Κίρκη Version 2.0: Beam Spectra for Simulating Linear Collider Physics

Thorsten Ohl*

Institute for Theoretical Physics and Astrophysics
Würzburg University
Campus Hubland Nord
Emil-Hilb-Weg 22
97074 Würzburg
Germany

June 2014
DRAFT: 01/02/2019, 07:06

Abstract

...

*e-mail: ohl@physik.uni-wuerzburg.de

Caveat

This manual is outdated and describes the old Fortran77 interface. This interface has been replaced by a similar, but thoroughly modern Fortran 2003 interface.

Also the new smoothing feature of `circe2_tool` ist not described here.

Please see the annotated template and other files in `share/examples/` for a starting point for rolling your own beam descriptions.

Program Summary:

- **Title of program:** K$\acute{\imath}\rho\kappa\eta$, Version 2.0 (June 2014)
- **Program obtainable** from `http://www.hepforge.org/downloads/whizard`.
- **Licensing provisions:** Free software under the GNU General Public License.
- **Programming languages used:** Fortran, OCaml[8] (available from `http://caml.inria.fr/ocaml` and `http://ocaml.org`).
- **Number of program lines in distributed program** $\approx$??? lines of Fortran (excluding comments) for the library; $\approx$??? lines of OCaml for the utility program
- **Computer/Operating System:** Any with a Fortran 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.

Contents

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.

K*ίρκη* Version 1.x (`circe1` for short) [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. K*ίρκη* 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 make the maintenance of parameterizations as FORTRAN77 `BLOCK DATA` unwieldy.

K*ίρκη* Version 2.0 (`circe2` for short) 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

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

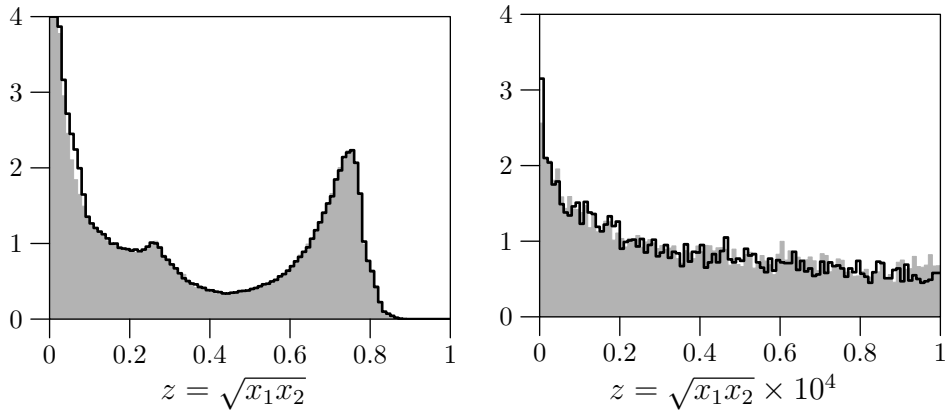

Figure 1: Comparison of a simulated realistic $\gamma\gamma$ luminosity spectrum (helicities: $(+,+)$) for a 500 GeV photon collider at TESLA [7] (filled area) with its `circe2` parameterization (solid line) using 50 bins in both directions. The 10^4-fold blow-up of the $z \to 0$ region is taken from the same pair of datasets as the plot including the large z region.

modeled on the implementation in VAMP [6], but changes were required for sampling static event sets instead of distributions given as functions. The 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 FORTRAN77 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.

FORTRAN77 has been chosen solely for practical reasons: at the time of writing, the majority of programs expected to use the `circe2` are legacy applications written in FORTRAN77. The simple functionality of the FORTRAN77 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

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

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. 14)
distributions	luminosity	`cir2lm`	(p. 16)
	probability density	`cir2dn`	(p. 19)
	density matrix	`cir2dm`	(extension, p. 21)
event generation	flavors/helicities	`cir2ch`	(p. 17)
	(x_1, x_2)	`cir2gn`	(p. 17)
	general polarization	`cir2gp`	(extension, p. 19)
internal	current beam	`/cir2cm/`	(p. 21)
	beam data base	`cir2bd`	(optional, p. 21)
	(cont'd)	`/cir2cd/`	(optional, p. 21)

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

`double precision 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 FORTRAN77 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 FORTRAN77 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.

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
      implicit none
      integer p1, h1, p2, h2, n, nevent, ierror
      double precision x1, x2
      external random
      nevent = 20
      ierror = 1
      call cir2ld ('default.circe', '*', 500D0, ierror)
      if (ierror .lt. 0) stop
      write (*, '(A7,4(X,A4),2(X,A10))')
     $     '#', 'pdg1', 'hel1', 'pdg2', 'hel2', 'x1', 'x2'
      do 10 n = 1, nevent
        call cir2ch (p1, h1, p2, h2, random)
        call cir2gn (p1, h1, p2, h2, x1, x2, random)
        write (*, '(I7,4(X,I4),2(X,F10.8))') n, p1, h1, p2, h2, x1, x2
 10   continue
      end
```

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)
      implicit none
      double precision r
      integer M, A, C
      parameter (M = 259200, A = 7141, C = 54773)
      integer n
      save n
      data n /0/
      n = mod (n*A + C, M)
```

```
      r = dble (n) / dble (M)
      end
```

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
      implicit none
      integer i1, i2, pdg(3), h1, h2, i, n, nevent, nev, ierror
      double precision x1, x2, lumi, cir2lm
      external random, cir2lm
      data pdg /22, 11, -11/
      nevent = 20
      ierror = 1
      call cir2ld ('default.circe', '*', 500D0, ierror)
      if (ierror .lt. 0) stop
      lumi = cir2lm (0, 0, 0, 0)
      write (*, '(A7,4(X,A4),2(X,A10))')
     $     '#', 'pdg1', 'hel1', 'pdg2', 'hel2', 'x1', 'x2'
      i = 0
      do 10 i1 = 1, 3
        do 11 i2 = 1, 3
          do 12 h1 = -1, 1, 2
            do 13 h2 = -1, 1, 2
              nev = nevent * cir2lm (pdg(i1), h1, pdg(i2), h2) / lumi
              do 20 n = 1, nev
                call cir2gn (pdg(i1), h1, pdg(i2), h2, x1, x2, random)
                i = i + 1
                write (*, '(I7,4(X,I4),2(X,F10.8))')
     $               i, pdg(i1), h1, pdg(i2), h2, x1, x2
 20           continue
 13         continue
 12       continue
 11     continue
 10   continue
      end
```

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
      implicit none
      integer p1, h1, p2, h2, n, nevent, ierror
      double precision x1, x2
      external random
      nevent = 20
      ierror = 1
      call cir2ld ('default_polavg.circe', '*', 500D0, ierror)
      if (ierror .lt. 0) stop
      write (*, '(A7,2(X,A4),2(X,A10))')
     $     '#', 'pdg1', 'pdg2', 'x1', 'x2'
      do 10 n = 1, nevent
        call cir2ch (p1, h1, p2, h2, random)
        call cir2gn (p1, h1, p2, h2, x1, x2, random)
        write (*, '(I7,2(X,I4),2(X,F10.8))') n, p1, p2, x1, x2
 10   continue
      end
```

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
      implicit none
      double precision x1, x2, lumipp, lumimm, cir2lm
      integer n, nevent, npp, nmm, ierror
      external random, cir2lm
      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(X,A10))') '#', 'x1', 'x2'
```

```
      do 10 n = 1, npp
        call cir2gn (22,  1, 22,  1, x1, x2, random)
        write (*, '(I7,2(X,F10.8))') n, x1, x2
10    continue
      do 20 n = 1, nmm
        call cir2gn (22, -1, 22, -1, x1, x2, random)
        write (*, '(I7,2(X,F10.8))') n, x1, x2
20    continue
      end
```

a second approach alternates between the two possibilities

```
      program demo5
      implicit none
      double precision x1, x2, u, lumipp, lumimm, cir2lm
      integer n, nevent, ierror
      external random, cir2lm
      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(X,A10))') '#', 'x1', 'x2'
      do 10 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(X,F10.8))') n, x1, x2
10    continue
      end
```

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

```
      program demo6
      implicit none
      integer p1, h1, p2, h2, n, nevent, ierror
      double precision x1, x2
      external random
      nevent = 20
      ierror = 1
```

```
      call cir2ld ('default.circe', '*', 500D0, ierror)
      if (ierror .lt. 0) stop
      write (*, '(A7,2(X,A10))') '#', 'x1', 'x2'
      n = 0
10    continue
        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(X,F10.8))') n, x1, x2
        end if
        if (n .lt. nevent) then
          goto 10
        end if
      end
```

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
      implicit none
      integer n, nevent, ierror
      double precision x1, x2, w, cir2dn
      nevent = 20
      ierror = 1
      call cir2ld ('default.circe', '*', 500D0, ierror)
      if (ierror .lt. 0) stop
      write (*, '(A7,3(X,A10))') '#', 'x1', 'x2', 'weight'
      do 10 n = 1, nevent
        call random (x1)
```

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

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

```
      program demo8
      implicit none
      integer n, nevent, ierror
      double precision x1, x2, w, cir2dn
      nevent = 20
      ierror = 1
      call cir2ld ('default_polavg.circe', '*', 500D0, ierror)
      if (ierror .lt. 0) stop
      write (*, '(A7,3(X,A10))') '#', 'x1', 'x2', 'weight'
      do 10 n = 1, nevent
        call random (x1)
        call random (x2)
        w = cir2dn (22, 0, 22, 0, x1, x2)
        write (*, '(I7,2(X,F10.8),X,E10.4)') n, x1, x2, w
10    continue
      end
```

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

```
      program demo9
      implicit none
      integer n, nevent, ierror
      double precision x1, x2, w, cir2dn, cir2lm
      double precision 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(X,A10))') '#', 'x1', 'x2', 'weight'
      do 10 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(X,F10.8),X,E10.4)') n, x1, x2, w
10    continue
      end
```

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.

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 GeV and we should bin each beam in steps of 500 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`

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

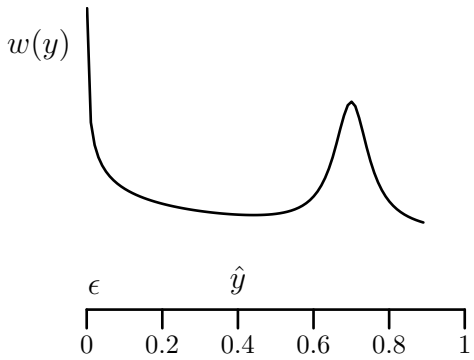

Figure 2: Distribution with both an integrable singularity $\propto x^{-0.2}$ and a peak at finite $x \approx 0.7$.

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.

- 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.

[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.

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.

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

Inverse

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

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

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

i. e.

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

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

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

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 14 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
```

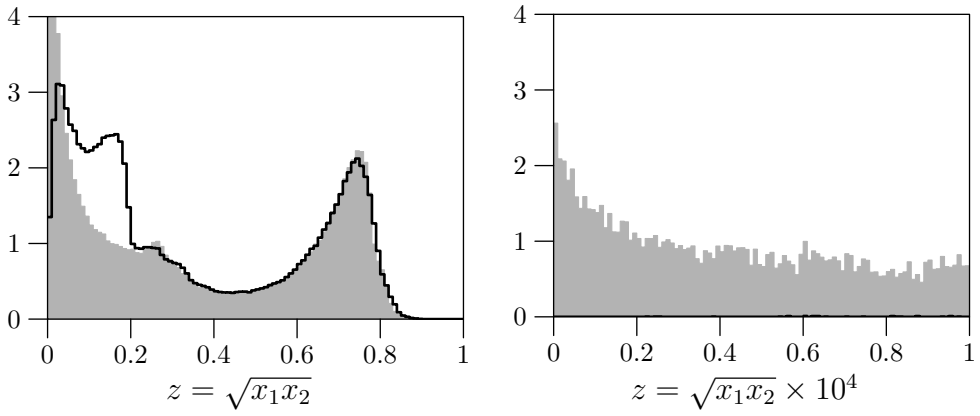

Figure 3: Using 20 equidistant bins in each direction. In the region blown up on the right neither 20 equidistant bins nor 50 equidistant bin produce enough events to be visible. In this and all following plots, the simulated input data is shown as a filled histogram, and the `circe2` parametrization is shown as a solid line.

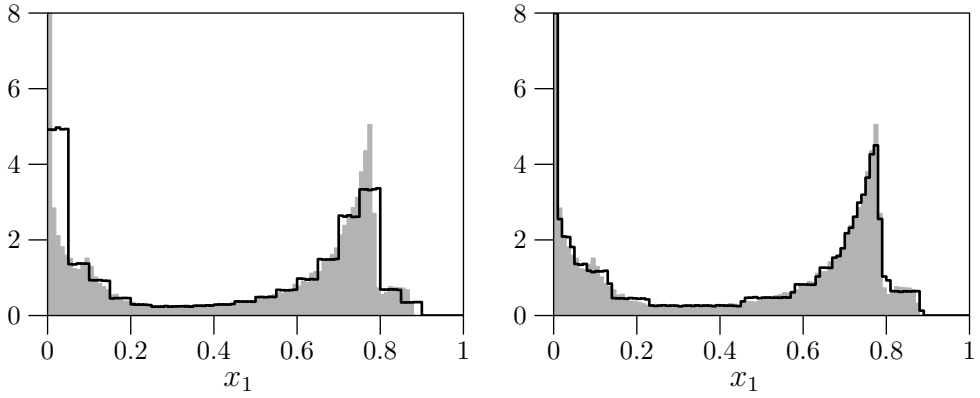

Figure 4: Using 20 bins, both equidistant (left) and adapted (right).

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

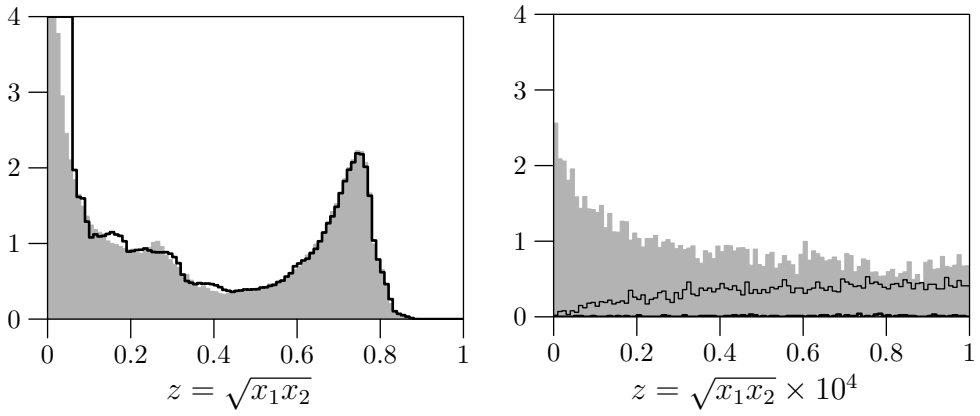

Figure 5: 20 adapted bins. In the blow-up, the 50 bin result is shown for illustration as a thin line, while the 20 bin result remains almost invisible.

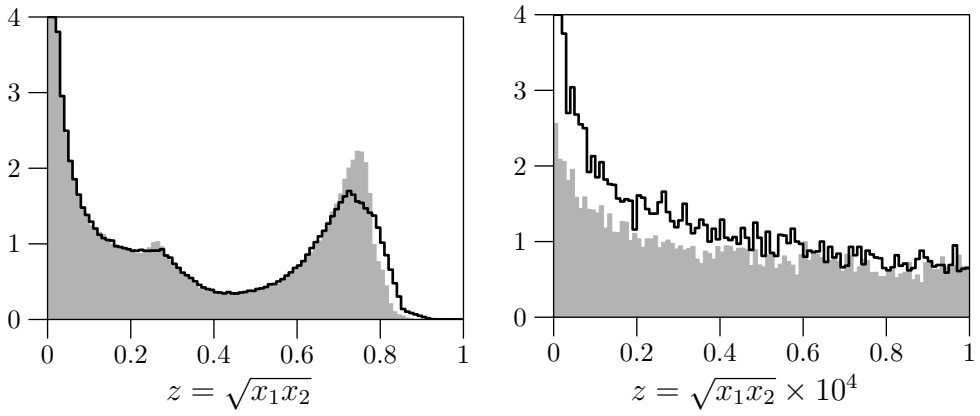

Figure 6: Using 20 equidistant bins in each direction with a power map appropriate for $x^{-0.67}$.

```
bins = 20
```

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

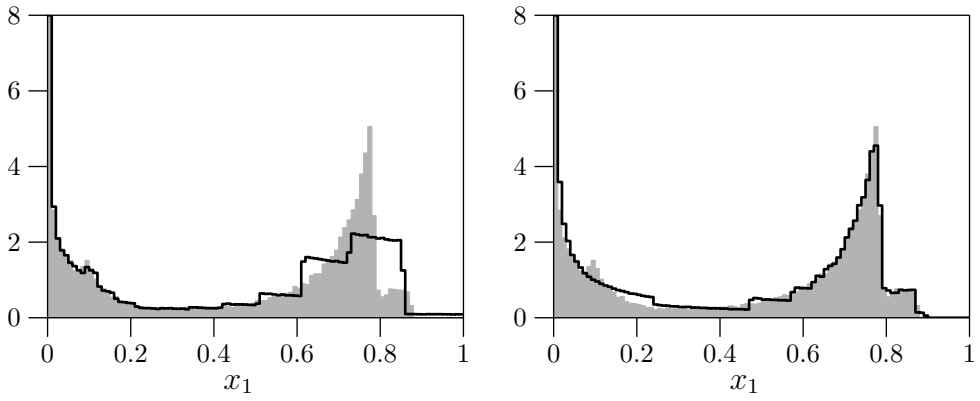

Figure 7: Using 20 bins with a power map appropriate for $x^{-0.67}$, equidistant (left) and adapted (right).

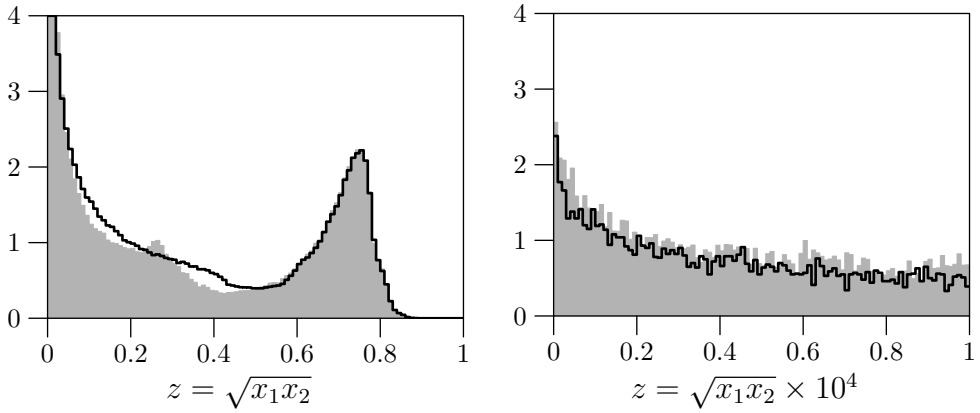

Figure 8: Using 20 adapted bins in each direction with a power map appropriate for $x^{-0.67}$.

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

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 }
```

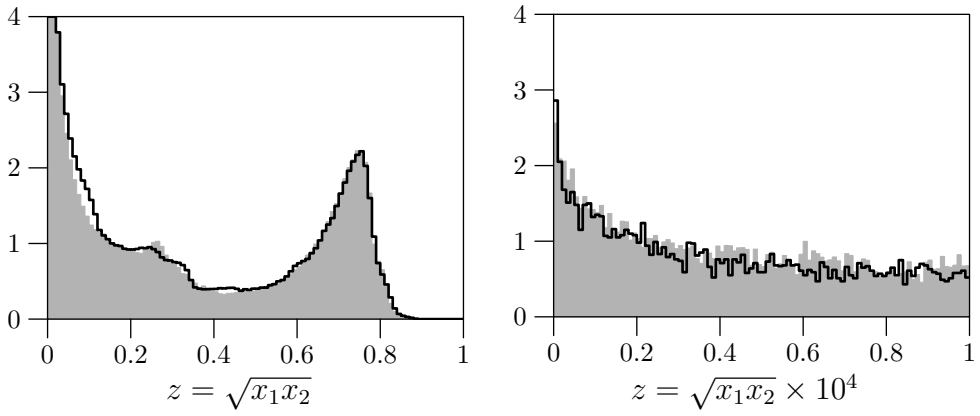

Figure 9: Using $4 + 16$ adapted bins in each direction with a power map appropriate for $x^{-0.67}$ in the 4 bins below $x < 0.05$.

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`		
$x_{1,0}$		
$x_{1,1}\ \ m_{1,1}\ \ \alpha_{1,1}\ \ \xi_{1,1}\ \ \eta_{1,1}\ \ a_{1,1}\ \ b_{1,1}$		
...		
$x_{1,n_1}\ m_{1,n_1}\ \alpha_{1,n_1}\ \xi_{1,n_1}\ \eta_{1,n_1}\ a_{1,n_1}\ b_{1,n_1}$		
`x2, map2, alpha2, xi2, eta2, a2, b2`		
$x_{2,0}$		
$x_{2,1}\ \ m_{2,1}\ \ \alpha_{2,1}\ \ \xi_{2,1}\ \ \eta_{2,1}\ \ a_{2,1}\ \ b_{2,1}$		
...		
$x_{2,n_2}\ m_{2,n_2}\ \alpha_{2,n_2}\ \xi_{2,n_2}\ \eta_{2,n_2}\ a_{2,n_2}\ b_{2,n_2}$		
`weights`		
$w_1\ \ [w_1\chi_1^{0,1}\ \ w_1\chi_1^{0,2}\ \ \ldots\ \ w_1\chi_1^{3,3}]$		optional $w\cdot\chi$
$w_2\ \ [w_1\chi_2^{0,1}\ \ w_1\chi_2^{0,2}\ \ \ldots\ \ w_1\chi_2^{3,3}]$		
...		
$w_{n_1n_2}\ \ [w_1\chi_{n_1n_2}^{0,1}\ \ w_1\chi_{n_1n_2}^{0,2}\ \ \ldots\ \ w_1\chi_{n_1n_2}^{3,3}]$	⎦	end repeat
`ECRIC2`		mandatory end line

Table 2: File format. The variable input lines (except comments) are designed to be readable by FORTRAN77 '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`

49a ⟨*Version* 49a⟩≡ (63d)
```
'Version 2.7.0'
```

49b ⟨`implicit none` 49b⟩≡
```
implicit none
```

49c ⟨`circe2.f90` 49c⟩≡
```
! circe2.f90 -- correlated beam spectra for linear colliders
⟨Copyleft notice 49e⟩
⟨Separator 49d⟩
module circe2
  use kinds
  implicit none
  private
  ⟨circe2 parameters 55d⟩
  ⟨circe2 declarations 50a⟩
contains
  ⟨circe2 implementation 54d⟩
end module circe2
```

49d ⟨*Separator* 49d⟩≡ (49c 54d 55b 60–63 72)
```
!------------------------------------------------------------------------
```

The following is usually not needed for scientific programs. Nobody is going to hijack such code. But let us include it anyway to spread the gospel of free software:

49e ⟨*Copyleft notice* 49e⟩≡ (49c 72)
```
! Copyright (C) 2001-2019 by Thorsten Ohl <ohl@physik.uni-wuerzburg.de>
!
! 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

50a ⟨circe2 *declarations* 50a⟩≡ (49c) 50b▷
```
type circe2_division
  ⟨circe2_division members 52b⟩
end type circe2_division
```

50b ⟨circe2 *declarations* 50a⟩+≡ (49c) ◁50a 50c▷
```
type circe2_channel
  ⟨circe2_channel members 50e⟩
end type circe2_channel
```

50c ⟨circe2 *declarations* 50a⟩+≡ (49c) ◁50b 54a▷
```
type circe2_state
  ⟨circe2_state members 50d⟩
end type circe2_state
public :: circe2_state
```

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.

50d ⟨circe2_state *members* 50d⟩≡ (50c) 53c▷
```
type(circe2_channel), dimension(:), allocatable :: ch
```

50e ⟨circe2_channel *members* 50e⟩≡ (50b) 50f▷
```
real(kind=default), dimension(:), allocatable :: wgt
```

50f ⟨circe2_channel *members* 50e⟩+≡ (50b) ◁50e 52a▷
```
type(circe2_division), dimension(2) :: d
```

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

$$i = i_1 + (i_2 - 1)n_1\,. \tag{51}$$

The inverse

$$i_1 = 1 + ((i-1) \mod n_1) \tag{52a}$$

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

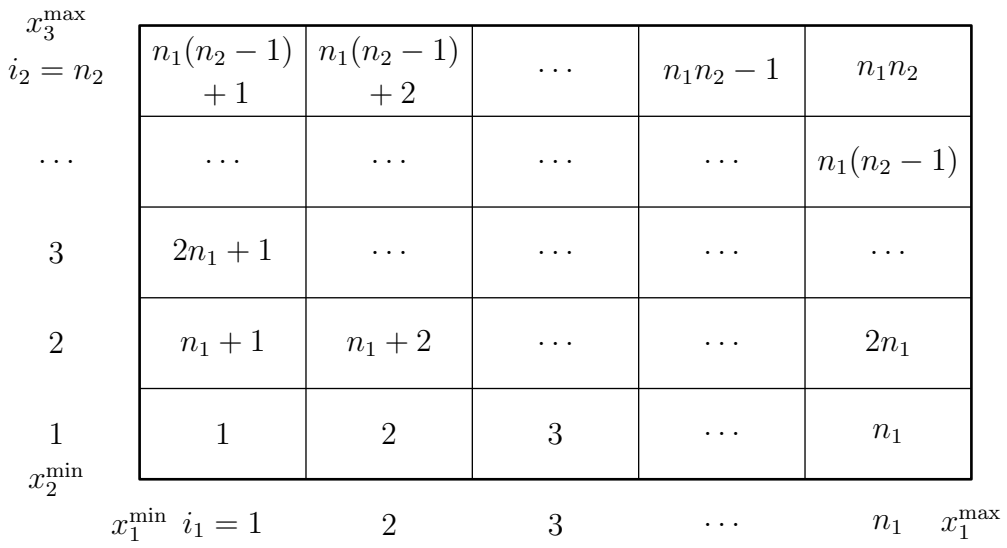

Figure 16: Enumerating the bins linearly, starting from 1 (Fortran style). Probability distribution functions will have a sentinel at 0 that's always 0.

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

can also be written

$$i_2 = 1 + \lfloor (i-1)/n_1 \rfloor \tag{53a}$$
$$i_1 = i - (i_2 - 1)n_1 \tag{53b}$$

because

$$\begin{aligned} 1 + \lfloor (i-1)/n_1 \rfloor &= 1 + \lfloor i_2 - 1 + (i_1 - 1)/n_1 \rfloor \\ &= 1 + \lfloor (i_1 + (i_2-1)n_1 - 1)/n_1 \rfloor = 1 + i_2 - 1 + \underbrace{\lfloor (i_1-1)/n_1 \rfloor}_{=0} = i_2 \end{aligned} \tag{54a}$$

and trivially

$$i - (i_2 - 1)n_1 = i_1 + (i_2 - 1)n_1 - (i_2 - 1)n_1 = i_1 \tag{54b}$$

51a ⟨(i1, i2) ← i 51a⟩≡ (67b)

```
i2 = 1 + (i - 1) / ubound (ch%d(1)%x, dim=1)
i1 = i - (i2 - 1) * ubound (ch%d(1)%x, dim=1)
```

51b ⟨ib ← i 51b⟩≡ (55b)

```
ib(2) = 1 + (i - 1) / ubound (ch%d(1)%x, dim=1)
ib(1) = i - (ib(2) - 1) * ubound (ch%d(1)%x, dim=1)
```

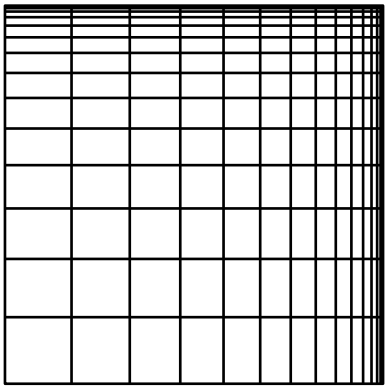

Figure 17: Almost factorizable distributions, like e^+e^-.

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

$$\frac{\mathrm{d}x}{\mathrm{d}y} \cdot \frac{1}{\Delta x} \approx \frac{1}{\Delta y} \tag{55}$$

We elect to use the former, because this reflects the distribution of the events generated by `circe2_generate` *inside* the bins as well. This quantity is more conveniently stored as a true two-dimensional array:

52a ⟨`circe2_channel` *members* 50e⟩+≡ (50b) ◁50f 52c▷

```
real(kind=default), dimension(:,:), allocatable :: val
```

52b ⟨`circe2_division` *members* 52b⟩≡ (50a) 53d▷

```
real(kind=default), dimension(:), allocatable :: x
```

52c ⟨`circe2_channel` *members* 50e⟩+≡ (50b) ◁52a 53a▷

```
logical :: triang
```

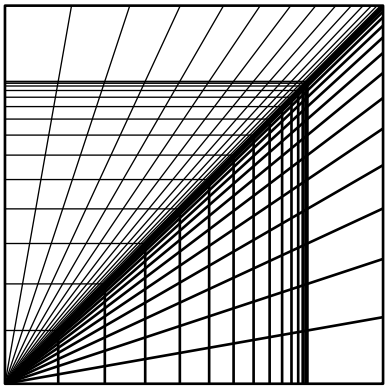

Figure 18: Symmetrical, strongly correlated distributions, e. g. with a ridge on the diagonal, like $\gamma\gamma$ at a γ-collider.

11.1 Channels

The number of available channels $\gamma\gamma$, $e^-\gamma$, e^-e^+, etc. can be found with `size (circe2_state%ch)`. The particles that are described by this channel and their polarizations:

53a ⟨`circe2_channel` *members* 50e⟩+≡ (50b) ◁52c 53b▷

```
integer, dimension(2) :: pid, pol
```

The integrated luminosity of the channel

53b ⟨`circe2_channel` *members* 50e⟩+≡ (50b) ◁53a

```
real(kind=default) :: lumi
```

The integrated luminosity of the channel

53c ⟨`circe2_state` *members* 50d⟩+≡ (50c) ◁50d 55c▷

```
real(kind=default), dimension(:), allocatable :: cwgt
```

11.2 Maps

53d ⟨`circe2_division` *members* 52b⟩+≡ (50a) ◁52b 53e▷

```
integer, dimension(:), allocatable :: map
```

53e ⟨`circe2_division` *members* 52b⟩+≡ (50a) ◁53d 53f▷

```
real(kind=default), dimension(:), allocatable :: y
```

53f ⟨`circe2_division` *members* 52b⟩+≡ (50a) ◁53e

```
real(kind=default), dimension(:), allocatable :: alpha, xi, eta, a, b
```

12 Random Number Generation

We use the new WHIZARD interface.

54a ⟨circe2 *declarations* 50a⟩+≡ (49c) ◁50c 54b▷

```
public :: rng_type
type, abstract :: rng_type
  contains
    procedure(rng_generate), deferred :: generate
end type rng_type
```

54b ⟨circe2 *declarations* 50a⟩+≡ (49c) ◁54a 54c▷

```
abstract interface
   subroutine rng_generate (rng_obj, u)
     import :: rng_type, default
     class(rng_type), intent(inout) :: rng_obj
     real(kind=default), 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.

54c ⟨circe2 *declarations* 50a⟩+≡ (49c) ◁54b 55a▷

```
public :: circe2_generate
interface circe2_generate
   module procedure circe2_generate_ph
end interface circe2_generate
```

54d ⟨circe2 *implementation* 54d⟩≡ (49c) 55b▷

```
subroutine circe2_generate_ph (c2s, rng, y, p, h)
  type(circe2_state), intent(in) :: c2s
  class(rng_type), intent(inout) :: rng
  real(kind=default), dimension(:), intent(out) :: y
  integer, dimension(:), intent(in) :: p
  integer, dimension(:), intent(in) :: h
  integer :: i, ic
  ⟨Find ic for p and h 55e⟩
  ⟨Complain and return iff ic ≤ 0 56a⟩
  call circe2_generate_channel (c2s%ch(ic), rng, y)
end subroutine circe2_generate_ph
⟨Separator 49d⟩
```

55a ⟨circe2 *declarations* 50a⟩+≡ (49c) ◁54c 58b▷

```
interface circe2_generate
   module procedure circe2_generate_channel
end interface circe2_generate
```

55b ⟨circe2 *implementation* 54d⟩+≡ (49c) ◁54d 57a▷

```
subroutine circe2_generate_channel (ch, rng, y)
  type(circe2_channel), intent(in) :: ch
  class(rng_type), intent(inout) :: rng
  real(kind=default), dimension(:), intent(out) :: y
  integer :: i, d, ibot, itop
  integer, dimension(2) :: ib
  real(kind=default), dimension(2) :: x, v
  real(kind=default) :: u, tmp
  call rng%generate (u)
  ⟨Do a binary search for wgt(i − 1) ≤ u < wgt(i) 56b⟩
  ⟨ib ← i 51b⟩
  ⟨x ∈ [x(ib − 1), x(ib)] 56c⟩
  y = circe2_map (ch%d, x, ib)
  ⟨Inverse triangle map 57c⟩
end subroutine circe2_generate_channel
⟨Separator 49d⟩
```

55c ⟨circe2_state *members* 50d⟩+≡ (50c) ◁53c

```
integer :: polspt
```

55d ⟨circe2 *parameters* 55d⟩≡ (49c)

```
integer, parameter :: POLAVG = 1, POLHEL = 2, 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.

55e ⟨*Find* ic *for* p *and* h 55e⟩≡ (54d 60d)

```
ic = 0
if ((c2s%polspt == POLAVG .or. c2s%polspt == POLGEN) .and. any (h /= 0)) then
   write (*, '(2A)') 'circe2: current beam description ', &
        'supports only polarization averages'
else if (c2s%polspt == POLHEL .and. any (h == 0)) then
   write (*, '(2A)') 'circe2: polarization averages ', &
        'not supported by current beam description'
else
   do i = 1, size (c2s%ch)
      if (all (p == c2s%ch(i)%pid .and. h == c2s%ch(i)%pol)) then
         ic = i
      end if
```

```
   end do
end if
```

56a ⟨*Complain and* `return` *iff* `ic` ≤ 0 56a⟩≡ (54d)

```
if (ic <= 0) then
   write (*, '(A,2I4,A,2I3)') &
        'circe2: no channel for particles', p, &
        ' and polarizations', h
   y = - huge (y)
   return
end if
```

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

56b ⟨*Do a binary search for* `wgt(i - 1)` $\leq$ `u` $<$ `wgt(i)` 56b⟩≡ (55b)

```
ibot = 0
itop = ubound (ch%wgt, dim=1)
do
   if (itop <= ibot + 1) then
      i = ibot + 1
      exit
   else
      i = (ibot + itop) / 2
      if (u < ch%wgt(i)) then
         itop = i
      else
         ibot = i
      end if
   end if
end do
```

56c ⟨`x` $\in$ [`x(ib - 1)`, `x(ib)`] 56c⟩≡ (55b)

```
call rng%generate (v(1))
call rng%generate (v(2))
do d = 1, 2
  x(d) = ch%d(d)%x(ib(d))*v(d) + ch%d(d)%x(ib(d)-1)*(1-v(d))
end do
```

The NAG compiler is picky and doesn't like $(-0)^\alpha$ at all.

56d ⟨$y \leftarrow (a(x-\xi))^\alpha/b + \eta$ 56d⟩≡ (57a)

```
z = d%a(b) * (x - d%xi(b))
if (abs (z) <= tiny (z)) then
   z = abs (z)
end if
y = z**d%alpha(b) / d%b(b) + d%eta(b)
```

57a ⟨circe2 *implementation* 54d⟩+≡ (49c) ◁55b 57b▷

```
elemental function circe2_map (d, x, b) result (y)
   type(circe2_division), intent(in) :: d
   real(kind=default), intent(in) :: x
   integer, intent(in) :: b
   real(kind=default) :: y
   real(kind=default) :: z
   select case (d%map(b))
   case (0)
      y = x
   case (1)
      ⟨y ← (a(x − ξ))^α/b + η 56d⟩
   case (2)
      y = d%a(b) * tan (d%a(b)*(x-d%xi(b)) / d%b(b)**2) + d%eta(b)
   case default
      y = - huge (y)
   end select
end function circe2_map
```

cf. (55)

57b ⟨circe2 *implementation* 54d⟩+≡ (49c) ◁57a 58c▷

```
elemental function circe2_jacobian (d, y, b) result (j)
   type(circe2_division), intent(in) :: d
   real(kind=default), intent(in) :: y
   integer, intent(in) :: b
   real(kind=default) :: j
   select case (d%map(b))
   case (0)
      j = 1
   case (1)
      j = d%b(b) / (d%a(b)*d%alpha(b)) &
        * (d%b(b)*(y-d%eta(b)))**(1/d%alpha(b)-1)
   case (2)
      j = d%b(b)**2 / ((y-d%eta(b))**2 + d%a(b)**2)
   case default
      j = - huge (j)
   end select
end function circe2_jacobian
```

There's still something wrong with *unweighted* events for the case that there is a triangle map *together* with a non-trivial x(2) → y(2) map. *Fix this!!!*

57c ⟨*Inverse triangle map* 57c⟩≡ (55b)

```
if (ch%triang) then
   y(2) = y(1) * y(2)
   ⟨Swap y(1) and y(2) in 50% of the cases 58a⟩
end if
```

58a ⟨*Swap* `y(1)` *and* `y(2)` *in 50% of the cases* 58a⟩≡ (57c)

```
call rng%generate (u)
if (2*u >= 1) then
   tmp = y(1)
   y(1) = y(2)
   y(2) = tmp
end if
```

14 Channel selection

We could call `circe2_generate` immediately, but then `circe2_generate` and `cir2_choose_channel` would have the same calling conventions and might have caused a lot of confusion.

58b ⟨`circe2` *declarations* 50a⟩+≡ (49c) ◁55a 59a▷

```
public :: circe2_choose_channel
interface circe2_choose_channel
   module procedure circe2_choose_channel
end interface circe2_choose_channel
```

58c ⟨`circe2` *implementation* 54d⟩+≡ (49c) ◁57b 59b▷

```
subroutine circe2_choose_channel (c2s, rng, p, h)
  type(circe2_state), intent(in) :: c2s
  class(rng_type), intent(inout) :: rng
  integer, dimension(:), intent(out) :: p, h
  integer :: ic, ibot, itop
  real(kind=default) :: u
  call rng%generate (u)
  ibot = 0
  itop = size (c2s%ch)
  do
     if (itop <= ibot + 1) then
        ic = ibot + 1
        p = c2s%ch(ic)%pid
        h = c2s%ch(ic)%pol
        return
     else
        ic = (ibot + itop) / 2
        if (u < c2s%cwgt(ic)) then
```

```
            itop = ic
         else
            ibot = ic
         end if
      end if
    end do
    write (*, '(A)') 'circe2: internal error'
    stop
  end subroutine circe2_choose_channel
```

Below, we will always have `h` = 0. but we don't have to check this explicitely, because `circe2_density_matrix` will do it anyway. The procedure could be made more efficient, since most of `circe2_generate` is undoing parts of `circe2_generate`.

59a ⟨`circe2` *declarations* 50a⟩+≡ (49c) ◁58b 60a▷

```
  public :: circe2_generate_polarized
  interface circe2_generate_polarized
     module procedure circe2_generate_polarized
  end interface circe2_generate_polarized
```

59b ⟨`circe2` *implementation* 54d⟩+≡ (49c) ◁58c 60b▷

```
  subroutine circe2_generate_polarized (c2s, rng, p, pol, x)
    type(circe2_state), intent(in) :: c2s
    class(rng_type), intent(inout) :: rng
    integer, dimension(:), intent(out) :: p
    real(kind=default), intent(out) :: pol(0:3,0:3)
    real(kind=default), dimension(:), intent(out) :: x
    integer, dimension(2) :: h
    integer :: i1, i2
    real(kind=default) :: pol00
    call circe2_choose_channel (c2s, rng, p, h)
    call circe2_generate (c2s, rng, x, p, h)
    call circe2_density_matrix (c2s, pol, p, x)
    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 circe2_generate_polarized
```

15 Luminosity

60a ⟨circe2 *declarations* 50a⟩+≡ (49c) ◁59a 60c▷

```
public :: circe2_luminosity
```

60b ⟨circe2 *implementation* 54d⟩+≡ (49c) ◁59b 60d▷

```
function circe2_luminosity (c2s, p, h)
  type(circe2_state), intent(in) :: c2s
  integer, dimension(:), intent(in) :: p
  integer, dimension(:), intent(in) :: h
  real(kind=default) :: circe2_luminosity
  integer :: ic
  circe2_luminosity = 0
  do ic = 1, size (c2s%ch)
     if (       all (p == c2s%ch(ic)%pid .or. p == 0) &
          .and. all (h == c2s%ch(ic)%pol .or. h == 0)) then
        circe2_luminosity = circe2_luminosity + c2s%ch(ic)%lumi
     end if
  end do
end function circe2_luminosity
⟨Separator 49d⟩
```

16 2D-Distribution

60c ⟨circe2 *declarations* 50a⟩+≡ (49c) ◁60a 61a▷

```
public :: circe2_distribution
interface circe2_distribution
   module procedure circe2_distribution_ph
end interface circe2_distribution
```

60d ⟨circe2 *implementation* 54d⟩+≡ (49c) ◁60b 61b▷

```
function circe2_distribution_ph (c2s, p, h, yy)
  type(circe2_state), intent(in) :: c2s
  integer, dimension(:), intent(in) :: p
  real(kind=default), dimension(:), intent(in)  :: yy
  integer, dimension(:), intent(in) :: h
  real(kind=default) :: circe2_distribution_ph
  integer :: i, ic
  ⟨Find ic for p and h 55e⟩
  if (ic <= 0) then
     circe2_distribution_ph = 0
  else
     circe2_distribution_ph = circe2_distribution_channel (c2s%ch(ic), yy)
```

```
  end if
end function circe2_distribution_ph
```
⟨*Separator* 49d⟩

61a ⟨circe2 *declarations* 50a⟩+≡ (49c) ◁60c 62c▷

```
interface circe2_distribution
   module procedure circe2_distribution_channel
end interface circe2_distribution
```

61b ⟨circe2 *implementation* 54d⟩+≡ (49c) ◁60d 62d▷

```
function circe2_distribution_channel (ch, yy)
  type(circe2_channel), intent(in) :: ch
  real(kind=default), dimension(:), intent(in)  :: yy
  real(kind=default) :: circe2_distribution_channel
  real(kind=default), dimension(2) :: y
  integer :: d, ibot, itop
  integer, dimension(2) :: ib
  ⟨y⟩ ← yy 61c⟩
  if (      y(1) < ch%d(1)%y(0) &
       .or. y(1) > ch%d(1)%y(ubound (ch%d(1)%y, dim=1)) &
       .or. y(2) < ch%d(2)%y(0) &
       .or. y(2) > ch%d(2)%y(ubound (ch%d(2)%y, dim=1))) then
     circe2_distribution_channel = 0
     return
  end if
  ⟨Do a binary search for y(ib − 1) ≤ y < y(ib) 62b⟩
  circe2_distribution_channel = &
      ch%val(ib(1),ib(2)) * product (circe2_jacobian (ch%d, y, ib))
  ⟨Apply Jacobian for triangle map 62a⟩
end function circe2_distribution_channel
```
⟨*Separator* 49d⟩

The triangle map

$$\begin{aligned}\tau : \{(x_1, x_2) \in [0,1] \times [0,1] : x_2 \leq x_1\} &\to [0,1] \times [0,1] \\ (x_1, x_2) &\mapsto (y_1, y_2) = (x_1, x_1 x_2)\end{aligned} \tag{56}$$

and its inverse

$$\begin{aligned}\tau^{-1} : [0,1] \times [0,1] &\to \{(x_1, x_2) \in [0,1] \times [0,1] : x_2 \leq x_1\} \\ (y_1, y_2) &\mapsto (x_1, x_2) = (y_1, y_2/y_1)\end{aligned} \tag{57}$$

61c ⟨y⟩ ← yy 61c⟩≡ (61b)

```
if (ch%triang) then
   y(1) = maxval (yy)
   y(2) = minval (yy) / y(1)
```

```
else
   y = yy
end if
```

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

$$\mathrm{d}x_1 \wedge \mathrm{d}x_2 = \frac{1}{y_2} \cdot \mathrm{d}y_1 \wedge \mathrm{d}y_2 \tag{58}$$

62a ⟨*Apply Jacobian for triangle map* 62a⟩≡ (61b)

```
if (ch%triang) then
   circe2_distribution_channel = circe2_distribution_channel / y(1)
end if
```

Careful: the loop over `d` *must* be executed sequentially, because of the shared local variables `ibot` and `itop`.

62b ⟨*Do a binary search for* $\mathtt{y(ib-1)} \le \mathtt{y} < \mathtt{y(ib)}$ 62b⟩≡ (61b)

```
do d = 1, 2
   ibot = 0
   itop = ubound (ch%d(d)%x, dim=1)
   search: do
      if (itop <= ibot + 1) then
         ib(d) = ibot + 1
         exit search
      else
         ib(d) = (ibot + itop) / 2
         if (y(d) < ch%d(d)%y(ib(d))) then
            itop = ib(d)
         else
            ibot = ib(d)
         end if
      end if
   end do search
end do
```

62c ⟨`circe2` *declarations* 50a⟩+≡ (49c) ◁61a 63b▷

```
public :: circe2_density_matrix
```

62d ⟨`circe2` *implementation* 54d⟩+≡ (49c) ◁61b 63d▷

```
subroutine circe2_density_matrix (c2s, pol, p, x)
  type(circe2_state), intent(in) :: c2s
  real(kind=default), dimension(0:,0:), intent(out) :: pol
  integer, dimension(:), intent(in) :: p
  real(kind=default), dimension(:), intent(in) :: x
  ⟨Test support for density matrices 63a⟩
  print *, 'circe2: circe2_density_matrix not implemented yet!'
```

```
  if (p(1) < p(2) .and. x(1) < x(2)) then
     ! nonsense test to suppress warning
  end if
  pol = 0
end subroutine circe2_density_matrix
```
⟨*Separator* 49d⟩

63a ⟨*Test support for density matrices* 63a⟩≡ (62d)
```
if (c2s%polspt /= POLGEN) then
   write (*, '(2A)') 'circe2: current beam ', &
        'description supports no density matrices'
   return
end if
```

17 Reading Files

63b ⟨circe2 *declarations* 50a⟩+≡ (49c) ◁62c
```
public :: circe2_load
```
⟨*Error codes for* circe2_load 63c⟩

63c ⟨*Error codes for* circe2_load 63c⟩≡ (63b)
```
integer, parameter, public :: &
     EOK = 0, EFILE = -1, EMATCH = -2, EFORMT = -3, ESIZE = -4
```

63d ⟨circe2 *implementation* 54d⟩+≡ (49c) ◁62d 65a▷
```
subroutine circe2_load (c2s, file, design, roots, ierror)
  type(circe2_state), intent(out) :: c2s
  character(len=*), intent(in) :: file, design
  real(kind=default), intent(in) :: roots
  integer, intent(out) :: ierror
  character(len=72) :: buffer, fdesgn, fpolsp
  real(kind=default) :: froots
  integer :: lun, loaded, prefix
  logical match
```
⟨*Local variables in* circe2_load 65c⟩
⟨*Find free logical unit for* lun 68c⟩
```
  if (lun < 0) then
     write (*, '(A)') 'circe2_load: no free unit'
     ierror = ESIZE
     return
  end if
  loaded = 0
```
⟨*Open* name *for reading on* lun 67d⟩
```
  if (ierror .gt. 0) then
```

```
      write (*, '(2A)') 'circe2_load: ', ⟨Version 49a⟩
    end if
    prefix = index (design, '*') - 1
    do
      ⟨Skip comments until CIRCE2 67f⟩
      if (buffer(8:15) == 'FORMAT#1') then
        read (lun, *)
        read (lun, *) fdesgn, froots
        ⟨Check if design and fdesgn do match 64a⟩
        if (match .and. abs (froots - roots) <= 1) then
          ⟨Load histograms 64b⟩
          loaded = loaded + 1
        else
          ⟨Skip data until ECRIC2 68a⟩
          cycle
        end if
      else
        write (*, '(2A)') 'circe2_load: invalid format: ', buffer(8:72)
        ierror = EFORMT
        return
      end if
      ⟨Check for ECRIC2 68b⟩
    end do
  end subroutine circe2_load
  ⟨Separator 49d⟩
```

64a ⟨*Check if* `design` *and* `fdesgn` *do* `match` 64a⟩≡ (63d)

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

64b ⟨*Load histograms* 64b⟩≡ (63d)

```
  read (lun, *)
  read (lun, *) nc, fpolsp
  allocate (c2s%ch(nc), c2s%cwgt(0:nc))
  ⟨Decode polarization support 65b⟩
  c2s%cwgt(0) = 0
```

```
do ic = 1, nc
   call circe2_load_channel (c2s%ch(ic), c2s%polspt, lun, ierror)
   c2s%cwgt(ic) = c2s%cwgt(ic-1) + c2s%ch(ic)%lumi
end do
c2s%cwgt = c2s%cwgt / c2s%cwgt(nc)
```

65a ⟨circe2 *implementation* 54d⟩+≡ (49c) ◁63d

```
subroutine circe2_load_channel (ch, polspt, lun, ierror)
  type(circe2_channel), intent(out) :: ch
  integer, intent(in) :: polspt, lun
  integer, intent(out) :: ierror
  integer :: d, i, ib
  integer :: i1, i2
  integer, dimension(2) :: nb
  real(kind=default) :: w
  ⟨Load channel ch 65d⟩
  ⟨Load divisions x 66b⟩
  ⟨Calculate y 67a⟩
  ⟨Load weights wgt and val 67b⟩
end subroutine circe2_load_channel
```

65b ⟨*Decode polarization support* 65b⟩≡ (64b)

```
if (fpolsp(1:1)=='a' .or. fpolsp(1:1)=='A') then
   c2s%polspt = POLAVG
else if (fpolsp(1:1)=='h' .or. fpolsp(1:1)=='H') then
   c2s%polspt = POLHEL
else if (fpolsp(1:1)=='d' .or. fpolsp(1:1)=='D') then
   c2s%polspt = POLGEN
else
   write (*, '(A,I5)') 'circe2_load: invalid polarization support: ', fpolsp
   ierror = EFORMT
   return
end if
```

65c ⟨*Local variables in* circe2_load 65c⟩≡ (63d) 67c▷

```
integer :: ic, nc
```

65d ⟨*Load channel* ch 65d⟩≡ (65a) 66a▷

```
read (lun, *)
read (lun, *) ch%pid(1), ch%pol(1), ch%pid(2), ch%pol(2), ch%lumi
⟨Check polarization support 65e⟩
```

65e ⟨*Check polarization support* 65e⟩≡ (65d)

```
if (polspt == POLAVG .and. any (ch%pol /= 0)) then
   write (*, '(A)') 'circe2_load: expecting averaged polarization'
   ierror = EFORMT
```

```
      return
   else if (polspt == POLHEL .and. any (ch%pol == 0)) then
      write (*, '(A)') 'circe2_load: expecting helicities'
      ierror = EFORMT
      return
   else if (polspt == POLGEN) then
      write (*, '(A)') 'circe2_load: general polarizations not supported yet'
      ierror = EFORMT
      return
   else if (polspt == POLGEN .and. any (ch%pol /= 0)) then
      write (*, '(A)') 'circe2_load: expecting pol = 0'
      ierror = EFORMT
      return
   end if
```

66a ⟨*Load channel* ch 65d⟩+≡ (65a) ◁65d

```
   read (lun, *)
   read (lun, *) nb, ch%triang
```

66b ⟨*Load divisions* x 66b⟩≡ (65a)

```
   do d = 1, 2
      read (lun, *)
      allocate (ch%d(d)%x(0:nb(d)), ch%d(d)%y(0:nb(d)))
      allocate (ch%d(d)%map(nb(d)), ch%d(d)%alpha(nb(d)))
      allocate (ch%d(d)%xi(nb(d)), ch%d(d)%eta(nb(d)))
      allocate (ch%d(d)%a(nb(d)), ch%d(d)%b(nb(d)))
      read (lun, *) ch%d(d)%x(0)
      do ib = 1, nb(d)
         read (lun, *) ch%d(d)%x(ib), ch%d(d)%map(ib), &
              ch%d(d)%alpha(ib), ch%d(d)%xi(ib), ch%d(d)%eta(ib), &
              ch%d(d)%a(ib), ch%d(d)%b(ib)
         if (ch%d(d)%map(ib) < 0 .or. ch%d(d)%map(ib) > 2) then
            write (*, '(A,I3)') 'circe2_load: invalid map: ', ch%d(d)%map(ib)
            ierror = EFORMT
            return
         end if
      end do
   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 `alpha(1,0,:)`, `xi(1,0,:)`, etc.

67a ⟨*Calculate* y 67a⟩≡ (65a)

```
do d = 1, 2
   do i = 0, ubound (ch%d(d)%x, dim=1)
      ch%d(d)%y(i) = circe2_map (ch%d(d), ch%d(d)%x(i), max (i, 1))
   end do
end do
```

cf. (55)

67b ⟨*Load weights* wgt *and* val 67b⟩≡ (65a)

```
read (lun, *)
allocate (ch%wgt(0:product(nb)), ch%val(nb(1),nb(2)))
ch%wgt(0) = 0
do i = 1, ubound (ch%wgt, dim=1)
   read (lun, *) w
   ch%wgt(i) = ch%wgt(i-1) + w
   ⟨(i1, i2) ← i 51a⟩
   ch%val(i1,i2) = w &
        / (  (ch%d(1)%x(i1) - ch%d(1)%x(i1-1)) &
           * (ch%d(2)%x(i2) - ch%d(2)%x(i2-1)))
end do
ch%wgt(ubound (ch%wgt, dim=1)) = 1
```

67c ⟨*Local variables in* circe2_load 65c⟩+≡ (63d) ◁65c 67e▷

17.1 Auxiliary Code For Reading Files

67d ⟨*Open* name *for reading on* lun 67d⟩≡ (63d)

```
open (unit = lun, file = file, status = 'old', iostat = status)
if (status /= 0) then
   write (*, '(2A)') 'circe2_load: can''t open ', file
   ierror = EFILE
   return
end if
```

67e ⟨*Local variables in* circe2_load 65c⟩+≡ (63d) ◁67c 69a▷

```
integer :: status
```

The outer do loop is never repeated!

67f ⟨*Skip comments until* CIRCE2 67f⟩≡ (63d)

```
find_circe2: do
   skip_comments: do
      read (lun, '(A)', iostat = status) buffer
```

```
            if (status /= 0) then
               close (unit = lun)
               if (loaded > 0) then
                  ierror = EOK
               else
                  ierror = EMATCH
               end if
               return
            else
               if (buffer(1:6) == 'CIRCE2') then
                  exit find_circe2
               else if (buffer(1:1) == '!') then
                  if (ierror > 0) then
                     write (*, '(A)') buffer
                  end if
               else
                  exit skip_comments
               end if
             end if
          end do skip_comments
          write (*, '(A)') 'circe2_load: invalid file'
          ierror = EFORMT
          return
        end do find_circe2
```

68a ⟨*Skip data until* ECRIC2 68a⟩≡ (63d)

```
skip_data: do
   read (lun, *) buffer
   if (buffer(1:6) == 'ECRIC2') then
      exit skip_data
   end if
end do skip_data
```

68b ⟨*Check for* ECRIC2 68b⟩≡ (63d)

```
read (lun, '(A)') buffer
if (buffer(1:6) /= 'ECRIC2') then
   write (*, '(A)') 'circe2_load: invalid file'
   ierror = EFORMT
   return
end if
```

68c ⟨*Find free logical unit for* lun 68c⟩≡ (63d 72)

```
scan: do lun = 10, 99
   inquire (unit = lun, exist = exists, opened = isopen, iostat = status)
   if (status == 0 .and. exists .and. .not.isopen) exit scan
```

```
  end do scan
  if (lun > 99) lun = -1
```

69a ⟨*Local variables in* `circe2_load` 65c⟩+≡ (63d) ◁67e

```
  logical exists, isopen
```

A Tests and Examples

A.1 Object-Oriented interface to `tao_random_numbers`

We need the object oriented interface to `tao_random_numbers` to be able to talk to the WHIZARD

69b ⟨`tao_random_objects.f90` 69b⟩≡

```
module tao_random_objects
  use kinds
  use tao_random_numbers
  use circe2
  implicit none
  private
  ⟨tao_random_objects declarations 69c⟩
contains
  ⟨tao_random_objects implementation 69d⟩
end module tao_random_objects
```

69c ⟨`tao_random_objects` *declarations* 69c⟩≡ (69b)

```
public :: rng_tao
type, extends (rng_type) :: rng_tao
   integer :: seed = 0
   integer :: n_calls = 0
   type(tao_random_state) :: state
 contains
   procedure :: generate => rng_tao_generate
   procedure :: init => rng_tao_init
end type rng_tao
```

69d ⟨`tao_random_objects` *implementation* 69d⟩≡ (69b) 70a▷

```
subroutine rng_tao_generate (rng_obj, u)
  class(rng_tao), intent(inout) :: rng_obj
  real(default), intent(out) :: u
  call tao_random_number (rng_obj%state, u)
  rng_obj%n_calls = rng_obj%n_calls + 1
end subroutine rng_tao_generate
```

70a ⟨tao_random_objects *implementation* 69d⟩+≡ (69b) ◁69d

```
subroutine rng_tao_init (rng_obj, seed)
  class(rng_tao), intent(inout) :: rng_obj
  integer, intent(in) :: seed
  rng_obj%seed = seed
  call tao_random_create (rng_obj%state, seed)
end subroutine rng_tao_init
```

A.2 circe2_generate: Standalone Generation of Samples

70b ⟨circe2_generate.f90 70b⟩≡

```
program circe2_generate_program
  use kinds
  use circe2
  use tao_random_objects
  implicit none
  type(circe2_state) :: c2s
  type(rng_tao), save :: rng
  character(len=1024) :: filename, design, buffer
  integer :: status, nevents, seed
  real(kind=default) :: roots
  real(kind=default), dimension(2) :: x
  integer :: i, ierror
  ⟨Process command line arguments for circe2_generate_program 70c⟩
  call circe2_load (c2s, trim(filename), trim(design), roots, ierror)
  if (ierror /= 0) then
     print *, "circe2_generate: failed to load design ", trim(design), &
          " for ", real (roots, kind=single), &
          " GeV from  ", trim(filename)
     stop
  end if
  do i = 1, nevents
     call circe2_generate (c2s, rng, x, [11, -11], [0, 0])
     write (*, '(F12.10,1X,F12.10)') x
  end do
  contains
     ⟨Module procedures for circe2_generate_program 71e⟩
end program circe2_generate_program
```

70c ⟨Process command line arguments for circe2_generate_program 70c⟩≡ (70b) 71a▷

```
call get_command_argument (1, value = filename, status = status)
if (status /= 0) filename = ""
```

71a ⟨*Process command line arguments for* `circe2_generate_program` 70c⟩+≡ (70b) ◁70c 71b▷

```
call get_command_argument (2, value = design, status = status)
if (status /= 0) design = ""
if (filename == "" .or. design == "") then
   print *, "usage: circe2_generate filename design [roots] [#events] [seed]"
   stop
end if
```

71b ⟨*Process command line arguments for* `circe2_generate_program` 70c⟩+≡ (70b) ◁71a 71c▷

```
call get_command_argument (3, value = buffer, status = status)
if (status == 0) then
   read (buffer, *, iostat = status) roots
   if (status /= 0) roots = 500
else
   roots = 500
end if
```

71c ⟨*Process command line arguments for* `circe2_generate_program` 70c⟩+≡ (70b) ◁71b 71d▷

```
call get_command_argument (4, value = buffer, status = status)
if (status == 0) then
   read (buffer, *, iostat = status) nevents
   if (status /= 0) nevents = 1000
else
   nevents = 1000
end if
```

71d ⟨*Process command line arguments for* `circe2_generate_program` 70c⟩+≡ (70b) ◁71c

```
call get_command_argument (5, value = buffer, status = status)
if (status == 0) then
   read (buffer, *, iostat = status) seed
   if (status == 0) then
      call random2_seed (rng, seed)
   else
      call random2_seed (rng)
   end if
else
   call random2_seed (rng)
end if
```

71e ⟨*Module procedures for* `circe2_generate_program` 71e⟩≡ (70b)

```
subroutine random2_seed (rng, seed)
  class(rng_tao), intent(inout) :: rng
  integer, intent(in), optional:: seed
  integer, dimension(8) :: date_time
  integer :: seed_value
  if (present (seed)) then
```

```
      seed_value = seed
    else
      call date_and_time (values = date_time)
      seed_value = product (date_time)
    endif
    call rng%init (seed_value)
  end subroutine random2_seed
```

A.3 `circe2_ls`: 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).

72 ⟨`circe2_ls.f90` 72⟩≡

```
! circe2_ls.f90 -- beam spectra for linear colliders and photon colliders
⟨Copyleft notice 49e⟩
⟨Separator 49d⟩
program circe2_ls
   use circe2
   use kinds
   implicit none
   integer :: i, lun
   character(len=132) :: buffer
   character(len=60) :: design, polspt
   integer :: pid1, hel1, pid2, hel2, nc
   real(kind=default) :: roots, lumi
   integer :: status
   logical :: exists, isopen
   character(len=1024) :: filename
   ⟨Find free logical unit for lun 68c⟩
   if (lun < 0) then
      write (*, '(A)') 'circe2_ls: no free unit'
      stop
   end if
   files: do i = 1, command_argument_count ()
      call get_command_argument (i, value = filename, status = status)
      if (status /= 0) then
         exit files
      else
         open (unit = lun, file = filename, status = 'old', iostat = status)
         if (status /= 0) then
            write (*, "(A,1X,A)") "circe2: can't open", trim(filename)
         else
```

```
            write (*, "(A,1X,A)") "file:", trim(filename)
            lines: do
               read (lun, '(A)', iostat = status) buffer
               if (status /= 0) exit lines
               if (buffer(1:7) == 'design,') then
                  read (lun, *) design, roots
                  read (lun, *)
                  read (lun, *) nc, polspt
                  ⟨Write design/beam data 73a⟩
                  ⟨Write channel header 73b⟩
               else if (buffer(1:5) == 'pid1,') then
                  read (lun, *) pid1, hel1, pid2, hel2, lumi
                  ⟨Write channel data 73c⟩
               end if
            end do lines
         end if
         close (unit = lun)
      end if
   end do files
end program circe2_ls
```

⟨*Separator* 49d⟩

73a ⟨*Write design/beam data* 73a⟩≡ (72)

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

73b ⟨*Write channel header* 73b⟩≡ (72)

```
write (*, '(4X,4(A5,2X),A)') &
     'pid#1', 'hel#1', 'pid#2', 'hel#2', 'luminosity / (10^32cm^-2sec^-1)'
```

73c ⟨*Write channel data* 73c⟩≡ (72)

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

A.4 β-distribitions

We need a fast generator of β-distribitions:

$$\beta^{a,b}_{x_{\min},x_{\max}}(x) = x^{a-1}(1-x)^{b-1} \cdot \frac{\Theta(x_{\max}-x)\Theta(x-x_{\min})}{I(x_{\min},a,b) - I(x_{\max},a,b)} \tag{59}$$

with the *incomplete Beta-function* I*:*

$$I(x,a,b) = \int_x^1 d\xi\, \xi^{a-1}(1-\xi)^{b-1} \tag{60}$$

$$I(0,a,b) = B(a,b) = \frac{\Gamma(a)\Gamma(b)}{\Gamma(a+b)} \tag{61}$$

This problem has been studied extensively [2] and we can use an algorithm [1] that is very fast for $0 < a \leq 1 \leq b$, which turns out to be the case in our application.

74a ⟨circe2_moments_library *declarations* 74a⟩≡ (90) 78d▷

```
public :: generate_beta
```

74b ⟨circe2_moments_library *implementation* 74b⟩≡ (90) 78f▷

```
subroutine generate_beta (rng, x, xmin, xmax, a, b)
  class(rng_type), intent(inout) :: rng
  real(kind=default), intent(out) :: x
  real(kind=default), intent(in) :: xmin, xmax, a, b
  real(kind=default) :: t, p, u, umin, umax, w
  ⟨Check a and b 74c⟩
  ⟨Set up generate_beta parameters 75a⟩
  do
     ⟨Generate a trial x and calculate its weight w 75b⟩
     call rng%generate (u)
     if (w > u) exit
  end do
end subroutine generate_beta
```

In fact, this algorithm works for $0 < a \leq 1 \leq b$ only:

74c ⟨Check a and b 74c⟩≡ (74b)

```
if (a >= 1 .or. b <= 1) then
   x = -1
   print *, 'ERROR: beta-distribution expects a<1<b'
   return
end if
```

The trick is to split the interval $[0,1]$ into two parts $[0,t]$ and $[t,1]$. In these intervals we obviously have

$$x^{a-1}(1-x)^{b-1} \leq \begin{cases} x^{a-1} & \text{for } x \leq t \\ t^{a-1}(1-x)^{b-1} & \text{for } x \geq t \end{cases} \tag{62}$$

because we have assumed that $0 < a < 1 < b$. The integrals of the two dominating distributions are t^a/a and $t^{a-1}(1-t)^b/b$ respectively and therefore the probability for picking a random number from the first interval is

$$P(x \leq t) = \frac{bt}{bt + a(1-t)^b} \tag{63}$$

We postpone the discussion of the choice of t until later:

75a ⟨*Set up* `generate_beta` *parameters* 75a⟩≡ (74b) 75c▷

```
⟨Set up best value for t 77a⟩
p = b*t / (b*t + a * (1 - t)**b)
```

The dominating distributions can be generated by simple mappings

$$\phi : [0,1] \to [0,1] \tag{64}$$

$$u \mapsto \begin{cases} t\left(\frac{u}{p}\right)^{\frac{1}{a}} & < t \text{ for } u < p \\ t & = t \text{ for } u = p \\ 1 - (1-t)\left(\frac{1-u}{1-p}\right)^{\frac{1}{b}} & > t \text{ for } u > p \end{cases} \tag{65}$$

The beauty of the algorithm is that we can use a single uniform deviate u for both intervals:

75b ⟨*Generate a trial* `x` *and calculate its weight* `w` 75b⟩≡ (74b)

```
call rng%generate (u)
u = umin + (umax - umin) * u
if (u <= p) then
   x = t * (u/p)**(1/a)
   w = (1 - x)**(b-1)
else
   x = 1 - (1 - t) * ((1 - u)/(1 - p))**(1/b)
   w = (x/t)**(a-1)
end if
```

The weights that are derived by dividing the distribution by the dominating distributions are already normalized correctly:

$$w : [0,1] \to [0,1] \tag{66}$$

$$x \mapsto \begin{cases} (1-x)^{b-1} & \in [(1-t)^{b-1}, 1] \text{ for } x \le t \\ \left(\frac{x}{t}\right)^{a-1} & \in [t^{1-a}, 1] \text{ for } x \ge t \end{cases} \tag{67}$$

To derive $u_{\min,\max}$ from $x_{\min,\max}$ we can use ϕ^{-1}:

$$\phi^{-1} : [0,1] \to [0,1] \tag{68}$$

$$x \mapsto \begin{cases} p\left(\frac{x}{t}\right)^{a} & < p \text{ for } x < t \\ p & = p \text{ for } x = t \\ 1 - (1-p)\left(\frac{1-x}{1-t}\right)^{b} & > p \text{ for } x > t \end{cases} \tag{69}$$

We start with $u_{\min}$. For efficiency, we handle the most common cases (small $x_{\min}$) first:

75c ⟨*Set up* `generate_beta` *parameters* 75a⟩+≡ (74b) ◁75a 76a▷

```
if (xmin <= 0) then
   umin = 0
elseif (xmin < t) then
   umin = p * (xmin/t)**a
elseif (xmin == t) then
   umin = p
elseif (xmin < 1) then
   umin = 1 - (1 - p) * ((1 - xmin)/(1 - t))**b
else
   umin = 1
endif
```

Same procedure for $u_{\max}$; again, handle the most common cases (large $x_{\max}$) first:

76a ⟨*Set up* `generate_beta` *parameters* 75a⟩+≡ (74b) ◁75c 76b▷

```
if (xmax >= 1) then
   umax = 1
elseif (xmax > t) then
   umax = 1 - (1 - p) * ((1 - xmax)/(1 - t))**b
elseif (xmax == t) then
   umax = p
elseif (xmax > 0) then
   umax = p * (xmax/t)**a
else
   umax = 0
endif
```

Check for absurd cases.

76b ⟨*Set up* `generate_beta` *parameters* 75a⟩+≡ (74b) ◁76a

```
if (umax < umin) then
   x = -1
   return
endif
```

It remains to choose he best value for t. The rejection efficiency ϵ of the algorithm is given by the ratio of the dominating distribution and the distribution

$$\frac{1}{\epsilon(t)} = \frac{B(a,b)}{ab}\left(bt^a + at^{a-1}(1-t)^b\right). \tag{70}$$

It is maximized for

$$bt - bt(1-t)^{b-1} + (a-1)(1-t)^b = 0 \tag{71}$$

This equation has a solution which can be determined numerically. While this determination is far too expensive compared to a moderate loss in efficiency,

we could perform it once after fitting the coefficients a, b. Nevertheless, it has been shown,[1] that

$$t = \frac{1-a}{b+1-a} \tag{72}$$

results in non-vanishing efficiency for all values $1 < a \leq 1 \leq b$. Empirically we have found efficiencies of at least 80% for this choice, which is enough for our needs.

77a ⟨*Set up best value for t* 77a⟩≡ (75a)

```
t = (1 - a) / (b + 1 - a)
```

A.5 Sampling

77b ⟨circe2_moments.f90 77b⟩≡ 90▷

```
module sampling
  use kinds
  implicit none
  private
  ⟨sampling declarations 77c⟩
contains
  ⟨sampling implementation 77e⟩
end module sampling
```

77c ⟨sampling *declarations* 77c⟩≡ (77b) 77d▷

```
type sample
  integer :: n = 0
  real(kind=default) :: w = 0
  real(kind=default) :: w2 = 0
end type sample
public :: sample
```

77d ⟨sampling *declarations* 77c⟩+≡ (77b) ◁77c 78a▷

```
public :: reset, record
```

77e ⟨sampling *implementation* 77e⟩≡ (77b) 77f▷

```
elemental subroutine reset (s)
  type(sample), intent(inout) :: s
  s%n = 0
  s%w = 0
  s%w2 = 0
end subroutine reset
```

77f ⟨sampling *implementation* 77e⟩+≡ (77b) ◁77e 78b▷

```
elemental subroutine record (s, w)
  type(sample), intent(inout) :: s
```

```
    real(kind=default), intent(in), optional :: w
    s%n = s%n + 1
    if (present (w)) then
      s%w = s%w + w
      s%w2 = s%w2 + w*w
    else
      s%w = s%w + 1
      s%w2 = s%w2 + 1
    endif
  end subroutine record
```

78a ⟨sampling *declarations* 77c⟩+≡ (77b) ◁77d

```
  public :: mean, variance
```

78b ⟨sampling *implementation* 77e⟩+≡ (77b) ◁77f 78c▷

```
  elemental function mean (s)
    type(sample), intent(in) :: s
    real(kind=default) :: mean
    mean = s%w / s%n
  end function mean
```

78c ⟨sampling *implementation* 77e⟩+≡ (77b) ◁78b

```
  elemental function variance (s)
    type(sample), intent(in) :: s
    real(kind=default) :: variance
    variance = (s%w2 / s%n - mean(s)**2) / s%n
    variance = max (variance, epsilon (variance))
  end function variance
```

A.6 Moments

This would probably be a good place for inheritance

78d ⟨circe2_moments_library *declarations* 74a⟩+≡ (90) ◁74a 78e▷

```
  type moment
    integer, dimension(2) :: n, m
    type(sample) :: sample = sample (0, 0.0_default, 0.0_default)
  end type moment
  public :: moment
```

78e ⟨circe2_moments_library *declarations* 74a⟩+≡ (90) ◁78d 79a▷

```
  public :: init_moments
```

78f ⟨circe2_moments_library *implementation* 74b⟩+≡ (90) ◁74b 79b▷

```
  subroutine init_moments (moments)
    type(moment), dimension(0:,0:,0:,0:), intent(inout) :: moments
    integer :: nx, mx, ny, my
```

```
    do nx = lbound(moments,1), ubound(moments,1)
       do mx = lbound(moments,2), ubound(moments,2)
          do ny = lbound(moments,3), ubound(moments,3)
             do my = lbound(moments,4), ubound(moments,4)
                moments(nx,mx,ny,my) = moment([nx,ny],[mx,my])
             end do
          end do
       end do
    end do
    call reset_moment (moments)
  end subroutine init_moments
```

79a ⟨circe2_moments_library *declarations* 74a⟩+≡ (90) ◁78e 80a▷

```
  public :: reset_moment, record_moment
```

79b ⟨circe2_moments_library *implementation* 74b⟩+≡ (90) ◁78f 79c▷

```
  elemental subroutine reset_moment (m)
    type(moment), intent(inout) :: m
    call reset (m%sample)
  end subroutine reset_moment
```

If we were pressed for time, we would compute the moments by iterative multiplications instead by powers, of course. In any case, we would like to combine `x1` and `x2` into an array. Unfortunately this is not possible for `elemental` procedures. NB: the NAG compiler appears to want a more careful evaluation of the powers. We enforce `0.0**0 == 0`.

79c ⟨circe2_moments_library *implementation* 74b⟩+≡ (90) ◁79b 80b▷

```
  elemental subroutine record_moment (m, x1, x2, w)
    type(moment), intent(inout) :: m
    real(kind=default), intent(in) :: x1, x2
    real(kind=default), intent(in), optional :: w
    real(kind=default) :: p
    p = pwr (x1, m%n(1)) * pwr (1-x1, m%m(1)) &
      * pwr (x2, m%n(2)) * pwr (1-x2, m%m(2))
    if (present (w)) p = p*w
    call record (m%sample, p)
  contains
    pure function pwr (x, n)
      real(kind=default), intent(in) :: x
      integer, intent(in) :: n
      real(kind=default) :: pwr
      if (n == 0) then
         pwr = 1
      else
         pwr = x**n
```

```
      end if
    end function pwr
  end subroutine record_moment
```

80a ⟨circe2_moments_library *declarations* 74a⟩+≡ (90) ◁79a 80d▷

```
  public :: mean_moment, variance_moment
```

80b ⟨circe2_moments_library *implementation* 74b⟩+≡ (90) ◁79c 80c▷

```
  elemental function mean_moment (m)
    type(moment), intent(in) :: m
    real(kind=default) :: mean_moment
    mean_moment = mean (m%sample)
  end function mean_moment
```

80c ⟨circe2_moments_library *implementation* 74b⟩+≡ (90) ◁80b 80e▷

```
  elemental function variance_moment (m)
    type(moment), intent(in) :: m
    real(kind=default) :: variance_moment
    variance_moment = variance (m%sample)
  end function variance_moment
```

A.6.1 Moments of β-distributions

80d ⟨circe2_moments_library *declarations* 74a⟩+≡ (90) ◁80a 81b▷

```
  public :: beta_moment
```

$$
\begin{aligned}
M_{n,m}(a,b) &= \int_0^1 \mathrm{d}x\, x^n (1-x)^m \beta_{0,1}^{a,b}(x) = \int_0^1 \mathrm{d}x\, x^n (1-x)^m \frac{x^{a-1}(1-x)^{b-1}}{B(a,b)} \\
&= \frac{1}{B(a,b)} \int_0^1 \mathrm{d}x\, x^{n+a-1}(1-x)^{m+b-1} = \frac{B(n+a,m+b)}{B(a,b)} \\
&= \frac{\Gamma(n+a)\Gamma(m+b)\Gamma(a+b)}{\Gamma(n+a+m+b)\Gamma(a)\Gamma(b)} = \frac{\Gamma(n+a)}{\Gamma(a)} \frac{\Gamma(m+b)}{\Gamma(b)} \frac{\Gamma(n+m+a+b)}{\Gamma(a+b)} \\
&= \frac{(a+n)(a+n-1)\cdots(a+1)a(b+m)(b+m-1)\cdots(b+1)b}{(a+b+n+m)(a+b+n+m-1)\cdots(a+b+1)(a+b)} \qquad (73)
\end{aligned}
$$

80e ⟨circe2_moments_library *implementation* 74b⟩+≡ (90) ◁80c 81a▷

```
  elemental function beta_moment (n, m, a, b)
    integer, intent(in) :: n, m
    real(kind=default), intent(in) :: a, b
    real(kind=default) :: beta_moment
    beta_moment = &
      gamma_ratio (a, n) * gamma_ratio (b, m) / gamma_ratio (a+b, n+m)
  end function beta_moment
```

$$\frac{\Gamma(x+n)}{\Gamma(x)} = (x+n)(x+n-1)\cdots(x+1)x \tag{74}$$

81a ⟨circe2_moments_library *implementation* 74b⟩+≡ (90) ◁80e 81d▷

```
elemental function gamma_ratio (x, n)
  real(kind=default), intent(in) :: x
  integer, intent(in) :: n
  real(kind=default) :: gamma_ratio
  integer :: i
  gamma_ratio = 1
  do i = 0, n - 1
    gamma_ratio = gamma_ratio * (x + i)
  end do
end function gamma_ratio
```

A.6.2 Channels

81b ⟨circe2_moments_library *declarations* 74a⟩+≡ (90) ◁80d 81c▷

```
type channel
  real(kind=default) :: w = 1
  real(kind=default), dimension(2) :: a = 1, b = 1
  logical, dimension(2) :: delta = .false.
end type channel
public :: channel
```

81c ⟨circe2_moments_library *declarations* 74a⟩+≡ (90) ◁81b 83a▷

```
public :: generate_beta_multi, beta_moments_multi
```

81d ⟨circe2_moments_library *implementation* 74b⟩+≡ (90) ◁81a 82b▷

```
subroutine generate_beta_multi (rng, x, channels)
  class(rng_type), intent(inout) :: rng
  real(kind=default), dimension(:), intent(out) :: x
  type(channel), dimension(:), intent(in) :: channels
  real(kind=default) :: u, accum
  integer :: i, n
  ⟨Select n according to the weight channels(n)%w 82a⟩
  do i = 1, size (x)
    if (channels(n)%delta(i)) then
      x(i) = 1
    else
      if (channels(n)%a(i) == 1 .and. channels(n)%b(i) == 1) then
        call rng%generate (x(i))
      else if (channels(n)%b(i) < channels(n)%a(i)) then
        call generate_beta (rng, x(i), 0.0_default, 1.0_default, &
                            channels(n)%b(i), channels(n)%a(i))
```

```
        x(i) = 1 - x(i)
      else
        call generate_beta (rng, x(i), 0.0_default, 1.0_default, &
                            channels(n)%a(i), channels(n)%b(i))
      end if
    end if
  end do
end subroutine generate_beta_multi
```

Subtlety: if the upper limit of the do loop where `size(channels)`, we could end up with `n` set to `size(channels)+1` when rounding errors produce `accum` $>$ `sum`(`channels`%`w`).

82a ⟨*Select* `n` *according to the weight* `channels(n)%w` 82a⟩≡ (81d)

```
call rng%generate (u)
u = u * sum (channels%w)
accum = 0
scan: do n = 1, size (channels) - 1
  accum = accum + channels(n)%w
  if (accum >= u) exit scan
end do scan
```

82b ⟨`circe2_moments_library` *implementation* 74b⟩+≡ (90) ◁81d 83b▷

```
pure function beta_moments_multi (n, m, channels)
  integer, intent(in), dimension(2) :: n, m
  type(channel), dimension(:), intent(in) :: channels
  real(kind=default) :: beta_moments_multi
  real(kind=default) :: w
  integer :: c, i
  beta_moments_multi = 0
  do c = 1, size (channels)
    w = channels(c)%w
    do i = 1, 2
      if (channels(c)%delta(i)) then
        if (m(i) > 0) w = 0
      else
        w = w * beta_moment (n(i), m(i), channels(c)%a(i), channels(c)%b(i))
      end if
    end do
    beta_moments_multi = beta_moments_multi + w
  end do
  beta_moments_multi = beta_moments_multi / sum (channels%w)
end function beta_moments_multi
```

A.6.3 Selftest

83a ⟨circe2_moments_library *declarations* 74a⟩+≡ (90) ◁81c 83c▷

```
public :: selftest
```

83b ⟨circe2_moments_library *implementation* 74b⟩+≡ (90) ◁82b 83d▷

```
subroutine selftest (rng, nevents)
  class(rng_type), intent(inout) :: rng
  integer, intent(in) :: nevents
  integer, parameter :: N = 1
  type(moment), dimension(0:N,0:N,0:N,0:N) :: moments
  integer :: i
  real(kind=default), dimension(2) :: x
  type(channel), dimension(:), allocatable :: channels
  call read_channels (channels)
  call init_moments (moments)
  do i = 1, nevents
     call generate_beta_multi (rng, x, channels)
     call record_moment (moments, x(1), x(2))
  end do
  call report_results (moments, channels)
end subroutine selftest
```

83c ⟨circe2_moments_library *declarations* 74a⟩+≡ (90) ◁83a 83e▷

```
public :: random2_seed
```

83d ⟨circe2_moments_library *implementation* 74b⟩+≡ (90) ◁83b 83f▷

```
subroutine random2_seed (rng, seed)
  class(rng_tao), intent(inout) :: rng
  integer, intent(in), optional:: seed
  integer, dimension(8) :: date_time
  integer :: seed_value
  if (present (seed)) then
     seed_value = seed
  else
     call date_and_time (values = date_time)
     seed_value = product (date_time)
  endif
  call rng%init (seed_value)
end subroutine random2_seed
```

83e ⟨circe2_moments_library *declarations* 74a⟩+≡ (90) ◁83c 84a▷

```
public :: read_channels
```

83f ⟨circe2_moments_library *implementation* 74b⟩+≡ (90) ◁83d 84b▷

```
subroutine read_channels (channels)
```

```
    type(channel), dimension(:), allocatable, intent(out) :: channels
    type(channel), dimension(100) :: buffer
    real(kind=default) :: w
    real(kind=default), dimension(2) :: a, b
    logical, dimension(2) :: delta
    integer :: n, status
    do n = 1, size (buffer)
      read (*, *, iostat = status) w, a(1), b(1), a(2), b(2), delta
      if (status == 0) then
        buffer(n) = channel (w, a, b, delta)
      else
        exit
      end if
    end do
    allocate (channels(n-1))
    channels = buffer(1:n-1)
  end subroutine read_channels
```

84a ⟨circe2_moments_library *declarations* 74a⟩+≡ (90) ◁83e 85a▷

```
  public :: report_results
```

84b ⟨circe2_moments_library *implementation* 74b⟩+≡ (90) ◁83f 85b▷

```
  subroutine report_results (moments, channels)
    type(moment), dimension(0:,0:,0:,0:), intent(in) :: moments
    type(channel), dimension(:), intent(in) :: channels
    integer :: nx, mx, ny, my
    real(kind=default) :: truth, estimate, sigma, pull, eps
    do nx = lbound(moments,1), ubound(moments,1)
       do mx = lbound(moments,2), ubound(moments,2)
          do ny = lbound(moments,3), ubound(moments,3)
             do my = lbound(moments,4), ubound(moments,4)
                truth = beta_moments_multi ([nx, ny], [mx, my], channels)
                estimate = mean_moment (moments(nx,mx,ny,my))
                sigma = sqrt (variance_moment (moments(nx,mx,ny,my)))
                pull = estimate - truth
                eps = pull / max (epsilon (1.0_default), epsilon (1.0_double))
                if (sigma /= 0.0_default) pull = pull / sigma
                write (*, "(' x^', I1, ' (1-x)^', I1, &
                          &' y^', I1, ' (1-y)^', I1, &
                          &': ', F8.5, ': est = ', F8.5, &
                          &' +/- ', F8.5,&
                          &', pull = ', F8.2,&
                          &', eps = ', F8.2)") &
                     nx, mx, ny, my, truth, estimate, sigma, pull, eps
             end do
```

```
            end do
         end do
      end do
   end subroutine report_results
```

85a ⟨circe2_moments_library *declarations* 74a⟩+≡ (90) ◁84a 86b▷

```
   public :: results_ok
```

85b ⟨circe2_moments_library *implementation* 74b⟩+≡ (90) ◁84b 86c▷

```
   function results_ok (moments, channels, threshold, fraction)
     ! use, intrinsic :: ieee_arithmetic
     type(moment), dimension(0:,0:,0:,0:), intent(in) :: moments
     type(channel), dimension(:), intent(in) :: channels
     real(kind=default), intent(in), optional :: threshold, fraction
     logical :: results_ok
     integer :: nx, mx, ny, my, failures
     real(kind=default) :: thr, frac, eps
     real(kind=default) :: truth, estimate, sigma
     ! we mut not expect to measure zero better than the
     ! double precision used in the ocaml code:
     eps = 200 * max (epsilon (1.0_default), epsilon (1.0_double))
     if (present(threshold)) then
        thr = threshold
     else
        thr = 5
     end if
     if (present(fraction)) then
        frac = fraction
     else
        frac = 0.01_default
     end if
     failures = 0
     do nx = lbound(moments,1), ubound(moments,1)
        do mx = lbound(moments,2), ubound(moments,2)
           do ny = lbound(moments,3), ubound(moments,3)
              do my = lbound(moments,4), ubound(moments,4)
                 truth = beta_moments_multi ([nx, ny], [mx, my], channels)
                 estimate = mean_moment (moments(nx,mx,ny,my))
                 sigma = sqrt (variance_moment (moments(nx,mx,ny,my)))
                 if (.not. (      ieee_is_normal (truth) &
                           .and. ieee_is_normal (estimate) &
                           .and. ieee_is_normal (sigma)) &
                     .or. abs (estimate - truth) > max (thr * sigma, eps)) then
                    failures = failures + 1
                 end if
```

```
            end do
          end do
        end do
      end do
      if (failures >= frac * size (moments)) then
        results_ok = .false.
      else
        results_ok = .true.
      end if
    contains
      ⟨The old ieee_is_normal kludge 86a⟩
    end function results_ok
```

gfortran doesn't have the intrinsic `ieee_arithmetic` module yet ...

86a ⟨*The old* `ieee_is_normal` *kludge* 86a⟩≡ (85b)

```
function ieee_is_normal (x)
  real(kind=default), intent(in) :: x
  logical :: ieee_is_normal
  ieee_is_normal = .not. (x /= x)
end function ieee_is_normal
```

A.6.4 Generate Sample

86b ⟨`circe2_moments_library` *declarations* 74a⟩+≡ (90) ◁85a 86d▷

```
public :: generate
```

86c ⟨`circe2_moments_library` *implementation* 74b⟩+≡ (90) ◁85b 87a▷

```
subroutine generate (rng, nevents)
  class(rng_type), intent(inout) :: rng
  integer, intent(in) :: nevents
  type(channel), dimension(:), allocatable :: channels
  real(kind=default), dimension(2) :: x
  integer :: i
  call read_channels (channels)
  do i = 1, nevents
     call generate_beta_multi (rng, x, channels)
     write (*, "(3(5x,F19.17))") x, 1.0_default
  end do
end subroutine generate
```

A.6.5 List Moments

86d ⟨`circe2_moments_library` *declarations* 74a⟩+≡ (90) ◁86b 87b▷

```
public :: compare
```

87a ⟨circe2_moments_library *implementation* 74b⟩+≡ (90) ◁86c 87c▷

```
subroutine compare (rng, nevents, file)
  class(rng_type), intent(inout) :: rng
  integer, intent(in) :: nevents
  character(len=*), intent(in) :: file
  type(channel), dimension(:), allocatable :: channels
  integer, parameter :: N = 1
  type(moment), dimension(0:N,0:N,0:N,0:N) :: moments
  real(kind=default), dimension(2) :: x
  character(len=128) :: design
  real(kind=default) :: roots
  integer :: ierror
  integer, dimension(2) :: p, h
  integer :: i
  type(circe2_state) :: c2s
  call read_channels (channels)
  call init_moments (moments)
  design = "CIRCE2/TEST"
  roots = 42
  p = [11, -11]
  h = 0
  call circe2_load (c2s, trim(file), trim(design), roots, ierror)
  do i = 1, nevents
     call circe2_generate (c2s, rng, x, p, h)
     call record_moment (moments, x(1), x(2))
  end do
  call report_results (moments, channels)
end subroutine compare
```

A.6.6 Check Generator

87b ⟨circe2_moments_library *declarations* 74a⟩+≡ (90) ◁86d

```
public :: check
```

87c ⟨circe2_moments_library *implementation* 74b⟩+≡ (90) ◁87a

```
subroutine check (rng, nevents, file, distributions, fail)
  class(rng_type), intent(inout) :: rng
  integer, intent(in) :: nevents
  character(len=*), intent(in) :: file
  logical, intent(in), optional :: distributions, fail
  type(channel), dimension(:), allocatable :: channels
  type(channel), dimension(1) :: unit_channel
  integer, parameter :: N = 1
  type(moment), dimension(0:N,0:N,0:N,0:N) :: moments, unit_moments
```

```
real(kind=default), dimension(2) :: x
character(len=128) :: design
real(kind=default) :: roots, weight
integer :: ierror
integer, dimension(2) :: p, h
integer :: i
logical :: generation_ok, distributions_ok
logical :: check_distributions, expect_failure
type(circe2_state) :: c2s
if (present (distributions)) then
   check_distributions = distributions
else
   check_distributions = .true.
end if
if (present (fail)) then
   expect_failure = fail
else
   expect_failure = .false.
end if
call read_channels (channels)
call init_moments (moments)
if (check_distributions) call init_moments (unit_moments)
design = "CIRCE2/TEST"
roots = 42
p = [11, -11]
h =   0
call circe2_load (c2s, trim(file), trim(design), roots, ierror)
do i = 1, nevents
   call circe2_generate (c2s, rng, x, p, h)
   call record_moment (moments, x(1), x(2))
   if (check_distributions) then
      weight = circe2_distribution (c2s, p, h, x)
      call record_moment (unit_moments, x(1), x(2), w = 1 / weight)
   end if
end do
generation_ok = results_ok (moments, channels)
if (check_distributions) then
   distributions_ok = results_ok (unit_moments, unit_channel)
else
   distributions_ok = .not. expect_failure
end if
if (expect_failure) then
   if (generation_ok .and. distributions_ok) then
```

```
            print *, "FAIL: unexpected success"
         else
            if (.not. generation_ok) then
               print *, "OK: expected failure in generation"
            end if
            if (.not. distributions_ok) then
               print *, "OK: expected failure in distributions"
            end if
         end if
         call report_results (moments, channels)
      else
         if (generation_ok .and. distributions_ok) then
            print *, "OK"
         else
            if (.not. generation_ok) then
               print *, "FAIL: generation"
               call report_results (moments, channels)
            end if
            if (.not. distributions_ok) then
               print *, "FAIL: distributions"
               call report_results (unit_moments, unit_channel)
            end if
         end if
      end if
   end subroutine check
```

A.7 circe2_moments: Compare Moments of distributions

89 ⟨*Main program* 89⟩≡ (91)

```
   program circe2_moments
      use circe2
      use circe2_moments_library !NODEP!
      use tao_random_objects !NODEP!
      implicit none
      type(rng_tao), save :: rng
      character(len=1024) :: mode, filename, buffer
      integer :: status, nevents, seed
      call get_command_argument (1, value = mode, status = status)
      if (status /= 0) mode = ""
      call get_command_argument (2, value = filename, status = status)
      if (status /= 0) filename = ""
      call get_command_argument (3, value = buffer, status = status)
```

```
    if (status == 0) then
       read (buffer, *, iostat = status) nevents
       if (status /= 0) nevents = 1000
    else
       nevents = 1000
    end if
    call get_command_argument (4, value = buffer, status = status)
    if (status == 0) then
       read (buffer, *, iostat = status) seed
       if (status == 0) then
          call random2_seed (rng, seed)
       else
          call random2_seed (rng)
       end if
    else
       call random2_seed (rng)
    end if
    select case (trim (mode))
    case ("check")
       call check (rng, nevents, trim (filename))
    case ("!check")
       call check (rng, nevents, trim (filename), fail = .true.)
    case ("check_generation")
       call check (rng, nevents, trim (filename), distributions = .false.)
    case ("!check_generation")
       call check (rng, nevents, trim (filename), fail = .true., &
                                                  distributions = .false.)
    case ("compare")
       call compare (rng, nevents, trim (filename))
    case ("generate")
       call generate (rng, nevents)
    case ("selftest")
       call selftest (rng, nevents)
    case default
       print *, &
        "usage: circe2_moments " // &
        "[check|check_generation|generate|selftest] " // &
        "filename [events] [seed]"
    end select
  end program circe2_moments
```

90 ⟨circe2_moments.f90 77b⟩+≡ ◁77b 91▷

```
  module circe2_moments_library
    use kinds
```

```
    use tao_random_objects !NODEP!
    use sampling !NODEP!
    use circe2
    implicit none
    private
    ⟨circe2_moments_library declarations 74a⟩
  contains
    ⟨circe2_moments_library implementation 74b⟩
  end module circe2_moments_library
```

91 ⟨circe2_moments.f90 77b⟩+≡ ◁90
⟨*Main program* 89⟩

References

[1] A. Atkinson and J. Whittaker, Appl. Stat. **28**, 90 (1979).

[2] L. Devroye, *Non-uniform Random Variate Generation*, Springer, 1986.

B Making Grids

B.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
```

B.2 Implementation of *Float*

```
open Printf
```

```
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 =
  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
```

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

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

  let input_binary_float = input_float_little_endian

  let input_binary_floats ic array =
    for i = 0 to Array.length array - 1 do
      array.(i) ← input_binary_float ic
    done

end
```

B.3 Interface of *ThoArray*

exception *Out_of_bounds* of *int* × *int*

Interpret optional array boundaries. Assuming that *Array.length* $a \mapsto n$, we have

- *decode_inf* $a \mapsto 0$
- *decode_sup* $a \mapsto n - 1$
- *decode_inf* ~*inf* : i $a \mapsto i$ for $0 \leq i \leq n - 1$
- *decode_sup* ~*sup* : i $a \mapsto i$ for $0 \leq i \leq n - 1$
- *decode_inf* ~*inf* : $(-i)$ $a \mapsto n - i$ for $1 \leq i \leq n$
- *decode_sup* ~*sup* : $(-i)$ $a \mapsto n - i$ for $1 \leq i \leq n$
- *decode_inf* ~*inf* : i a raises *Out_of_bounds* for $i \geq n \vee i < -n$

- *decode_sup ~sup* : *i a* raises *Out_of_bounds* for $i \geq n \vee i < -n$

In particular

- *decode_inf ~inf* : (-2) $a \mapsto n - 2$, i. e. the idex of the next-to-last element.
- *decode_sup ~sup* : (-1) $a \mapsto n - 1$, i. e. the idex of the last element.

val *decode_inf* : *?inf* :*int* → α *array* → *int*
val *decode_sup* : *?sup* :*int* → α *array* → *int*

Just like the functions from *Array* of the same name, but acting only on the subarray specified by the optional *~inf* and *~sup*, interpreted as above. E. g. *copy ~inf* : 1 *~sup* : (-2) *a* chops off the first and last elements.

val *map* : *?inf* :*int* → *?sup* :*int* → $(\alpha \to \beta)$ → α *array* → β *array*
val *copy* : *?inf* :*int* → *?sup* :*int* → α *array* → α *array*
val *iter* : *?inf* :*int* → *?sup* :*int* → $(\alpha \to unit)$ → α *array* → *unit*
val *fold_left* : *?inf* :*int* → *?sup* :*int* →
$(\alpha \to \beta \to \alpha) \to \alpha \to \beta$ *array* → α

A convenience function.

val *sum_float* : *?inf* :*int* → *?sup* :*int* → *float array* → *float*

val *suite* : *OUnit.test*

B.4 Implementation of *ThoArray*

```
exception Out_of_bounds of int × int

let decode_limit i a =
  let n = Array.length a in
  if i ≥ n then
    raise (Out_of_bounds (i, n))
  else if i ≥ 0 then
    i
  else if i ≥ − n then
    n + i
  else
    raise (Out_of_bounds (i, n))

let decode_inf ?inf a =
  match inf with
  | None → 0
  | Some i → decode_limit i a
```

```
let decode_sup ?sup a =
  match sup with
  | None -> Array.length a - 1
  | Some i -> decode_limit i a

let decode_limit_suite =
  let ten = Array.init 10 (fun i -> i) in
  let open OUnit in
  "decode_limit" >:::
  ["0" >:: (fun () -> assert_equal 0 (decode_limit 0 ten));
   "9" >:: (fun () -> assert_equal 9 (decode_limit 9 ten));
   "10" >::
   (fun () ->
     assert_raises (Out_of_bounds (10, 10))
       (fun () -> decode_limit 10 ten));
   "-1" >:: (fun () -> assert_equal 9 (decode_limit (-1) ten));
   "-10" >:: (fun () -> assert_equal 0 (decode_limit (-10) ten));
   "-11" >::
   (fun () ->
     assert_raises (Out_of_bounds (-11, 10))
       (fun () -> decode_limit (-11) ten))]

let map ?inf ?sup f a =
  let n = decode_inf ?inf a in
  Array.init (decode_sup ?sup a - n + 1) (fun i -> f a.(n + i))

let copy ?inf ?sup a =
  map ?inf ?sup (fun x -> x) a

let map_suite =
  let five = Array.init 5 succ in
  let twice n = 2 * n in
  let open OUnit in
  "map" >:::
  ["2␣*␣.." >:: (fun () ->
    assert_equal [|2;4;6;8;10|] (map twice five));
   "2␣*␣1.." >:: (fun () ->
     assert_equal [|4;6;8;10|] (map twice ~inf:1 five));
   "2␣*␣..-2" >:: (fun () ->
     assert_equal [|2;4;6;8|] (map twice ~sup:(-2) five));
   "2␣*␣1..-2" >:: (fun () ->
     assert_equal [|4;6;8|] (map twice ~inf:1 ~sup:(-2) five));
   "2␣*␣1..2" >:: (fun () ->
```

```
      assert_equal [|4;6|] (map twice ~inf:1 ~sup:2 five))]

let copy_suite =
  let five = Array.init 5 succ in
  let open OUnit in
  "copy" >:::
  [".." >:: (fun () -> assert_equal five (copy five));
   "1.." >:: (fun () -> assert_equal [|2;3;4;5|] (copy ~inf:1 five));
   "..-2" >:: (fun () -> assert_equal [|1;2;3;4|] (copy ~sup:(-2) five));
   "1..-2" >:: (fun () -> assert_equal [|2;3;4|] (copy ~inf:1 ~sup:
(-2) five));
   "1..2" >:: (fun () -> assert_equal [|2;3|] (copy ~inf:1 ~sup:2 five))]

let fold_left ?inf ?sup f x a =
  let acc = ref x in
  try
    for i = decode_inf ?inf a to decode_sup ?sup a do
      acc := f !acc a.(i)
    done;
    !acc
  with
  | Out_of_bounds (_, _) -> x

let iter ?inf ?sup f a =
  fold_left ?inf ?sup (fun () x -> f x) () a

let iter ?inf ?sup f a =
  try
    for i = decode_inf ?inf a to decode_sup ?sup a do
      f a.(i)
    done
  with
  | Out_of_bounds (_, _) -> ()

let sum_float ?inf ?sup a =
  fold_left ?inf ?sup (+.) 0.0 a

let sum_float_suite =
  let ten = Array.init 10 (fun i -> float i +. 1.0) in
  let open OUnit in
  "sum_float" >:::
  [".." >:: (fun () -> assert_equal 55.0 (sum_float ten));
   "1.." >:: (fun () -> assert_equal 54.0 (sum_float ~inf:1 ten));
   "..-2" >:: (fun () -> assert_equal 45.0 (sum_float ~sup:(-2) ten));
```

```
    "1..-2" >:: (fun () → assert_equal 44.0 (sum_float ~inf : 1 ~sup :
(−2) ten));
    "1..2" >:: (fun () → assert_equal 5.0 (sum_float ~inf : 1 ~sup : 2 ten))]

let suite =
  let open OUnit in
  "Array" >:::
  [decode_limit_suite;
   map_suite;
   copy_suite;
   sum_float_suite]
```

B.5 Interface of *ThoMatrix*

```
val copy : ?inf1 :int → ?sup1 :int → ?inf2 :int → ?sup2 :int →
  α array array → α array array
val map : ?inf1 :int → ?sup1 :int → ?inf2 :int → ?sup2 :int →
  (α → β) → α array array → β array array
val iter : ?inf1 :int → ?sup1 :int → ?inf2 :int → ?sup2 :int →
  (α → unit) → α array array → unit
val fold_left : ?inf1 :int → ?sup1 :int → ?inf2 :int → ?sup2 :int →
  (α → β → α) → α → β array array → α
val sum_float : ?inf1 :int → ?sup1 :int → ?inf2 :int → ?sup2 :int →
  float array array → float
val size : α array array → int

val transpose : α array array → α array array

val suite : OUnit.test
```

B.6 Implementation of *ThoMatrix*

```
let map ?inf1 ?sup1 ?inf2 ?sup2 f a =
  ThoArray.map ?inf : inf1 ?sup : sup1
    (ThoArray.map ?inf : inf2 ?sup : sup2 f) a

let copy ?inf1 ?sup1 ?inf2 ?sup2 a =
  map ?inf1 ?sup1 ?inf2 ?sup2 (fun x → x) a

let iter ?inf1 ?sup1 ?inf2 ?sup2 f a =
  ThoArray.iter ?inf : inf1 ?sup : sup1
    (ThoArray.iter ?inf : inf2 ?sup : sup2 f) a
```

```
let fold_left ?inf1 ?sup1 ?inf2 ?sup2 f x a =
  ThoArray.fold_left ?inf : inf1 ?sup : sup1
    (ThoArray.fold_left ?inf : inf2 ?sup : sup2 f) x a

let sum_float ?inf1 ?sup1 ?inf2 ?sup2 a =
  fold_left ?inf1 ?sup1 ?inf2 ?sup2 (+.) 0.0 a

let size a =
  Array.fold_left (fun acc v -> Array.length v + acc) 0 a

let transpose a =
  let n1 = Array.length a
  and n2 = Array.length a.(0) in
  let a' = Array.make_matrix n2 n1 a.(0).(0) in
  for i1 = 0 to pred n1 do
    for i2 = 0 to pred n2 do
      a'.(i2).(i1) <- a.(i1).(i2)
    done
  done;
  a'

let suite =
  let open OUnit in
  "Matrix" >:::
  []
```

B.7 Interface of *Filter*

```
type t
val unit : t
val gaussian : float -> t
val apply : ?inf :int -> ?sup :int -> t -> float array -> float array
val apply1 : ?inf1 :int -> ?sup1 :int -> ?inf2 :int -> ?sup2 :int ->
  t -> float array array -> float array array
val apply2 : ?inf1 :int -> ?sup1 :int -> ?inf2 :int -> ?sup2 :int ->
  t -> float array array -> float array array
val apply12 : ?inf1 :int -> ?sup1 :int -> ?inf2 :int -> ?sup2 :int ->
  t -> t -> float array array -> float array array

exception Out_of_bounds of int × int

val suite : OUnit.test
```

B.8 Implementation of *Filter*

```
exception Out_of_bounds of int × int
```

We will assume $left.(0) = center = right.(0)$ and use only $center$.

```
type t' =
    { left' : float array;
      center' : float;
      right' : float array }

type t =
    { left : float array;
      center : float;
      right : float array;
      norm : float array array }

let unit =
   { left = [| 1.0 |];
     center = 1.0;
     right = [| 1.0 |];
     norm = [| [| 1.0 |] |] }

let normalize f =
  let left_sum = ThoArray.sum_float ~inf:1 f.left'
  and right_sum = ThoArray.sum_float ~inf:1 f.right' in
  let norm = f.center' +. left_sum +. right_sum in
  let left = Array.map (fun x -> x /. norm) f.left'
  and center = f.center' /. norm
  and right = Array.map (fun x -> x /. norm) f.right' in
  let norm =
    Array.make_matrix (Array.length left) (Array.length right) center in
  for i = 1 to Array.length left - 1 do
    norm.(i).(0) <- norm.(pred i).(0) +. left.(i)
  done;
  for i = 0 to Array.length left - 1 do
    for j = 1 to Array.length right - 1 do
      norm.(i).(j) <- norm.(i).(pred j) +. right.(j)
    done
  done;
  { left; center; right; norm }

let upper x =
  truncate (ceil x)
```

```
let gaussian width =
  let n = upper (width *. sqrt (2. *. log 10^6)) in
  let weights =
    Array.init (succ n) (fun i -> exp (-. 0.5 *. (float i /. width) ** 2.)) in
  normalize
    { left' = weights;
      center' = 1.0;
      right' = weights }
```

Idea: avoid bleeding into empty regions by treating their edges like boundaries.

```
let apply ?inf ?sup f a =
  let inf = ThoArray.decode_inf ?inf a
  and sup = ThoArray.decode_sup ?sup a in
  let n_left = Array.length f.left
  and n_right = Array.length f.right
  and a' = Array.copy a in
  for i = inf to sup do
    let num_left = min (pred n_left) (i - inf)
    and num_right = min (pred n_right) (sup - i) in
    let sum = ref (f.center *. a.(i)) in
    for j = 1 to num_left do
      sum := !sum +. f.left.(j) *. a.(i - j)
    done;
    for j = 1 to num_right do
      sum := !sum +. f.right.(j) *. a.(i + j)
    done;
    a'.(i) <- !sum /. f.norm.(num_left).(num_right)
  done;
  a'

module Real =
  struct
    type t = float
    let compare = compare
    let compare x y =
      if abs_float (x -. y) <=
        Float.Double.epsilon *. (max (abs_float x) (abs_float y)) then
        0
      else if x < y then
        -1
      else
```

```
        1
    let pp_printer = Format.pp_print_float
    let pp_print_sep = OUnitDiff.pp_comma_separator
  end

module Reals = OUnitDiff.ListSimpleMake (Real)

let array_assert_equal a1 a2 =
  Reals.assert_equal (Array.to_list a1) (Array.to_list a2)

let limits_suite =
  let fence = Array.init 10 (fun i -> if i = 0 ∨ i = 9 then 1.0 else 0.0) in
  let open OUnit in
  "limits" >:::
  ["1..-2" >::
   (fun () ->
     array_assert_equal fence
       (apply ~inf : 1 ~sup : (−2) (gaussian 10.0) fence))]

let norm_suite =
  let flat = Array.make 10 1.0 in
  let open OUnit in
  "norm" >:::
  ["gausian␣1" >::
   (fun () ->
     array_assert_equal flat (apply (gaussian 1.0) flat));
   "gausian␣5" >::
   (fun () ->
     array_assert_equal flat (apply (gaussian 5.0) flat));
   "gausian␣10" >::
   (fun () ->
     array_assert_equal flat (apply (gaussian 10.0) flat))]

let apply_suite =
  let open OUnit in
  "apply" >:::
  [limits_suite;
   norm_suite]

let array_map ?inf ?sup f a =
  let a' = Array.copy a in
  for i = ThoArray.decode_inf ?inf a to ThoArray.decode_sup ?sup a do
    a'.(i) ← f a.(i)
  done;
  a'
```

```
let array_map_suite =
  let five = Array.init 5 (fun i -> float (succ i)) in
  let open OUnit in
  "array_map" >:::
  ["..-2" >::
   (fun () ->
     array_assert_equal [| 2.0; 4.0; 6.0; 8.0; 5.0 |]
       (array_map ~sup:(-2) (fun x -> 2.0 *. x) five));
   "2.." >::
   (fun () ->
     array_assert_equal [| 1.0; 2.0; 6.0; 8.0; 10.0 |]
       (array_map ~inf:2 (fun x -> 2.0 *. x) five));
   "1..-2" >::
   (fun () ->
     array_assert_equal [| 1.0; 4.0; 6.0; 8.0; 5.0 |]
       (array_map ~inf:1 ~sup:(-2) (fun x -> 2.0 *. x) five))]

let apply1 ?inf1 ?sup1 ?inf2 ?sup2 f a =
  ThoMatrix.transpose
    (array_map ?inf:inf2 ?sup:sup2
       (apply ?inf:inf1 ?sup:sup1 f)
       (ThoMatrix.transpose a))

let apply2 ?inf1 ?sup1 ?inf2 ?sup2 f a =
  array_map ?inf:inf1 ?sup:sup1
    (apply ?inf:inf2 ?sup:sup2 f) a

let apply12 ?inf1 ?sup1 ?inf2 ?sup2 f1 f2 a =
  array_map ?inf:inf1 ?sup:sup1
    (apply ?inf:inf2 ?sup:sup2 f2)
    (ThoMatrix.transpose
       (array_map ?inf:inf2 ?sup:sup2
          (apply ?inf:inf1 ?sup:sup1 f1)
          (ThoMatrix.transpose a)))

let apply12_suite =
  let open OUnit in
  "apply12" >:::
  []

let suite =
  let open OUnit in
  "Filter" >:::
  [apply_suite;
```

array_map_suite;
apply12_suite]

B.9 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* : $t \rightarrow$ *domain*
val *x_max* : $t \rightarrow$ *domain*

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

type *codomain*
val *y_min* : $t \rightarrow$ *codomain*
val *y_max* : $t \rightarrow$ *codomain*

the map proper

$$\begin{aligned} \phi : [x_{\min}, x_{\max}] &\rightarrow [y_{\min}, y_{\max}] \\ x &\mapsto y = \phi(x) \end{aligned} \tag{75}$$

val *phi* : $t \rightarrow$ *domain* $\rightarrow$ *codomain*

the inverse map

$$\begin{aligned} \phi^{-1} : [y_{\min}, y_{\max}] &\rightarrow [x_{\min}, x_{\max}] \\ y &\mapsto x = \phi^{-1}(y) \end{aligned} \tag{76}$$

val *ihp* : $t \rightarrow$ *codomain* $\rightarrow$ *domain*

the jacobian of the map

$$\begin{aligned} J : [x_{\min}, x_{\max}] &\rightarrow \mathbf{R} \\ x &\mapsto J(x) = \frac{\mathrm{d}\phi}{\mathrm{d}x}(x) \end{aligned} \tag{77}$$

val *jac* : $t \rightarrow$ *domain* $\rightarrow$ *float*

and finally the jacobian of the inverse map

$$\begin{aligned} J^* : [y_{\min}, y_{\max}] &\rightarrow \mathbf{R} \\ y &\mapsto J^*(y) = \frac{\mathrm{d}\phi^{-1}}{\mathrm{d}y}(y) = \left(\frac{\mathrm{d}\phi}{\mathrm{d}x}(\phi^{-1}(y))\right)^{-1} \end{aligned} \tag{78}$$

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*

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*

B.10 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*

B.11 Specific Real Maps

module *Id* :
  sig
    include *Real*

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

$$\begin{aligned} \iota : [x_{\min}, x_{\max}] &\to [x_{\min}, x_{\max}] \\ x &\mapsto \iota(x) = x \end{aligned} \tag{79}$$

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.

$$\begin{aligned} \lambda_{a,b} : [x_{\min}, x_{\max}] &\to [y_{\min}, y_{\max}] \\ x &\mapsto \lambda_{a,b}(x) = ax + b \end{aligned} \tag{80}$$

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.

$$\begin{aligned} \psi_{a,b}^{\alpha,\xi,\eta} : [x_{\min}, x_{\max}] &\to [y_{\min}, y_{\max}] \\ x &\mapsto \psi_{a,b}^{\alpha,\xi,\eta}(x) = \frac{1}{b}(a(x-\xi))^{\alpha} + \eta \end{aligned} \tag{81}$$

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 $[x_{\min}, x_{\max}] \to [y_{\min}, y_{\max}]$.

$$\begin{aligned} \rho_{a,b}^{\xi,\eta} : [x_{\min}, x_{\max}] &\to [y_{\min}, y_{\max}] \\ x &\mapsto \rho_{a,b}^{\xi,\eta}(x) = a \tan\left(\frac{a}{b^2}(x-\xi)\right) + \eta \end{aligned} \tag{82}$$

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
```

B.12 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

  end

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

B.13 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
```

B.14 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 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 closure ~x_min ~x_max ~y_min ~y_max ~a ~b =
```

$$x \mapsto \lambda_{a,b}(x) = ax + b \tag{83}$$

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

$$y \mapsto (\lambda_{a,b})^{-1}(y) = \frac{y-b}{a} \tag{84}$$

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

$$x \mapsto \psi_{a,b}^{\alpha,\xi,\eta}(x) = \frac{1}{b}(a(x-\xi))^{\alpha} + \eta \tag{85}$$

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

$$y \mapsto (\psi_{a,b}^{\alpha,\xi,\eta})^{-1}(y) = \frac{1}{a}(b(y-\eta))^{1/\alpha} + \xi \tag{86}$$

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

$$\frac{\mathrm{d}y}{\mathrm{d}x}(x) = \frac{a\alpha}{b}(a(x-\xi))^{\alpha-1} \tag{87}$$

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

$$\frac{\mathrm{d}x}{\mathrm{d}y}(y) = \frac{b}{a\alpha}(b(y-\eta))^{1/\alpha-1} \tag{88}$$

```
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 }
```

$$a_i = \frac{(b_i(y_i-\eta_i))^{1/\alpha_i} - (b_i(y_{i-1}-\eta_i))^{1/\alpha_i}}{x_i - x_{i-1}} \tag{89a}$$

$$\xi_i = \frac{x_{i-1}|y_i-\eta_i|^{1/\alpha_i} - x_i|y_{i-1}-\eta_i|^{1/\alpha_i}}{|y_i-\eta_i|^{1/\alpha_i} - |y_{i-1}-\eta_i|^{1/\alpha_i}} \tag{89b}$$

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

$$x \mapsto \rho_{a,b}^{\xi,\eta}(x) = a \tan\left(\frac{a}{b^2}(x-\xi)\right) + \eta \tag{90}$$

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

$$y \mapsto (\rho_{a,b}^{\xi,\eta})^{-1}(y) = \frac{b^2}{a}\mathrm{atan}\left(\frac{y-\eta}{a}\right) + \xi \tag{91}$$

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

$$\frac{\mathrm{d}y}{\mathrm{d}x}(x(y)) = \frac{1}{\dfrac{\mathrm{d}x}{\mathrm{d}y}(y)} = \left(\frac{b^2}{(y-\eta)^2 + a^2}\right)^{-1} \tag{92}$$

```
  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 }
```

$$b_i = \sqrt{a_i \frac{x_i - x_{i-1}}{\operatorname{atan}\left(\frac{y_i-\eta_i}{a_i}\right) - \operatorname{atan}\left(\frac{y_{i-1}-\eta_i}{a_i}\right)}} \tag{93a}$$

$$\xi_i = \frac{x_{i-1}\operatorname{atan}\left(\frac{y_i-\eta_i}{a_i}\right) - x_i\operatorname{atan}\left(\frac{y_{i-1}-\eta_i}{a_i}\right)}{x_i - x_{i-1}} \tag{93b}$$

```
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
```

B.15 Interface of *Diffmaps*

B.16 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
```

B.17 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;
          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 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;
    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;
    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;
    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

B.18 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 $copy\ :\ t\ \rightarrow\ t$

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 $find\ :\ t\ \rightarrow\ float \rightarrow\ int$

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

val $record\ :\ t\ \rightarrow\ float \rightarrow\ float \rightarrow\ unit$

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

val $rebin\ :\ ?power\ :float \rightarrow\ ?fixed\_min\ :bool \rightarrow\ ?fixed\_max\ :bool \rightarrow\ t\ \rightarrow\ t$

$J^*(y)$


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

val $caj\ :\ t\ \rightarrow\ float \rightarrow\ float$

val $n\_bins\ :\ t\ \rightarrow\ int$
val $bins\ :\ t\ \rightarrow\ float\ array$
val $to\_channel\ :\ out\_channel\ \rightarrow\ t\ \rightarrow\ unit$

end

exception *Above_max* of *float* × (*float* × *float*) × *int*
exception *Below_min* of *float* × (*float* × *float*) × *int*
exception *Out_of_range* of *float* × (*float* × *float*)
exception *Rebinning_failure* of *string*

B.18.1 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 $x \rightarrow 1.0$, of course.

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

  end

module *Mono* : *Mono*

B.18.2 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*

B.19 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 Above_max of float * (float * float) * int
exception Below_min of float * (float * float) * int
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 then
    raise (Below_min (x, (d.(0), d.(n_max)), 0))
  else if x > d.(n_max) +. eps then
    raise (Above_max (x, (d.(0), d.(n_max)), n_max - 1))
  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
  end
```

B.19.1 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.make n 0;
    w = Array.make n 0.0;
    w2 = Array.make 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]"
```

$$
\begin{aligned}
d_1 &\to \frac{1}{2}(d_1 + d_2) \\
d_2 &\to \frac{1}{3}(d_1 + d_2 + d_3) \\
&\dots \\
d_{n-1} &\to \frac{1}{3}(d_{n-2} + d_{n-1} + d_n) \\
d_n &\to \frac{1}{2}(d_{n-1} + d_n)
\end{aligned}
\tag{94}
$$

```
let smooth3 f =
  match Array.length f with
  | 0 → f
  | 1 → Array.copy f
  | 2 → Array.make 2 ((f.(0) +. f.(1)) /. 2.0)
  | n →
      let f' = Array.make 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'
```

$$
m_i = \left(\frac{\frac{\bar{f}_i \Delta x_i}{\sum_j \bar{f}_j \Delta x_j} - 1}{\ln\left(\frac{\bar{f}_i \Delta x_i}{\sum_j \bar{f}_j \Delta x_j}\right)} \right)^{\alpha}
\tag{95}
$$

```
let rebinning_weights' power fs =
  let sum_f = Array.fold_left (+.) 0.0 fs in
  if sum_f ≤ 0.0 then
    Array.make (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.make (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.

```
      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.make n 0;
      w = Array.make n 0.0;
      w2 = Array.make 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

end
```

B.19.2 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.make (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.make 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.make ndiv 0;
        w = Array.make ndiv 0.0;
        w2 = Array.make ndiv 0.0 }
```

We can safely assume that *find_raw pd.x y = find_raw pd.x x*.

$$w = \frac{f}{\frac{\mathrm{d}x}{\mathrm{d}y}} = f \cdot \frac{\mathrm{d}y}{\mathrm{d}x} \tag{96}$$

Here, the jacobian makes 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.make ndiv 0;
      w = Array.make ndiv 0.0;
      w2 = Array.make 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

end
```

module *Poly* = *Make_Poly* (*Diffmaps.Default*)

B.20 Interface of *Grid*

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

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* →
            ?*areas* : *Syntax.area list* →
              (*float*, *Bigarray.float64_elt*,
                Bigarray.fortran_layout) *Bigarray.Array2.t* → *t* → *t*

```
    val smooth : float → Syntax.area → t → t
    val variance_area : Syntax.area → t → float
    val to_channel_2d : out_channel → t → unit
    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 variance : t → float
  end
module Make (D : Division.T) : T with module D = D
```

B.21 Implementation of *Grid*

```
exception Out_of_range of string × float × (float × float)
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 →
          ?areas : Syntax.area list →
            (float, Bigarray.float64_elt,
             Bigarray.fortran_layout) Bigarray.Array2.t → t → t

  val smooth : float → Syntax.area → t → t

  val variance_area : Syntax.area → t → float

  val to_channel_2d : out_channel → t → unit

  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 variance : t → float

end
```

```
module Make (D : Division.T) =
  struct
    module D = D

    type t =
        { d1 : D.t;
          d2 : D.t;
          w : float array array;
          var : float array array;
          triangle : bool }

    let copy grid =
      { d1 = D.copy grid.d1;
        d2 = D.copy grid.d2;
        w = ThoMatrix.copy grid.w;
        var = ThoMatrix.copy grid.var;
        triangle = grid.triangle }

    let create ?(triangle = false) d1 d2 =
      let n1 = D.n_bins d1
      and n2 = D.n_bins d2 in
      { d1 = d1;
        d2 = d2;
        w = Array.make_matrix n1 n2 0.0;
        var = Array.make_matrix n1 n2 0.0;
        triangle = triangle }
```

We need

$$upper\, x] = lower\, [x \tag{97a}$$
$$upper\, x] = lower\, (x - 1 \tag{97b}$$
$$upper\, x) = lower\, [x - 1 \tag{97c}$$
$$upper\, x) = lower\, (x - 2 \tag{97d}$$

and

$$upper\, x] = upper\, x) + 1 \tag{98a}$$
$$lower\, [x = lower\, (x - 1 \tag{98b}$$

lower_bin had *Open* and *Closed* mixed up! (tho:2014-12-09)

```
let lower_bin div limit =
  try
    begin match limit with
    | Syntax.Closed x → D.find div x
    | Syntax.Open x → D.find div x + 1
    | Syntax.Bin n → n
    end
  with
  | Division.Below_min (_, _, n) → n
  | Division.Above_max (x, range, _) →
      raise (Out_of_range ("Grid.lower_bin", x, range))

let upper_bin div limit =
  try
    begin match limit with
    | Syntax.Closed x → D.find div x
    | Syntax.Open x → D.find div x - 1
    | Syntax.Bin n → n
    end
  with
  | Division.Above_max (_, _, n) → n
  | Division.Below_min (x, range, _) →
      raise (Out_of_range ("Grid.upper_bin", x, range))

let enclosed_bins div (x1, x2) =
  (lower_bin div x1, upper_bin div x2)

let enclosing_bin div = function
  | Syntax.Delta x → D.find div x
  | Syntax.Box n → n

let smooth width area grid =
  let gaussian = Filter.gaussian width in
  let w =
    begin match area with
    | Syntax.Rect (i1, i2) →
        let nx1, nx2 = enclosed_bins grid.d1 i1
        and ny1, ny2 = enclosed_bins grid.d2 i2 in
        Filter.apply12
          ~inf1:nx1 ~sup1:nx2 ~inf2:ny1 ~sup2:ny2
          gaussian gaussian grid.w
    | Syntax.Slice1 (i1, y) →
        let nx1, nx2 = enclosed_bins grid.d1 i1
```

```
        and ny = enclosing_bin grid.d2 y in
        Filter.apply1
            ~inf1 : nx1 ~sup1 : nx2 ~inf2 : ny ~sup2 : ny
            gaussian grid.w
    | Syntax.Slice2 (x, i2) →
        let nx = enclosing_bin grid.d1 x
        and ny1, ny2 = enclosed_bins grid.d2 i2 in
        Filter.apply2
          ~inf1 : nx ~sup1 : nx ~inf2 : ny1 ~sup2 : ny2
          gaussian grid.w
    end in
  { grid with w }

let to_channel_2d oc grid =
  for i = 0 to D.n_bins grid.d1 - 1 do
    Printf.fprintf oc "%g" grid.w.(i).(0);
    for j = 1 to D.n_bins grid.d2 - 1 do
      Printf.fprintf oc " %g" grid.w.(i).(j)
    done;
    Printf.fprintf oc "\n"
  done

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 later by the Fortran program interpreting the grid as a 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 n1 = D.find grid.d1 x'
  and n2 = D.find grid.d2 y' in
  grid.w.(n1).(n2) ← grid.w.(n1).(n2) +. f;
```

```
    grid.var.(n1).(n2) ← grid.var.(n1).(n2)
        +. 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 n1 = D.n_bins grid.d1
  and n2 = 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.make_matrix n1 n2 0.0;
    var = Array.make_matrix n1 n2 0.0;
    triangle = grid.triangle }

let normalize grid =
  let sum_w = ThoMatrix.sum_float grid.w in
  { d1 = D.copy grid.d1;
    d2 = D.copy grid.d2;
    w = ThoMatrix.map (fun w → w /. sum_w) grid.w;
    var = ThoMatrix.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_area area grid =
  let (nx1, nx2), (ny1, ny2) =
    begin match area with
    | Syntax.Rect (i1, i2) →
        (enclosed_bins grid.d1 i1, enclosed_bins grid.d2 i2)
    | Syntax.Slice1 (i1, y) →
        let ny = enclosing_bin grid.d2 y in
        (enclosed_bins grid.d1 i1, (ny, ny))
    | Syntax.Slice2 (x, i2) →
        let nx = enclosing_bin grid.d1 x in
        ((nx, nx), enclosed_bins grid.d2 i2)
    end in
  let n = float ((nx2 − nx1 + 1) × (ny2 − ny1 + 1)) in
  let w =
    ThoMatrix.sum_float
```

```
      ~inf1 : nx1 ~sup1 : nx2 ~inf2 : ny1 ~sup2 : ny2 grid.w /. n
  and w2 =
    ThoMatrix.fold_left
      ~inf1 : nx1 ~sup1 : nx2 ~inf2 : ny1 ~sup2 : ny2
      (fun acc w -> acc +. w *. w) 0.0 grid.w /. n in
  w2 -. w *. w

let variance grid =
  let n = float (D.n_bins grid.d1 * D.n_bins grid.d2) in
  let w = ThoMatrix.sum_float grid.w /. n
  and w2 =
    ThoMatrix.fold_left (fun acc w -> acc +. w *. w) 0.0 grid.w /. 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
```

Scan a bigarray. Assume a uniform weight, if it has only 2 columns.

```
let record_data data grid =
  let columns = Bigarray.Array2.dim1 data in
  if columns < 2 then
```

```
      eprintf "error: not enough columns"
  else
    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 =
        if columns > 2 then
          Bigarray.Array2.get data 3 i2
        else
          1.0 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
```

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
    ?areas data initial =

  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 data grid;
      let var = variance grid in
      begin match areas with
      | None | Some [] → ()
      | Some areas →
          let normalized_grid = normalize grid in
          let variances =
            List.map
              (fun area →
                 variance_area area normalized_grid) areas in
          let msg =
```

```
            "␣(" ^ Printf.sprintf "%g" (variance normalized_grid) ^ ":␣" ^
            String.concat ";␣"
              (List.map (fun x → Printf.sprintf "%g" x) variances) ^
            ")" in
          prerr_string msg;
          flush stderr
      end;
      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 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 }
```

NB: we need to transpose the weight matrix to get from our row major to `Fortran`'s column major array format expected by `circe2`!

```
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";
  ThoMatrix.iter
    (fun x → fprintf oc " %s\n" (Float.Double.to_string x))
    (ThoMatrix.transpose ch.g.w)

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 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_to_file name ?comments designs =
    let oc = open_out name in
    designs_to_channel oc ?comments designs;
    close_out oc

end
```

B.22 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*

Rescale the entries.

val *rescale* : *float* → *float* → *t* → *unit*

Utilities for reading ASCII representations.

val *lexer* : *char Stream.t* → *Genlex.token Stream.t*
val *next_float* : *Genlex.token Stream.t* → *float*

B.23 Implementation of *Events*

B.23.1 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
      let line = input_line ic in
      if line ≠ "" then begin
        reader array !i2 line;
        incr i2
      end
    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] 644_8 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] 644₈ 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'

let rescale scale1 scale2 data =
  for i2 = 1 to Array2.dim2 data do
    Array2.set data 1 i2 (Array2.get data 1 i2 /. scale1);
    Array2.set data 2 i2 (Array2.get data 2 i2 /. scale2)
  done
```

B.24 Interface of *Syntax*

```
exception Syntax_Error of string × int × int
```

B.25 Abstract Syntax and Default Values

```
val epsilon : float
```

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 boundary =
| Closed of float
| Open of float
| Bin of int
```

```
type point =
| Delta of float
| Box of int

type interval = boundary × boundary

type area =
| Rect of interval × interval
| Slice1 of interval × point
| Slice2 of point × interval

type channel =
    { pid1 : int;
      pol1 : int;
      pid2 : int;
      pol2 : int;
      lumi : float;
      bins1 : int;
      scale1 : float option;
      x1_min : float;
      x1_max : float;
      fixed_x1_min : bool;
      fixed_x1_max : bool;
      intervals1 : (int × Diffmaps.Default.t) list;
      bins2 : int;
      scale2 : float option;
      x2_min : float;
      x2_max : float;
      fixed_x2_min : bool;
      fixed_x2_max : bool;
      intervals2 : (int × Diffmaps.Default.t) list;
      smooth : (float × area) list;
      triangle : bool;
      iterations : int;
      events : string;
      histogram : string option;
      binary : bool;
      columns : int }
```

A parameter set is uniquely specified by PDG particle ids (*par abus de langage*), polarizations (now a floating point number for the effective polarization 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;
      design_scale1 : float option;
      design_scale2 : float option;
      channels : channel list;
      comments : string list }

val default_design : design

val default_channel : design → channel
```

One file can hold more than one grid.

```
type file = { name : string; designs : design list }

val default_file : file

type t = file list

type coord = X1 | X2 | X12
type side = Min | Max | Minmax

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
  | Scale of float × coord
  | Diffmap of (int × Diffmaps.Default.t) × coord
  | Smooth of float × area
  | Triangle of bool
  | Iterations of int
  | Events of string
  | Histogram of string
  | Binary of bool
  | Columns of int
  | Fix of bool × coord × side
```

```
type design_cmd =
  | Design of string
  | Roots of float
  | Design_Bins of int × coord
  | Design_Scale of float × coord
  | Channels of channel_cmd list
  | Comment of string

type file_cmd =
  | File of string
  | Designs of design_cmd list

type file_cmds = file_cmd list
```

B.26 Implementation of *Syntax*

```
exception Syntax_Error of string × int × int

let epsilon = 100. *. epsilon_float

type boundary =
| Closed of float
| Open of float
| Bin of int

type point =
| Delta of float
| Box of int

type interval = boundary × boundary

type area =
| Rect of interval × interval
| Slice1 of interval × point
| Slice2 of point × interval

type channel =
    { pid1 : int;
      pol1 : int;
      pid2 : int;
      pol2 : int;
      lumi : float;
      bins1 : int;
      scale1 : float option;
      x1_min : float;
```

```
    x1_max : float;
    fixed_x1_min : bool;
    fixed_x1_max : bool;
    intervals1 : (int × Diffmaps.Default.t) list;
    bins2 : int;
    scale2 : float option;
    x2_min : float;
    x2_max : float;
    fixed_x2_min : bool;
    fixed_x2_max : bool;
    intervals2 : (int × Diffmaps.Default.t) list;
    smooth : (float × area) list;
    triangle : bool;
    iterations : int;
    events : string;
    histogram : string option;
    binary : bool;
    columns : int }
type design =
  { design : string;
    roots : float;
    design_bins1 : int;
    design_bins2 : int;
    design_scale1 : float option;
    design_scale2 : float option;
    channels : channel list;
    comments : string list }
let default_design =
  { design = "TESLA";
    roots = 500.0;
    design_bins1 = 20;
    design_bins2 = 20;
    design_scale1 = None;
    design_scale2 = None;
    channels = [];
    comments = [] }
let default_channel design =
  { pid1 = 11 (* e⁻ *);
    pol1 = 0;
    pid2 = -11 (* e⁺ *);
```

```
    pol2 = 0;
    lumi = 0.0;
    bins1 = design.design_bins1;
    scale1 = design.design_scale1;
    x1_min = 0.0;
    x1_max = 1.0;
    fixed_x1_min = false;
    fixed_x1_max = false;
    intervals1 = [];
    bins2 = design.design_bins2;
    scale2 = design.design_scale2;
    x2_min = 0.0;
    x2_max = 1.0;
    fixed_x2_min = false;
    fixed_x2_max = false;
    intervals2 = [];
    smooth = [];
    triangle = false;
    iterations = 1000;
    events = "circe2.events";
    histogram = None;
    binary = false;
    columns = 3 }

type file = { name : string; designs : design list }

let default_file = { name = "circe2_tool.out"; designs = [] }

type t = file list

type coord = X1 | X2 | X12
type side = Min | Max | Minmax

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
  | Scale of float × coord
  | Diffmap of (int × Diffmaps.Default.t) × coord
  | Smooth of float × area
  | Triangle of bool
```

```
  | Iterations of int
  | Events of string
  | Histogram of string
  | Binary of bool
  | Columns of int
  | Fix of bool × coord × side

type design_cmd =
  | Design of string
  | Roots of float
  | Design_Bins of int × coord
  | Design_Scale of float × coord
  | Channels of channel_cmd list
  | Comment of string

type file_cmd =
  | File of string
  | Designs of design_cmd list

type file_cmds = file_cmd list
```

Module Lexer (Lex)

```
{
open Parser
let unquote s =
  String.sub s 1 (String.length s - 2)
}

let digit = ['0'-'9']
let upper = ['A'-'Z']
let lower = ['a'-'z']
let char = upper | lower
let white = [' ' '\t' '\n']

rule token = parse
    white { token lexbuf } (* skip blanks *)
  | '#' [^'\n']* '\n'
                { token lexbuf } (* skip comments *)
  | ['+''-']? digit+
    ( '.' digit* ( ['e''E'] digit+ )? | ['e''E'] digit+ )
                { FLOAT (float_of_string (Lexing.lexeme lexbuf)) }
  | ['+''-']? digit+
```

```
                { INT (int_of_string (Lexing.lexeme lexbuf)) }
| '"' [^'"']* '"'
                { STRING (unquote (Lexing.lexeme lexbuf)) }
| '/' { SLASH }
| '[' { LBRACKET }
| '(' { LPAREN }
| '<' { LANGLE }
| ',' { COMMA }
| ']' { RBRACKET }
| ')' { RPAREN }
| '>' { RANGLE }
| '{' { LBRACE }
| '}' { RBRACE }
| '=' { EQUALS }
| '*' { STAR }
| '+' { PLUS }
| '-' { MINUS }
| "ascii" { Ascii }
| "beta" { Beta }
| "binary" { Binary }
| "bins" { Bins }
| "center" { Center }
| "columns" { Columns }
| "comment" { Comment }
| "design" { Design }
| "electron" { Electron }
| "eta" { Eta }
| "events" { Events }
| "file" { File }
| "fix" { Fix }
| "free" { Free }
| "histogram" { Histogram }
| "id" { Id }
| "iterations" { Iterations }
| "lumi" { Lumi }
| "map" { Map }
| "max" { Max }
| "min" { Min }
| "notriangle" { Notriangle }
| "photon" { Photon }
| "gamma" { Photon }
```

```
  | "pid" { Pid }
  | "pol" { Pol }
  | "positron" { Positron }
  | "power" { Power }
  | "resonance" { Resonance }
  | "roots" { Roots }
  | "scale" { Scale }
  | "smooth" { Smooth }
  | "triangle" { Triangle }
  | "unpol" { Unpol }
  | "width" { Width }
  | eof { END }
```

Module Parser (Yacc)

Header

```
open Syntax
module Maps = Diffmaps.Default
let parse_error msg =
  raise (Syntax_Error (msg, symbol_start (), symbol_end ()))
```

Token declarations

```
%token < int > INT
%token < float > FLOAT
%token < string > STRING
%token SLASH EQUALS STAR PLUS MINUS
%token LBRACKET LPAREN LANGLE COMMA RBRACKET RPAREN RANGLE
%token LBRACE RBRACE
%token Ascii Binary
%token Beta Eta
%token Bins Scale
%token Center
%token Columns
```

%token *Comment*
%token *Design*
%token *Electron Positron Photon*
%token *Events Histogram File*
%token *Fix*
%token *Free*
%token *Id*
%token *Iterations*
%token *Lumi Roots*
%token *Map*
%token *Min Max*
%token *Notriangle*
%token *Pid*
%token *Pol Unpol*
%token *Power Resonance*
%token *Smooth*
%token *Triangle*
%token *Width*
%token *END*

%start *main*
%type < *Syntax.file_cmds list* > *main*

Grammar rules

main ::=
    files END { $1 }

files ::=
                                    { [] }
  | *file files* { $1 :: $2 }

file ::=
  | *LBRACE file_cmds RBRACE* { $2 }

file_cmds ::=
                                    { [] }
  | *file_cmd file_cmds* { $1 :: $2 }

```
file_cmd ::=
     File EQUALS STRING { Syntax.File $3 }
   | LBRACE design_cmds RBRACE { Syntax.Designs $2 }

design_cmds ::=
                                          { [] }
   | design_cmd design_cmds { $1 :: $2 }

design_cmd ::=
     Bins coord EQUALS INT { Syntax.Design_Bins ($4, $2) }
   | Scale coord EQUALS float { Syntax.Design_Scale ($4, $2) }
   | Design EQUALS STRING { Syntax.Design $3 }
   | Roots EQUALS float { Syntax.Roots $3 }
   | LBRACE channel_cmds RBRACE { Syntax.Channels $2 }
   | Comment EQUALS STRING { Syntax.Comment $3 }

channel_cmds ::=
                                          { [] }
   | channel_cmd channel_cmds { $1 :: $2 }

channel_cmd ::=
     Pid coord EQUALS particle { Syntax.Pid ($4, $2) }
   | Pol coord EQUALS polarization { Syntax.Pol ($4, $2) }
   | Fix coord EQUALS side { Syntax.Fix (true, $2, $4) }
   | Free coord EQUALS side { Syntax.Fix (false, $2, $4) }
   | Bins coord EQUALS INT { Syntax.Bins ($4, $2) }
   | Scale coord EQUALS float { Syntax.Scale ($4, $2) }
   | Min coord EQUALS float { Syntax.Xmin ($4, $2) }
   | Max coord EQUALS float { Syntax.Xmax ($4, $2) }
   | Map coord EQUALS map { Syntax.Diffmap ($4, $2) }
   | Lumi EQUALS float { Syntax.Lumi $3 }
   | Columns EQUALS INT { Syntax.Columns $3 }
   | Iterations EQUALS INT { Syntax.Iterations $3 }
   | Events EQUALS STRING { Syntax.Events $3 }
   | Histogram EQUALS STRING { Syntax.Histogram $3 }
   | Binary { Syntax.Binary true }
   | Ascii { Syntax.Binary false }
   | Smooth EQUALS float area { Syntax.Smooth ($3, $4) }
   | Triangle { Syntax.Triangle true }
```

```
  | Notriangle { Syntax.Triangle false }

particle ::=
     INT { $1 }
  | Electron { 11 }
  | Positron { -11 }
  | Photon { 22 }

polarization ::=
     INT { $1 }
  | Unpol { 0 }

coord ::=
                                          { Syntax.X12 }
  | SLASH STAR { Syntax.X12 }
  | SLASH INT {
      match $2 with
      | 1 → Syntax.X1
      | 2 → Syntax.X2
      | n →
          Printf.eprintf "circe2: ignoring dimension %d (not 1, 2, or *)\n" n;
          Syntax.X12 }

side ::=
     Min { Syntax.Min }
  | Max { Syntax.Max }
  | STAR { Syntax.Minmax }

map ::=
    Id LBRACE id RBRACE { $3 }
 | Power LBRACE power RBRACE { $3 }
 | Resonance LBRACE resonance RBRACE{ $3 }

area ::=
    interval interval { Syntax.Rect ($1, $2) }
 | interval point { Syntax.Slice1 ($1, $2) }
 | point interval { Syntax.Slice2 ($1, $2) }
```

```
point ::=
 | LBRACKET float RBRACKET { Syntax.Delta $2 }
 | LANGLE INT RANGLE { Syntax.Box $2 }

id ::=
    INT real_interval {
      let x_min, x_max = $2 in
      ($1, Maps.id x_min x_max) }

real_interval ::=
     left float COMMA float right { ($2, $4) }

left ::=
     LBRACKET { }
  | LPAREN { }

right ::=
     RBRACKET { }
  | RPAREN { }

interval ::=
     lower COMMA upper { ($1, $3) }

lower ::=
     LBRACKET float { Syntax.Closed $2 }
  | LPAREN float { Syntax.Open $2 }
  | LANGLE INT { Syntax.Bin $2 }

upper ::=
     float RBRACKET { Syntax.Closed $1 }
  | float RPAREN { Syntax.Open $1 }
  | INT RANGLE { Syntax.Bin $1 }

power ::=
    INT real_interval power_params {
      let x_min, x_max = $2
      and beta, eta = $3 in
      if beta ≤ -1.0 then begin
```

```
        Printf.eprintf "circe2: ignoring invalid beta: %g <= -1\n" beta;
        flush stderr;
        ($1, Maps.id x_min x_max)
      end else
        let alpha = 1.0 /. (1.0 +. beta) in
        ($1, Maps.power ~alpha ~eta x_min x_max) }

power_params ::=
    beta eta { ($1, $2) }
  | eta beta { ($2, $1) }

beta ::=
    Beta EQUALS float { $3 }

eta ::=
    Eta EQUALS float { $3 }

resonance ::=
   INT real_interval resonance_params {
     let x_min, x_max = $2
     and eta, a = $3 in
     ($1, Maps.resonance ~eta ~a x_min x_max) }

resonance_params ::=
    center width { ($1, $2) }
  | width center { ($2, $1) }

center ::=
    Center EQUALS float { $3 }

width ::=
    Width EQUALS float { $3 }

float ::=
    float_or_int { $1 }
  | float_or_int PLUS { $1 +. Syntax.epsilon }
  | float_or_int MINUS { $1 -. Syntax.epsilon }
```

float_or_int ::=
    INT { *float* $1 }
  | *FLOAT* { $1 }

B.27 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" } } }
```

exception *Invalid_interval* of *float* × *float*

type t
val *parse_file* : *string* → t
val *parse_string* : *string* → t
val *execute* : t → *unit*

B.28 Implementation of *Commands*

exception *Invalid_interval* of *float* × *float*

type t = *Syntax.t*

open *Printf*

```
module Maps = Diffmaps.Default
module Div = Division.Make_Poly (Maps)
module Grid = Grid.Make (Div)
```

B.28.1 Processing

```
let smooth_grid channel grid =
  List.fold_left
    (fun acc (width, area) → Grid.smooth width area acc)
    grid channel.Syntax.smooth

let report msg =
  prerr_string msg;
  flush stderr

let process_channel ch =
  report ("reading: " ^ ch.Syntax.events ^ " ...");
  let data =
    if ch.Syntax.binary then
      Events.of_binary_file ch.Syntax.columns ch.Syntax.events
    else
      Events.of_ascii_file ch.Syntax.columns ch.Syntax.events in
  report "done.\n";
  begin match ch.Syntax.scale1, ch.Syntax.scale2 with
  | None, None → ()
  | Some scale1, None → Events.rescale scale1 1.0 data
  | None, Some scale2 → Events.rescale 1.0 scale2 data
  | Some scale1, Some scale2 → Events.rescale scale1 scale2 data
  end;
  let initial_grid =
    Grid.create ~triangle : ch.Syntax.triangle
      (Div.create ch.Syntax.intervals1 ch.Syntax.bins1
         ch.Syntax.x1_min ch.Syntax.x1_max)
      (Div.create ch.Syntax.intervals2 ch.Syntax.bins2
         ch.Syntax.x2_min ch.Syntax.x2_max) in
  let grid =
    Grid.of_bigarray ~verbose :true
      ~iterations : ch.Syntax.iterations
      ~fixed_x1_min : ch.Syntax.fixed_x1_min
      ~fixed_x1_max : ch.Syntax.fixed_x1_max
      ~fixed_x2_min : ch.Syntax.fixed_x2_min
```

```
      ~fixed_x2_max : ch.Syntax.fixed_x2_max
      ~areas : (List.map snd ch.Syntax.smooth)
      data initial_grid in
  let smoothed_grid = smooth_grid ch grid in
  begin match ch.Syntax.histogram with
  | Some name →
      let oc = open_out name in
      Grid.to_channel_2d oc smoothed_grid;
```

B.29 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
```

B.30 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.make n_bins 0;
    w = Array.make n_bins 0.0;
    w2 = Array.make 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
```

B.31 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

$$f(a,b) = \sum_i w_i(ax_i + b - y_i)^2 = \langle (ax+b-y)^2 \rangle \tag{99}$$

i. e.

$$\frac{1}{2}\frac{\partial f}{\partial a}(a,b) = \langle x(ax+b-y)\rangle = a\langle x^2\rangle + b\langle x\rangle - \langle xy\rangle = 0 \tag{100a}$$

$$\frac{1}{2}\frac{\partial f}{\partial b}(a,b) = \langle ax+b-y\rangle = a\langle x\rangle + b - \langle y\rangle = 0 \tag{100b}$$

and

$$a = \frac{\langle xy\rangle - \langle x\rangle\langle y\rangle}{\langle x^2\rangle - \langle x\rangle^2} \tag{101a}$$

$$b = \langle y\rangle - a\langle x\rangle \tag{101b}$$

```
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
```

B.32 Implementation of *Circe2_tool*

B.32.1 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)
```

B.32.2 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
```

B.32.3 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
```

B.32.4 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
```

B.32.5 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
```

B.32.6 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)
```

B.32.7 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
  | Unit_Tests
  | Log10
  | Exp10
  | Power of format × string

let rec passed = function
  | [] → true
  | (OUnit.RFailure _ | OUnit.RError _ | OUnit.RTodo _) :: _ → false
  | (OUnit.RSuccess _ | OUnit.RSkip _) :: tail → passed tail

let _ =
  let usage = "usage: " ^ Sys.argv.(0) ^ " [options]" in
  let nbins = ref 100
  and triangle = ref false
  and shrink = ref false
  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 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), "");
```

```
    ("-b", Arg.Int (fun n -> nbins := n), "#bins");
    ("-s", Arg.Set shrink, "shrinkwrap interval");
    ("-S", Arg.Clear shrink, "don't shrinkwrap interval [default]");
    ("-t", Arg.Set triangle,
     "project symmetrical distribution onto triangle");
    ("-v", Arg.Set verbose, "verbose");
    ("-test", Arg.Unit (fun () -> action := Unit_Tests),
     "run unit test suite");
    ("-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
  | Unit_Tests →
      let suite =
        OUnit.(>:::) "All"
          [ThoArray.suite;
           ThoMatrix.suite;
           Filter.suite] in
      if passed (OUnit.run_test_tt ~verbose :!verbose suite) then
        exit 0
      else
        exit 1
with
| Syntax.Syntax_Error (msg, _, _) →
    Printf.eprintf "%s: parse error: %s\n" Sys.argv.(0) msg;
    exit 1
end;
exit 0
```

Identifiers

`generate_beta`: 74a, 74b, 81d

Refinements

⟨(i1, i2) ← i 51a⟩
⟨ib ← i 51b⟩
⟨x ∈ [x(ib − 1), x(ib)] 56c⟩
⟨y) ← yy 61c⟩
⟨$y \leftarrow (a(x-\xi))^{\alpha}/b + \eta$ 56d⟩
⟨circe2.f90 49c⟩
⟨circe2 *declarations* 50a⟩
⟨circe2 *implementation* 54d⟩
⟨circe2 *parameters* 55d⟩
⟨circe2_channel *members* 50e⟩
⟨circe2_division *members* 52b⟩

⟨circe2_generate.f90 70b⟩
⟨circe2_ls.f90 72⟩
⟨circe2_moments.f90 77b⟩
⟨circe2_moments_library *declarations* 74a⟩
⟨circe2_moments_library *implementation* 74b⟩
⟨circe2_state *members* 50d⟩
⟨implicit none 49b⟩
⟨sampling *declarations* 77c⟩
⟨sampling *implementation* 77e⟩
⟨tao_random_objects.f90 69b⟩
⟨tao_random_objects *declarations* 69c⟩
⟨tao_random_objects *implementation* 69d⟩
⟨*Apply Jacobian for triangle map* 62a⟩
⟨*Calculate* y 67a⟩
⟨*Check* a *and* b 74c⟩
⟨*Check for* ECRIC2 68b⟩
⟨*Check if* design *and* fdesgn *do* match 64a⟩
⟨*Check polarization support* 65e⟩
⟨*Complain and* return *iff* ic ≤ 0 56a⟩
⟨*Copyleft notice* 49e⟩
⟨*Decode polarization support* 65b⟩
⟨*Do a binary search for* wgt(i − 1) $\leq$ u $<$ wgt(i) 56b⟩
⟨*Do a binary search for* y(ib − 1) $\leq$ y $<$ y(ib) 62b⟩
⟨*Error codes for* circe2_load 63c⟩
⟨*Find* ic *for* p *and* h 55e⟩
⟨*Find free logical unit for* lun 68c⟩
⟨*Generate a trial* x *and calculate its weight* w 75b⟩
⟨*Inverse triangle map* 57c⟩
⟨*Load channel* ch 65d⟩
⟨*Load divisions* x 66b⟩
⟨*Load histograms* 64b⟩
⟨*Load weights* wgt *and* val 67b⟩
⟨*Local variables in* circe2_load 65c⟩
⟨*Main program* 89⟩
⟨*Module procedures for* circe2_generate_program 71e⟩
⟨*Open* name *for reading on* lun 67d⟩
⟨*Process command line arguments for* circe2_generate_program 70c⟩
⟨*Select* n *according to the weight* channels(n)%w 82a⟩
⟨*Separator* 49d⟩
⟨*Set up* generate_beta *parameters* 75a⟩
⟨*Set up best value for t* 77a⟩

⟨*Skip comments until* `CIRCE2` 67f⟩
⟨*Skip data until* `ECRIC2` 68a⟩
⟨*Swap* `y(1)` *and* `y(2)` *in 50% of the cases* 58a⟩
⟨*Test support for density matrices* 63a⟩
⟨*The old* `ieee_is_normal` *kludge* 86a⟩
⟨*Version* 49a⟩
⟨*Write channel data* 73c⟩
⟨*Write channel header* 73b⟩
⟨*Write design/beam data* 73a⟩

Index