡
```
subroutine taornl (luxury)
  integer :: luxury
  ⟨Buffer 178d⟩
  ⟨Check magic number 179a⟩
  ⟨Set n from luxury 182c⟩
end subroutine taornl
```

182c ⟨*Set* n *from* luxury 182c⟩≡
```
if (luxury .gt. NN) then
   print *, 'taornl: luxury ', luxury, ' too high!'
   print *, 'taornl: will use 1 random number out of ', NN, '!'
   n = 1
else if (luxury .lt. 1) then
   print *, 'taornl: luxury ', luxury, ' invalid!'
   print *, 'taornl: will use every random number!'
   n = NN
else
   n = NN / luxury
end if
i = min (i, n)
```

B.2.2 30-Bit Low Level Routines

Generation

$$X_j = (X_{j-100} - X_{j-37}) \mod 2^{30} \tag{82}$$

182d ⟨*Lags* K, L 182d⟩≡
```
integer, parameter :: K = 100, L = 37
```

Fill the array $a_1, \ldots, a_n$ with random integers $0 \le a_i < 2^{30}$. As mentioned above, we *must* have $n \ge K$, while higher values don't change the results and make thing more efficient. Since users are not expected to call `taorng` directly, we do *not* check for $n \ge K$ and assume that the caller knows what (s)he's doing ...

182e ⟨taorng.f90: *public* 178b⟩+≡
```
public :: taorng
```

183a ⟨taorng.f90: *subroutines* 178c⟩+≡

```
subroutine taorng (a, n)
  integer :: n
  integer, dimension(n) :: a
  ⟨Lags K, L 182d⟩
  ⟨Modulus M 178a⟩
  ⟨State 183c⟩
  integer :: j
  ⟨Load a and refresh ranx 183b⟩
end subroutine taorng
```

First make sure that `taorns` has been called to initialize the generator state:

183b ⟨*Load* `a` *and refresh* `ranx` 183b⟩≡

```
if (magic .ne. MAGIC0) call taorns (0)
```

183c ⟨*State* 183c⟩≡

```
⟨Magic number MAGIC0 179b⟩
integer ranx(K), magic
common /taornc/ ranx, magic
save /taornc/
```

`ranx(1:K)` is already set up properly:

183d ⟨*Load* `a` *and refresh* `ranx` 183b⟩+≡

```
do 10 j = 1, K
   a(j) = ranx(j)
10 continue
```

The remaining $n - K$ random numbers can be gotten directly from the recursion (82):

183e ⟨*Load* `a` *and refresh* `ranx` 183b⟩+≡

```
do 11 j = K+1, n
   a(j) = a(j-K) - a(j-L)
   if (a(j) .lt. 0) a(j) = a(j) + M
11 continue
```

Do the recursion (82) K more times to prepare `ranx(1:K)` for the next invokation of `taorng`.

183f ⟨*Load* `a` *and refresh* `ranx` 183b⟩+≡

```
do 20 j = 1, L
   ranx(j) = a(n+j-K) - a(n+j-L)
   if (ranx(j) .lt. 0) ranx(j) = ranx(j) + M
20 continue
   do 21 j = L+1, K
   ranx(j) = a(n+j-K) - ranx(j-L)
   if (ranx(j) .lt. 0) ranx(j) = ranx(j) + M
```

```
21    continue
```

In the future, we will be able to save four of of the above 15 lines, by using Fortran90's array assignments. While the terse syntax might be a matter of taste, it is certainly useful for suggesting aggressive optimizations to the compiler. The two other loops implement the lagged Fibonacci and *can not* be replaced by array asignments because the assignments overlap.

184a ⟨*Load* `a` *and refresh* `ranx` *(Fortran90)* 184a⟩≡

```
a(1:K) = ranx(1:K)
do j = K+1, n
  a(j) = a(j-K) - a(j-L)
  if (a(j) < 0) a(j) = a(j) + M
end do
ranx(1:L) = a(n+1-K:n+L-K) - a(n+1-L:n)
where (ranx(1:L) < 0) ranx(1:L) = ranx(1:L) + M
do j = L+1, K
  ranx(j) = a(n+j-K) - ranx(j-L)
  if (ranx(j) < 0) ranx(j) = ranx(j) + M
end do
```

Initialization The non-trivial and most beautiful part is the algorithm to initialize the random number generator state `ranx` with the first 100 numbers. I haven't studied algebra over finite fields in sufficient depth to consider the mathematics behind it straightforward. The commentary below is rather verbose and reflects my understanding of DEK's rather terse remarks (solution to exercise 3.6-8 [12]).

184b ⟨`taorng.f90`*: public* 178b⟩+≡

```
public :: taorns
```

184c ⟨`taorng.f90`*: subroutines* 178c⟩+≡

```
subroutine taorns (seedin)
  integer, intent(in) :: seedin
  integer :: seed
  ⟨Lags K, L 182d⟩
  ⟨Modulus M 178a⟩
  ⟨Other parameters 185a⟩
  ⟨State 183c⟩
  ⟨Local variables 185d⟩
  ⟨Verify seed 185b⟩
  ⟨Bootstrap the x buffer 185e⟩
  ⟨Set up s and t 185g⟩
  do
      ⟨p(z) → p(z)^2 (modulo 2 and z^100 + z^37 + 1) 186a⟩
```

```
⟨p(z) → zp(z) (modulo 2 and z^100 + z^37 + 1) 187a⟩
⟨Shift s or t 187b⟩
if (t .le. 0) exit
end do
⟨Fill ranx from x 187c⟩
magic = MAGIC0
end subroutine taorns
```

185a ⟨*Other parameters* 185a⟩≡

```
integer, parameter :: SEEDMX = 2**30 - 3
```

185b ⟨*Verify* seed 185b⟩≡

```
seed = seedin
if ((seed .lt. 0) .or. (seed .gt. SEEDMX)) then
   print *, 'taorns: seed (', seed, ') not in [0,', SEEDMX, ']!'
   seed = mod (abs (seed), SEEDMX+1)
   print *, 'taorns: seed set to ', seed, '!'
end if
```

185c ⟨*Other parameters* 185a⟩+≡

```
integer, parameter :: TT= 70, KK = K+K-1
```

185d ⟨*Local variables* 185d⟩≡

```
integer x(KK), j, s, t
```

Fill the array $x_1, \ldots, x_{K=100}$ with even integers, shifted cyclically by 29 bits.

185e ⟨*Bootstrap the* x *buffer* 185e⟩≡

```
s = seed - mod (seed, 2) + 2
do j = 1, K
   x(j) = s
   s = s + s
   if (s .ge. M) s = s - M + 2
end do
do j = K+1, KK
   x(j) = 0
end do
```

Make x_2 (and only x_2) odd:

185f ⟨*Bootstrap the* x *buffer* 185e⟩+≡

```
x(2) = x(2) + 1
```

185g ⟨*Set up* s *and* t 185g⟩≡

```
s = seed
t = TT - 1
```

Consider the polynomial

$$p(z) = \sum_{n=1}^{K} x_n z^{n-1} = x_{100} z^{99} + \ldots + x_2 z + x_1 \tag{83}$$

We have $p(z)^2 = p(z^2) \mod 2$ because cross terms have an even coefficient and $x_n^2 = x_n \mod 2$. Therefore we can square the polynomial by shifting the coefficients. The coefficients for $n > K$ will be reduced $\mod 2$ below.

186a ⟨$p(z) \to p(z)^2$ (*modulo* 2 *and* $z^{100} + z^{37} + 1$) 186a⟩≡

```
do j = K, 2, -1
   x(j+j-1) = x(j)
end do
```

The coefficients of the odd powers (those with the even indices) have not been changed yet. Set them to a flipped version

$$\begin{aligned} x_2 &\leftarrow \quad \text{even } x_{2K-1=199} \\ x_4 &\leftarrow \quad \text{even } x_{2K-3=197} \\ &\ldots \\ x_{K+L-1=136} &\leftarrow \quad \text{even } x_{K-L+2=65} \end{aligned} \tag{84}$$

of the other coefficients with the least significant bit set to 0.

DEK's notes contain an (insignificant) typo here `do 21 j = KK, K-L+1, -2`, because `K` is even and `L` is odd. (If it is on purpose to accomodate simultaneously odd lags, then the C version is wrong.)

186b ⟨$p(z) \to p(z)^2$ (*modulo* 2 *and* $z^{100} + z^{37} + 1$) 186a⟩+≡

```
do j = KK, K-L+2, -2
   x(KK-j+2) = x(j) - mod (x(j), 2)
end do
```

Let's return to the coefficients for $n > K$ generated by the shifting above. Subtract $z^n(z^{100} + z^{37} + 1)$ iff the coefficient of $z^n z^{100}$ doesn't vanish $\mod 2$ after squaring. The coefficient of $z^n z^{100}$ is left alone, because it doesn't belong to $p(z)$ anyway.

186c ⟨$p(z) \to p(z)^2$ (*modulo* 2 *and* $z^{100} + z^{37} + 1$) 186a⟩+≡

```
do j = KK, K+1, -1
   if (mod (x(j), 2) .eq. 1) then
      x(j-(K-L)) = x(j-(K-L)) - x(j)
      if (x(j-(K-L)) .lt. 0) x(j-(K-L)) = x(j-(K-L)) + M
      x(j-K) = x(j-K) - x(j)
      if (x(j-K) .lt. 0) x(j-K) = x(j-K) + M
   end if
end do
```

187a ⟨$p(z) \to zp(z)$ (*modulo* 2 *and* $z^{100} + z^{37} + 1$) 187a⟩≡

```
if (mod (s, 2) .eq. 1) then
   do j = K, 1, -1
      x(j+1) = x(j)
   end do
   x(1) = x(K+1)
   if (mod (x(K+1), 2) .eq. 1) then
      x(L+1) = x(L+1) - x(K+1)
      if (x(L+1) .lt. 0) x(L+1) = x(L+1) + M
   end if
end if
```

187b ⟨*Shift* `s` *or* `t` 187b⟩≡

```
if (s .ne. 0) then
   s = s / 2
else
   t = t - 1
end if
```

187c ⟨*Fill* `ranx` *from* `x` 187c⟩≡

```
do j = 1, L
   ranx(j+K-L) = x(j)
end do
do j = L+1, K
   ranx(j-L) = x(j)
end do
```

B.2.3 Testing

187d ⟨`taorng.f90`*: public* 178b⟩+≡

```
public :: taornt
```

187e ⟨`taorng.f90`*: subroutines* 178c⟩+≡

```
subroutine taornt ()
  implicit none
  ⟨Buffer 178d⟩
  integer :: j, r
  integer, dimension(10), save :: expect = &
    (/ 640345214,  443605255,  411993687, 618952382, 123106306, &
       949854402,  429877922,  261135009, 574783260, 1043288376 /)
  write (*, 100) 'testing taorng ...'
  write (*, 100) '  call taornl (luxury=1)'
100  format (1X, A)
  call taornl (1)
  write (*, 100) '  call taorns (seed=0)'
```

```
      call taorns (0)
      ⟨Reset the buffer 188a⟩
      print *, '  10000 warmup calls to taorni'
      do j = 1, 10000
         call taorni (r)
      end do
      do j = 1, 10
         call taorni (r)
         if (r .eq. expect(j)) then
            write (*, 101) 10000+j, r
101          format (3X, I5, ': ', I10, ' OK.')
          else
             write (*, 102) 10000+j, r, expect(j)
102          format (3X, I5, ': ', I10, ' not OK, (expected ', I10, ')!')
           end if
        end do
        write (*, 100) 'done.'
        stop
      end subroutine taornt
```

This is crucial, in case the buffer has been used:

188a ⟨*Reset the buffer* 188a⟩≡

```
      i = N
```

B.3 Testing

188b ⟨sample_taorng.f90 188b⟩≡

```
    program circe2_taotst
      use kinds
      use tao_rng

      implicit none
      integer :: i, N, S
      real(kind=double) :: r
      real(kind=double) :: sum30
      call taornt ()
      S = 0
      N = 10000000
      sum30 = 0
      call taorns (S)
      do i = 1, N
         call taornu (r)
```

```
    sum30 = sum30 + r
  end do
  print *, 'sum30 = ', sum30
end program circe2_taotst
```