Foam: A General Purpose Cellular Monte Carlo Event Generator[†]

S. Jadach[†]

Institute of Nuclear Physics, ul. Kawiory 26a, Kraków, Poland
and
CERN Theory Division, CH-1211 Geneva 23, Switzerland

Abstract

A general purpose, self-adapting, Monte Carlo (MC) event generator (simulator) is described. The high efficiency of the MC, that is small maximum weight or variance of the MC weight is achieved by means of dividing the integration domain into small cells. The cells can be n-dimensional simplices, hyperrectangles or Cartesian product of them. The grid of cells, called "foam", is produced in the process of the binary split of the cells. The choice of the next cell to be divided and the position/direction of the division hyper-plane is driven by the algorithm which optimizes the ratio of the maximum weight to the average weight or (optionally) the total variance. The algorithm is able to deal, in principle, with an arbitrary pattern of the singularities in the distribution. As any MC generator, it can also be used for the MC integration. With the typical personal computer CPU, the program is able to perform adaptive integration/simulation at relatively small number of dimensions (≤ 16). With the continuing progress in the CPU power, this limit will get inevitably shifted to ever higher dimensions. Foam is aimed (and already tested) as a component in the MC event generators for the high energy physics experiments. A few simple examples of the related applications are presented. Foam is written in fully object-oriented style, in the C++ language. Two other versions with a slightly limited functionality, are available in the Fortran77 language. The source codes are available from http://jadach.home.cern.ch/jadach/.

Submitted to Comput. Phys. Commun.

Contents

PROGRAM SUMMARY

Title of the program: `Foam`, version 2.05.
Computer: any computer with the C++ or Fortran 77 compilers and the UNIX operating system
Operating system: UNIX, program was tested under Linux 6.x.
Programming languages used: ANSI C++ and FORTRAN 77 with popular extensions such as long names, etc.
High-speed storage required: < 50 MB
No. of lines in combined program and test deck: 4235 lines of C++ code and 9826 lines of F77 code.
Keywords: Monte Carlo (MC) simulation and generation, particle physics, phase space.
Nature of the physical problem: Monte Carlo simulation or generation of unweighted (weight equal one) events is a standard problem in many areas of research. It is highly desirable to have in the program library a general-purpose numerical tool (program) with a MC generation algorithm featuring built-in capability of adjusting automatically the generation procedure to an arbitrary pattern of singularities in the probability distribution.
Method of solution: In the algorithm a grid (foam) of cells is built in the process of the binary split of the cells. The resulting foam is adapted automatically to the shape of the integrand in such a way that the resulting ratio of average weight to maximum weight or variance to average weight is arbitrarily good. The above algorithm a substantial improvement of the previous version in Ref. [1]. The division of the cell is improved and, in addition to simplical cells, a hyperrectangular cell shape is also available.
Restrictions on the complexity of the problem: The program is memory-hungry and therefore presently limited to relatively small dimensions ≤ 16. (In `Foam` 1.x of Ref. [1] the dimension was limited to $n \leq 6$.)
Typical running time: The CPU time necessary to build up a foam of cells depends strongly on the number of dimensions and the requested size of the grid. On the PC with the 550MHz Intel chip it takes about 30 seconds to build a hyperrectangular grid of 10000 cells for a simple 3-dimensional distribution.
[1] S. Jadach, Comput. Phys. Commun. **130**, 244 (2000).

1 Introduction

This work describe a new version of an algorithm for producing random points according to an arbitrary, user defined, distribution in the n-dimensional space – much improved with respect to the original version of Ref. [1]. A new implementation is realized in the C++ programming language in a fully object-oriented manner[1]. Since the changes both in the algorithm and in the implementation with respect to Ref. [1] are quite essential, a complete description of the method and the new code is provided, instead of only an update with respect to Ref. [1].

For the problem of function minimization, integration (quadrature) there are plenty of *general purpose* programs which can be applied to an arbitrary user-defined function. "general-purpose" means that all these tools work, in principle, for a very wide range of user-functions. For multi-dimensional Monte Carlo simulation problem, that is for the problem of generating randomly points according to a given n-dimensional distribution, there is precious little examples of the *General Purpose Monte Carlo Simulators* (GPMCS), that is programs which work (in principle) for arbitrary distribution [3–6], In this work we are concerned mainly with the MC applications of the high energy physics An example of the work on GPMCS applied in other fields see an interesting works[2] of Refs. [7–9].

GPMCS is essentially a random number generator which generates points in multi-dimensional space with non-uniform user-define probability distribution. Two essential reasons for scarcity of GPMCS's are the lack of novel ideas about an efficient algorithm and the need of much CPU power and memory – only recently available or affordable.

Inevitably the GPMCS works in two stages: *exploration* and *generation*[3]. During an *exploration* phase the GPMCS is "digesting" the entire shape of the n-dimensional distribution $\rho(\vec{x})$ to be generated, memorizing its shape as efficiently as possible, using all available CPU processing power and memory[4]. In `Foam`, the exploration phase is the phase of the *build-up of the system of cells* covering entirely the integration space, which will be called "foam", produced in the process of the binary split of the cells. In the *generation* phase, GPMCS provides a method of the MC generation of the points $\vec{x}$ *exactly* according to $\rho(\vec{x})$. The vector $x = \vec{x} = (x_1, x_2, ...x_n)$ will also be called in the following a *Monte Carlo event*. In `Foam`, the MC generation is very simple: a cell is chosen randomly, and next, a point is generated within the cell with uniform distribution, see below for more details. The value of the integrand is already estimated in the exploration; it can be calculated with an arbitrary precision in the generation phase.

During the exploration `Foam` constructs a distribution $\rho'(x)$, which is uniform within

---

[1]The early C++ version of the Foam was coded by M. Ciesla and M. Slusarczyk [2]. It was a translation of a version 1.x from Fortran77 to C++. The analogous translation to JAVA language was also done.

[2]In these works there is far more emphasis on the parallel computing aspects of the integration (not simulation) than on the cell geometry, as compared with our work.

[3]Exploration and generation could be done simultaneously, at the expense of complications in the algorithm and the code.

[4]The procedure of memorizing multidimensional distribution $\rho(\vec{x}) \geq 0$ is a kind of interpolation, in which the grid of cells is denser in places where the distribution peaks and/or varies strongly.

each cell, and is used for the MC generation. Events are weighted with the weight $w = \rho/\rho'$. The quality of the distribution of this weight, measured in terms of the weight *distribution parameters* like small variance, good ratio of maximum to average, is determined by the quality of the exploration. The basic principle of the Foam algorithm is that the parameters of the anticipated "target weight distribution" in the MC generation phase are used as a driving force guiding the cell build-up (exploration). In the case of a successful exploration, weighted MC events can be turned efficiently into unweighted ones with the usual rejection method, that is with a small rejection rate.

Since the exploration phase may be CPU-time consuming, it is a natural to expect that GPMCS has a built-in mechanism of *persistency*, that is, there is a mechanism of writing into a mass-storage (computer disk) the whole information on the memorized shape of the distribution obtained from the exploration phase, such that the generation of the MC events can be (re)started at any later time, without any need of repeating the time consuming exploration. One small step further is to require that the generation of events with GPMCS can be stopped at any time, the entire status of the GPMCS can be written on the disk, and the generation of the next event can be resumed at any later time upon reading the stored information; the next generated event will be such as if there was no any break in the generation process. In fact, this is what we shall really mean in the following as a *persistency* mechanism for GPMCS, and what is actually implemented in the Foam. In Foam the persistency is realized using ROOT[5] package [10].

The GPMCS programs will be always limited to "small dimensions". With presently available computers "small" means in practice $n \leq 10$, up to $n \leq 16$ for a "mildly" singular distributions. This is already quite satisfactory, especially if we remember that this limit will pushed higher, as the available hardware gets more powerful, without any need of modifying the existing code[6]. – twelve years from now, with 100GHz processor and 1TByte disk portable computers the same version of Foam will work for even higher dimensions.

Foam has been developed having in mind that it will be used as a part of a bigger MC program; typically, to generate a subset of variables in which a model distribution is the most singular (has strong peaks). This is why we are not so much concerned by the fact the the cellular Foam algorithm is inefficient for, say, 150 variables. The user is supposed to select $n \leq 16$ "wild variables" [4] and apply Foam to them. For the remaining "mild variables" Foam may merely serve a role of a uniform random number generator, if the user of Foam wants to exploit that option. On the other hand, for smaller MC problems, Foam may play a role of a "standalone MC generator" or "standalone MC integrator". Also, from the following description of the various modes of the use of Foam it will be clear that the subprogram providing the model distribution to Foam can have a quite complicated structure. Nevertheless, this user-provided part of the program will be smaller as compared to a solution without Foam, because Foam provides for essential

---

[5]The use of ROOT is optional in Foam. However, version of Foam without ROOT does not feature any kind of persistency.

[6]The present implementation of Foam is fully based on the dynamic allocation of the memory and the space dimension is a user defined parameter.

functionalities concerning weight optimisation. This remark is especially true for the case of implementation of the multibranching with help of Foam.

Let us also note that the two-dimensional cellular MC sampler VESKO2 with the primitive binary split was already included in the program LESKO-F of Ref. [11] long time ago.

It is worth to mention that `Foam` is not based on the "principle of factorizability" of the integrand distribution, $\rho(\vec{x}) = \prod_1^n \rho_i(x_i)$, on which `VEGAS`-family programs are built [3–5].

The outline of the paper is the following: In Section 2 we describe the cellular `Foam` algorithm, delegating the description of the cell division procedure to Section 3. Section 4 is devoted to description of the `Foam` code in C++. Usage of the `Foam` is described in Section 5 and examples of the numerical results (MC efficiency) are given in Section 6. Conclusions and Appendix on the variance minimization finalize the paper.

Buil–up of the foam of cells

Split root cell if necessary

Choose next cell for the split

MC exploration of the cell

Generate series of MC events inside cell

Choose best direction (division edge)

Find out best division ratio (division plane)

Generate MC event

Choose randomly a cell
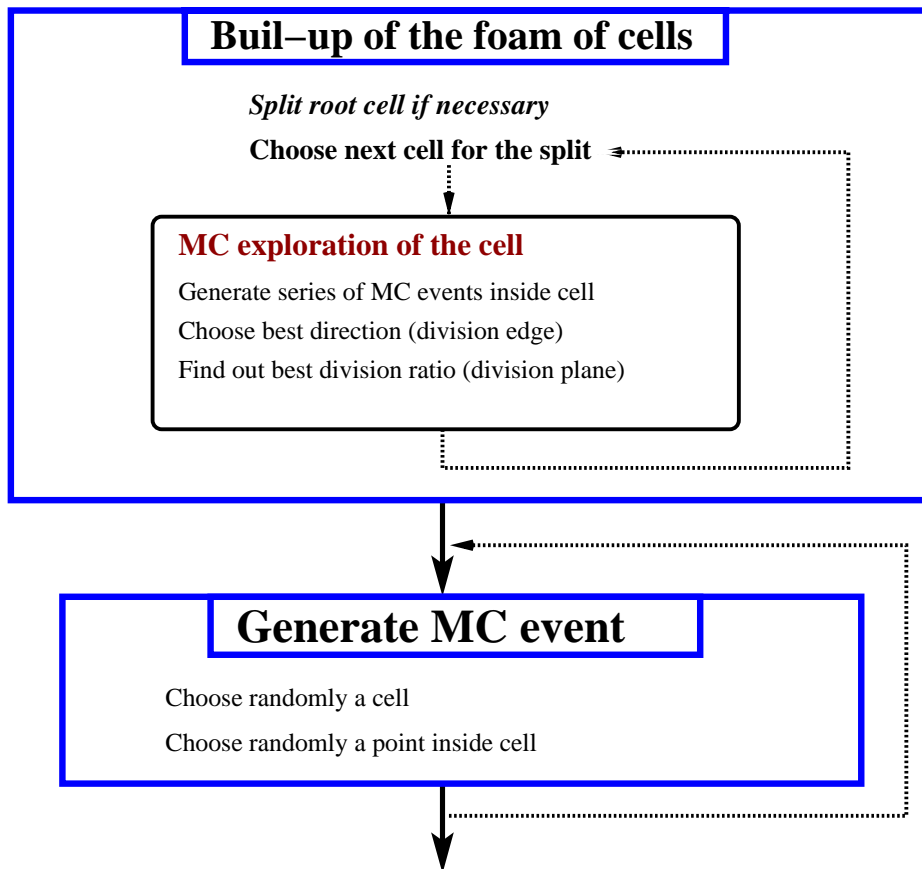
Choose randomly a point inside cell

Figure 1: Two stages in the cellular algorithm of the Foam.

2 The Foam algorithm

As already mentioned, the execution of the `Foam` algorithm is clearly separated into the first stage of the "distribution exploration" consisting of the "build-up of the foam of the cell", which in a sense memorizes the n-dimensional shape of the distribution, and the second stage of the actual "MC generation", see Fig. 1. The most essential part of the present `Foam` algorithm is the procedure of the binary split of the cell, in which it is decided which cell is picked up for the next split and the necessary parameters of the geometry of the cell split are determined. This part of the `Foam` algorithm description is delegated to the next Section. In the present section we describe other, more general, aspects of the `Foam` algorithm.

2.1 Cellular exploration of the distribution

The most obvious method to minimize the variance or maximum weight of the target weight distribution in generation, proposed already some 40 years ago, is to split the integration domain into many cells, such that the distribution $\rho(\vec{x})$ is approximated by $\rho'(\vec{x})$, which is constant within each cell[7]. This is a *cellular class* of the general-purpose MC algorithms[8].

The immediate questions are: what kind or shapes of the cells to use and how to cover the integration domain with cells? The reader may find in Ref. [6] an example of rather general discussion of these questions. In the `Foam` program the user may opt for one of the three geometries of the cells: (1) simplices, (2) hyperrectangles and (3) Cartesian products of simplices and hyperrectangles. For these particular types of cells there exist an efficient method of parametrizing them in the computer memory and handling their geometry.

The system of many cells can be created and reorganized all at once, like in `VEGAS`-type programs [3–5], or in a more evolutionary way, like the cell split process of this work. In the `Foam` algorithm we rely on the *binary split* of cells. Starting from the entire integration domain (unit hyperrectangle or simplex) cells are split into two daughter cells, step by step, until the user-defined memory limit is reached. A choice of a next cell to be split and the geometry of the split in the exploration phase is driven by the "target weight distribution" of the generation process, see Section 3. The important advantage of any cell split algorithm is that it assures automatically the *full coverage* of the integration domain – simply because the primary *root cell* is identical with the entire integration domain and the two daughter cells always cover entirely the parent cell. The problem of *blind spots* discussed in Ref. [6] is avoided by construction.

In the early version of the `Foam` of Ref. [1], there was a possibility in the algorithm that "unsuccessful" branch in the tree of all cells can be erased and rebuild. This was called

---

[7]For the MC simulation, our main aim, more sophisticated interpolation of $\rho(\vec{x})$ within a cell does not seem to be worth an effort – it would be interesting if our main aim was the integration of $\rho(\vec{x})$.

[8]The term "stratified sampling", used in the literature, has in my opinion a narrower meaning than "cellular class".

"collapse" and "rebuild". In the present version this option was removed[9], because the experience with many testing functions has shown that the algorithm of the cell build-up is rather "deterministic" and the "rebuild" procedure was usually leading to a new branch of foam with about the same features, as the old one.

Let us finally remark, that the version of the cellular algorithm presented in this paper is, in fact, a result of many experiments in the constructing different variants of the algorithm. The presented version is the best one out of several development versions. In the code one may still see some "hooks" and unused features (class members or methods) related to these alternative variants. We have left them just in case if some new idea of improving the algorithm emerges, or for certain kinds of debugging/testing.

2.2   Variance reduction versus maximum weight reduction

In the construction of the `Foam` algorithm most effort was invested into a minimization of the ratio of the maximum weight to the average weight $w_{max}/\langle w\rangle$. This parameter is essential, if we want to transform variable-weight events into $w = 1$ events, at the latter stage of the MC generation[10].

Minimizing the maximum weight w_{max} is not the same as minimizing the variance $\sigma = \sqrt{\langle w^2\rangle - \langle w\rangle^2}$. Usually, minimizing w_{max} is more difficult – but it is worth an effort because `Foam` is really meant to be a part of a bigger MC program, where it is usually essential that the "inner part" of the program provides events with an excellent weight distribution, or even $w = 1$ events. Nevertheless, minimizing the variance is also implemented in Foam and available optionally. It can be useful if one is satisfied with the variable-weight events, and/or if the main aim is evaluation of the integral and not the MC simulation.

The difference between the above two options is well illustrated in Fig. 2, which shows two examples of the evolution of the MC weight distribution due to gradual increase of the number of cells. For the default configuration, `Foam` is optimising the ratio $w_{max}/\langle w\rangle$. This case is shown in plots (a-c) in Fig. 2. Here, the weight distribution features sharper and sharper drop of the weight distribution at $w = 1$, with the increasing number of cells. Also, the average weight increases gradually and the weight distribution gets narrower. The optional case of the optimization of $\sigma/\langle w\rangle$ is shown in plots (d-f) of Fig. 2. In this case the variance is decreasing with the growing numbers of the cells. On the other hand, the maximum weight is much higher than before. All weight distributions were obtained for the same 2-dimensional testing function $\rho_b(x)$, used also in Section 6.

2.3   Hyperrectangles or simplices?

In Ref. [1] simplical cells have been chosen instead of simpler hyperrectangles, mainly because of the author's "prejudice" that simplices may adapt more efficiently to complicated

---

[9]A "flush method" which erases the entire foam of cells from the computer memory and allows for its reinitialization is, however, available.

[10]We provide optionaly in the `Foam` for the rejection leading to $w = 1$ events.
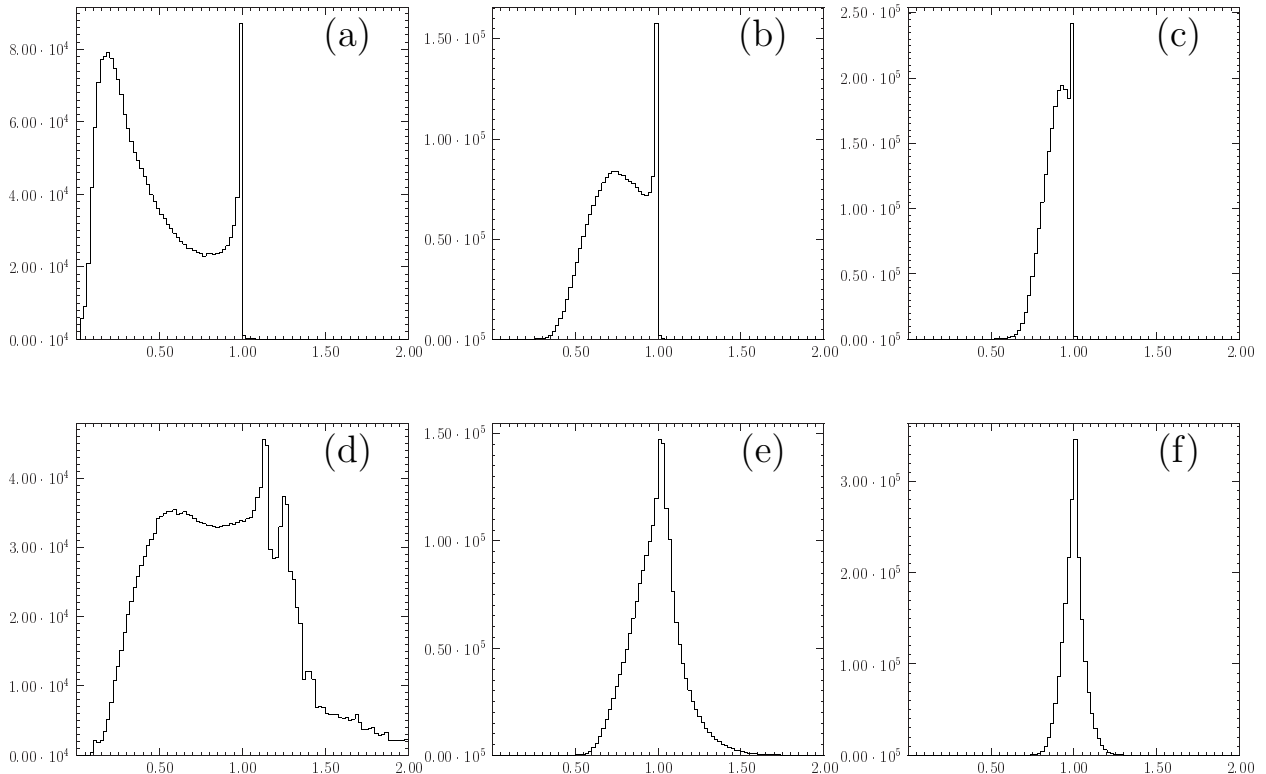
Figure 2: Weight distribution of the `Foam` for the default option with the maximum weight optimization (a-c) compared to analogous distributions obtained for an option with the variance optimization (d-f). Number of cells is 200, 2000 and 20000 for (a-c) and (d-f), correspondingly.

singularities in the distribution $\rho(x)$ spanned along subspaces, not necessarily parallel to axes of the global reference frame. Hyperrectangles tend to remember orientation of the parent hyperrectangle, while simplices feature, in principle, a kind of "angular mobility", i.e. they may forget orientation of grand-grand-parents, and adapt to orientation of the singularity in $\rho(x)$. An experience with tens of testing functions has shown that in many cases hyperrectangles provide the same or even better final MC efficiency than simplices, for the same number of cells. Moreover, simplices have certain additional disadvantages. Presently, `Foam` with simplices is practically limited to rather low dimensions $n \leq 5$, because in most cases the starting integration domain is a unit hyperrectangle, which has to be divided into $n!$ simplices, where $n!$ becomes quickly a large number[11]. This limitation is, of course, not valid, if the integration domain is actually a simplex of the high dimensionality instead of hyperrectangle. (`Foam` can be configured to start cell evolution from a simplex or Cartesian product of a simplex and a hyperrectangle.) Furthermore, geometry manipulations in the simplical case require calculation of many determinants – this slows

---

[11]Mapping of the hyperrectangle into simplex is possible, but it usually introduces nasty singularities in $\rho(x)$ located at the vertices, edges and walls of the simplex.

9

down the program execution at higher dimensions. In addition, in the present implementation, the memory consumption in a simplical foam build-up is $\sim 16 \times n$ Bytes/Cell, while for hyperrectangles we have found a method which limits memory consumption to below ~ 80 Bytes/Cell *independently* of n, see Section 2.6. We can therefore reach easily the level 10^6 hyperrectangular cells at any dimension (in practice $n \leq 16$) and about 50000 simplical cells, for $n \leq 5$. As we see, hyperrectangular foam seems to win on many fronts. Nevertheless, we keep simplical foam as an option, because in certain application one encounters distributions for which it turns out to be more efficient to use simplices, in spite of all their limitations, at least for a subset of the integration variables.

2.4   Build up of the foam and data organization

The foam of cell is built-up starting form the *root cell*, which is the entire integration domain, through process of binary split of a parent cell into two daughter cell. The root cell is either a unit hypercube $0 \leq x_i \leq 1$ (default) or a simplex $0 \leq x_1 \leq x_2 \leq x_3 \leq \cdots \leq x_n \leq 1$. Also a Cartesian product of these two shapes is optionally available. Any cell being a product of the cell split can be also a hyperrectangle, a simplex or Cartesian product of the k-dimensional hyperrectangle and n-dimensional simplex, with the total dimensionality $k + n$. If the starting root cell is a hypercube and cells are simplical (or mixed type) then root cell is immediately divided into $n!$ simplical (or mixed type) cells.

Each cell is *explored* immediately after its creation. In the exploration of the cell about $100 - 1000$ MC events (user may reset this number) are generated inside the cell with flat (uniform) distribution and using MC weight equal $\rho(x)$; certain averages and certain integrals over the cell are estimated. Also, the best geometry of the binary split of the cell is established and recorded for the future use. In this way, every created cell is ready for an immediate split. The determination of the best split is described in a fine detail in Section 3. In the exploration the estimate of the integral $R_I = \int_{\omega_I} \rho(x) dx^n$ is calculated for each cell ω_I. Far more important is another functional $R_{loss}|_I = \int_{\omega_I} \rho_{loss}(x) dx^n$, see Section 3 for its definition, which determines the evolution of the foam and the split of the cell. Next cell to be divided into two is a cell chosen randomly, according to probability proportional to $R_{loss}|_I$ or, optionally, a cell with the biggest $R_{loss}|_I$.

The process of the division of the cells continues until the user defined maximum number N_c of the cells is reached. N_c includes also all cells which has been split, that is all parent and grand-grand-parent cells, which we shall call *inactive* cells contrary to normal ones called *active*. Usually, when we refer to a cells, we mean both active and inactive ones. Keeping inactive cells in the record may look like a waste of the memory, but due to the binary character of the cell split, the loss is only a mere factor of two and it is profitable to keep all inactive cells (including the root cell) for many reasons, in particular, as we shall see in Section 2.6, keeping all cells in the record will help us to encode cells in memory in an economic way, such that at higher dimensions we finally gain in terms of total consumption of a memory. Furthermore, for certain quantities which are the integrals over the cell like R_I we do the following: just after the split, when a new more precise value of R_I is known for the daughter cells – the value of the R_I of the

parent cell is updated with the sum of the contributions from two daughter cells. This correcting procedure is repeated for all grand-parent cell up to the root cell. In this way, the root cell (and any other inactive cell) always keeps track of the actual value of the total R_I during the whole foam build-up process. This can be done for any other integral quantity as well, and can be exploited for various purposes.

Since maximum number of the cells N_c is defined in the beginning of the foam build-up, all the cell objects and/or other related objects (vertices) are allocated in the computer memory at once, in the very beginning of the cell build-up. On the other hand, the cell objects are organized as multiply linked list, with pointers pointing to parents and daughters. In addition, an array of pointers to all active cells is created at the end of the foam build-up.

Let us now explain briefly how the geometry of an individual cell is parametrized and stored in the memory. It is relatively easy to parametrize n-dimensional hyperrectangle or simplex in a way which does not require much computer memory. An n-dimensional simplex is fully determined by its $n+1$ vertices. Since most of vertices are common to two or more adjacent simplices, the most efficient method is to build an array of *all* vertices $\vec{V}_K, K = 1, 2, ..., N_V$, each of them being n-component vector and to define every simplex as a list $n+1$ vertex indices (integers or pointers) $K_1, K_2, ..., K_{n+1}$. For N_c simplical cells resulting from the binary split of a single "root" simplex cell the number of vertices is $n + 1 + N_c$, because each binary split adds one new vertex. (We include in N_c also cells which has got split). The interior points of the simplex are parametrized as follows

$$\vec{x} = \sum_{i \neq p}^{n} \lambda_i (\vec{V}_{K_i} - \vec{V}_{K_p}), \quad \lambda_i > 0, \quad \sum_{i \neq p} \lambda_i < 1, \quad i = 1, 2, \ldots, n, \tag{1}$$

using basis vectors relative to the p-th vertex. The above method would be inefficient for n-dimensional hyperrectangles, because memorizing all 2^n vertices would require too much memory at higher dimensions. Instead, we use another way of parametrization: each hyperrectangle is defined by the n-dimensional vector $\vec{q}$ defining the origin of the cell and another vector $\vec{h} = (h_1, h_2, ..., h_n)$, where each component h_i is the length of the hyperrectangle along the i-th direction. This is even clearer from the explicit parametrization of the interior of the hyperrectangle:

$$x_i = q_i + \lambda_i h_i, \quad 0 < \lambda_i < 1 \quad i = 1, 2, \ldots, k. \tag{2}$$

For cells with mixed topology, we apply eq. (2) for $i = 1, 2, \ldots, k$ and eq. (1) for $i = k + 1, k + 2, \ldots, k + n$. In Section 2.6 we describe an optional method of storing hyperrectangular cell, in which just two integer numbers are recorded instead of two vectors $\vec{q}$ and h (two of 2-Byte integers instead of $2n$ of 8-Byte floating-point numbers). This method is implemented for hyperrectangular part of the space only.

2.5   Monte Carlo generation

Once the build-up of the cells is finished, the Monte Carlo generation takes place. There is no need for any reorganization of the cells. MC generation can be started immediately.

The only one thing done at the very end of the foam build-up is preparation of the list of pointers to active cells and the array of the corresponding R'_I.

The MC point is generated in two steps. First, a cell is chosen with a probability proportional to $R'_I = \int_{x \in Cell_I} \rho'(x)$ and next a MC point x is chosen with the uniform probability inside the cell. The MC weight $w = \rho'(x)/\rho(x)$ is associated with the event. For a successful foam of the cells the MC weight is close to one and the user may turn weighted events into $w = 1$ event by means of rejection method (with the acceptance rate $\sim \langle w \rangle / w_{max}$). Foam can do this for the user. However, the user can sometimes organize better the calculation of the $\langle w \rangle$ and bookkeeping of other parameters of the weight, in a way which fits the best his own aims. This is why the mode of variable weights MC events is also available. The total integral, usually necessary for the proper normalization of the MC sample is calculated using $R = R' \langle w \rangle$. Foam provides both, the exact value of the R' and the MC estimate of the integral R.

2.6 Economic use of the computer memory

The actual implementation of the single cell object occupies about 80 Bytes (it could be reduced to about 40 Bytes if necessary) of various integer and double precision attributes, plus the dimension-dependent part. In the case of a simplical cell, each new cell adds one n-component double-precision vector (vertex) and the total memory consumption is therefore $(80 + 8 \times n)$ Bytes/cell. For $n = 5$ and 100k cells it is therefore $\sim$15MB of the memory, still an affordable amount. For the hyperrectangle cells we have to count two n-component double-precision vectors per cell, that is $(100 + 16 \times n)$ Bytes/cell. For the 10^6 cells and $n = 15$ that would mean $\sim$340MB for the entire foam of cells and this could be annoying. Fortunately, we have found a method of reducing substantially the memory consumption for a hyperrectangular foam. As discussed in the Section 3 the geometry of the division of the cell is fully determined in terms of two integers, one of them is the index of an edge to which the division plane is perpendicular and another one defines the position of a division plane. The position parameter is a rational number, and only the integer numerator has to be remembered, while the denominator is common to all cells. The above two integers define uniquely the position of the two daughter cells relative to a parent cell. With this method the memory consumption is down to about 80 Bytes/cell independently of n in the present implementation[12]. There is, however, a price to be payed in terms of CPU time. For generation of the point inside cell, or even evaluation of the weight, we need the "absolute" components of x, that is in the reference frame of the root cell, not relative to vertices of the cell. It is, therefore, necessary to use a procedure (a method in the class of cells) which is able to construct the absolute position of a given cell "in flight". This is done by means of tracing all grand-parents of a given cell up to the root cell and translating position and size with respect to its parent into absolute ones, relative to the root cell. It is implemented by means of exploiting the fact, that cell objects are organized into a linked binary tree. The average number of the cells

---

[12]In fact, it can be reduced below 40 Bytes/cell, if really necessary.

to be traced back from a given active cell up to the root cell for $N_c = 10^6$ cells is on the average about $\ln_2 N_c \sim 20$. This may cause $\sim 20\%$ increase in the CPU time of the MC generation – an affordable price, if we remember that the MC efficiency increases mainly with the number of cells. In principle, this kind of the memory saving arrangement is also possible for simplical cells, however, in this case the CPU time overhead would be bigger, because of the necessity of the full linear transformation for each step, on the way from a given cell up to the root cell. In the case of hyperrectangular cells the transformation is much simpler (and faster); it is the translation and/or dilatation along a *single* spatial direction at each step.

Figure 3: Inhibited cell division for first variable, that is for x_1 (right). Foam with 250 cells.

2.7   CPU time saving solution

Final MC efficiency is improved mainly by means of increasing the number of cells N_c. The CPU time of the cell build-up is $T \sim n \times N_c \times N_{samp}$, where N_{samp} is the number of MC events used in the exploration of each newly created cell. The important practical question is: can one somehow reduce N_{samp} without much loss of the final MC efficiency, in order to be able to increase N_c, within the same CPU time budget?

A simple solution is the following: during the MC exploration of a new cell we continuously monitor an accumulated "number of effective events with $w = 1$" defined as $N_{eff} = (\sum w_i)^2 / \sum w_i^2$, and terminate cell exploration when[13] $N_{eff}/n_{bin} > 25$, where n_{bin} is the number of bins in each histogram, which is used to estimate the best division direction/edge parameters. This method helps to cut total CPU time, because the increase

---

[13]The actual limit of equivalent events per bin is the user defined parameter, not necessarily equal 25.

of N_{samp} is not wasted for cells, in which the distribution $\rho(x)$ is already varying very little. At the later stage of the foam evolution this happens quite often. In this method the user may set N_{samp} to a very high value and the program will distribute economically the total CPU time (in terms of N_{samp}) among all cells, giving more CPU time to these cells which really need it, that is to cells with the stronger variation of $\rho(x)$.

2.8   Inhibited variables – flat dependence

In some cases the user may not want `Foam` to intervene into certain variables in the distribution $\rho(x)$, simply because there is little or no dependence on them in $\rho(x)$. The user may draw, of course, these variables directly from any uniform random number generator. He may, however, find it more convenient to get them from the `Foam` program. This is easily implemented in Foam: any variable x_i may be "inhibited" for the purpose of cell splitting procedure. In the `Foam` code it is actually done in such a way that `Foam` is excluding this variable (edge) from the procedure of determining the best binary division of the cell. This provision makes practical sense mainly for the hyperrectangular part of the variable subspace.

In Fig. 3 we show two 2-dimensional foam (250 cells) for the same testing distribution $\rho(x)$ (two Gaussian peaks on the diagonal). In one of them (right plot) we have inhibited split in the first variable, that is for x_1.

2.9   Predefined split points – provision for very narrow peaks

In the practical applications (see refs. [12, 13]) one may encounter in certain variables extremely narrow spikes (narrow resonances). `Foam` exploration algorithm may find it difficult to locate these spikes with the usual method of the MC sampling in the cells, at the early stage of the Foam build-up. For very narrow spikes, or low number of the requested cells, it may not find them at all! The user usually knows in advance the position of these spikes and the `Foam` should have a build in mechanism to exploit this knowledge. The solution is very simple. (It applies for the hyperrectangular subspace of the parameter space only.) The user of `Foam` has a possibility to provide `Foam`, for each variable, with the list of a number of predefined values the first splitting positions of the root cell. In the `Foam` algorithm, it is checked if the list of predefined division points is not empty. If it is the case, then instead adopting the division parameter from the usual procedure described in Section 3, `Foam` takes the division parameter from the list, and removes it from the list. In this way the first few division points are taken from the "user defined menu", if available, and the next ones are chosen with the usual methods. For narrow spikes this method helps `Foam` to locate them and surround with as dense group of cells as necessary.

In fig. 4 we show an example with two Gaussian peaks in which we requested the `Foam` program to use the three predefined division points for the x_1 variable. They are clearly seen as three vertical division lines dividing the entire root cell. In this case peaks are not so narrow and there is no real need for a predefined division. The example is just
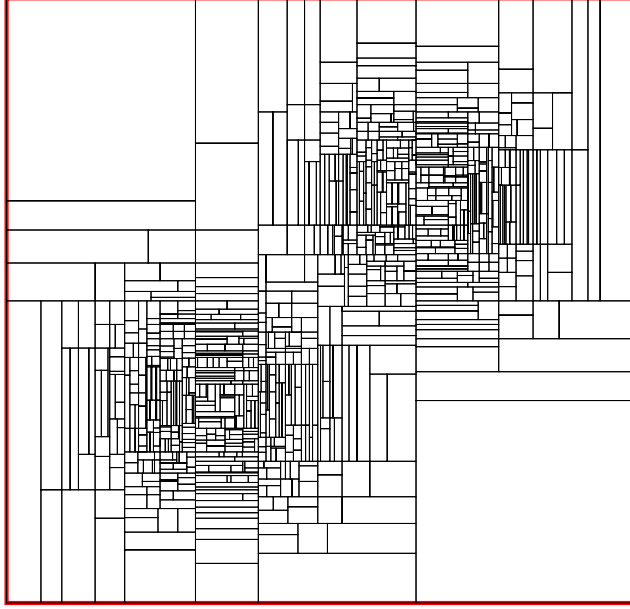
14

Figure 4: Predefined division points at $x_1 = 0.30, 0.40$, and 0.65, for 2000 cells.

illustrating the principle of the method.

2.10 Mapping of variables

If the structure of the singularities is known and/or `Foam` is unable get a reasonable weight distribution for a reasonable number of cells, then it is worth to perform an additional change of variables, such that the transformation Jacobian compensates for the singularities, at least partly. In such a case the user subprogram provides `Foam` with the distribution

$$\rho^\star(y) = \frac{d\rho}{dy_1 \ldots dy_n} = \rho(x_1(y), x_2(y), \ldots ,_n (y)) \left| \frac{\partial x^{(j)}(y)}{\partial y} \right|, \tag{3}$$

instead of the original $\rho(x) = d^n\rho/dx^n$. For each vector y generated by `Foam`, the image vector x is well known in the subprogram calculating $\rho^\star(y)$. A mechanism for exporting x to the outside world has to be usually provided, because `Foam` itself does not know anything about x; it only knows y.

Note that in the limiting case of the of the "ideal mapping" we have

$$\left| \frac{\partial x(y)}{\partial y} \right| \equiv \frac{R}{\rho(x)}, \tag{4}$$

15

consequently $\rho^\star(y) = R$ and this case `Foam` would play merely a role of a provider of the random numbers for y.

The user of `Foam` may also need to apply mapping in the case of a "weak" integrable singularity in the distribution ρ like $\log(x)$ or $\sqrt{x}$. `Foam` can deal with them by brute force, at the expense of a larger number of cells. However, a wiser approach is to apply mapping, in order to remove such a singularities from the distribution.

In the next section we shall describe how to combine mapping method with the multi-branching. Such a mixture is well known as the most powerful method of improving the efficiency of the Monte Carlo method.

2.11   Provisions for the multibranching

In the following we elaborate on the various methods of implementing multibranching [14] with help of `Foam`.

2.11.1   Single discrete variable

As a warm-up exercise let us consider the question: Is `Foam` capable to generate (and sum-up) a discrete variable $i = 1, 2, \ldots, N$ according to the (unnormalised) distribution $r_1, \ldots, r_N$? Of course it can. The simplest way is to define an auxiliary 1-dimensional distribution

$$\rho(x) = r_i, \quad \text{for} \quad \frac{i-1}{N} \le x_i \le \frac{i}{N}, \quad i = 1, 2, \ldots N. \tag{5}$$

The user subprogram providing the above $\rho(x)$ is trivial. If plotted, this $\rho(x)$ would look like histogram with N equal-width bins. `Foam` will build up its own grid of cells (intervals), and if we request enough number of cells (that is $N_c > N$), it will approximate the above $\rho(x)$ very well, with its own "histogram-like" distribution $\rho'(x)$. However, the `Foam` approximation will be never ideal, because Foam is not able to detect the exact position of the discontinuities in $\rho(x)$. (Nevertheless, this will be a workable solution with a very good weight distribution.) The present `Foam` algorithm provides for an essential improvement: one may predefine the division points as $x_i = i/N$, $i = 1, \ldots N$, and set the number of cells to be $N_c \ge N$. In such a case `Foam` will define its cells matching exactly the shape of $\rho(x)$. It will generate points with $w \equiv 1$ and provide the exact sum $R = R' = \sum r_i$, already at the end of the foam build-up. During the generation, `Foam` will generate continuous variable x, which is easily translated into discrete index i.

2.11.2   Discrete and continuous variables

How the above extends to the case of the distribution ρ depending on one discrete variable and the usual n continuous variables? For such a distribution $\rho(y_1, \ldots, y_n, i)$ we define

$$\rho(x_1, x_2, \ldots, x_{n+1}) = \rho(x_1, \ldots, x_n, i), \quad \text{for} \quad \frac{i-1}{N} \le x_{n+1} \le \frac{i}{N}, \quad i = 1, 2, \ldots N, \tag{6}$$

in a completely analogy to Eq. (5). As previously, we provide for the variable x_{n+1} a list of predefined division points $x_{n+1}^{(i)} = i/N$, $i = 1, 2, \ldots N$ and, of course, we request for $N_c >> N$. There is still one small problem: `Foam` may "by mistake" perform an unnecessary cell division for variable x_{n+1}, simply due to statistical errors in the "projection histogram" described in Section 3.4.1. This problem is solved in `Foam` in an elegant way: in addition to providing for x_{n+1} predefined division points the user of `Foam` may declare x_{n+1} as an "inhibited variable" in the sense of Section 2.8. In this case `Foam` will still split cells according to a list of predefined division points for x_{n+1}, but will not perform any additional division in this variable! For the generated MC events the translation of the continuous x_{n+1} to the discrete i is done as trivially as before. The above method is the basic method of the implementation of the "multibranching" (or "multichannel") MC method using Foam. Let us call it "predefined and inhibited division", for short a PAID method. We shall also describe below how to combine PAID method with mapping, etc.

In order to appreciate more fully the advantages of PAID, let us consider a more straightforward implementation of the multibranching. In the object oriented environment one may construct N instances of the `Foam` object, each of them for the n-dimensional function $\rho(x_1, \ldots, x_n, i)$, initialize them (creating separate foam of cells) and generate event (x, i) with the associated weight $w_i(x)$. Index i can be chosen according to probability $p_i = R'_i / \sum_j R'_j$, where R'_i are provided by the i-th object of the `Foam` class (at the end of its initialization). The total weight of the event is $w(x, i) = w_i(x)/p_i$. Let is call this scenario an "externally organized multibranching", for short EOM.

Both methods have certain advantages and disadvantages. In PAID the user does not need to organize the optimal/efficient generation of the branching index i. The root cell is divided into N equal size *sub-root* cells, which then evolve separately into independent system of the cells, adapting individually to the singularities in the i-th component of ρ. `Foam` adjusts relative importance of the sub-root cells and their descendants, and finds the optimal number of the division cells in the N sub-foams within the requested total memory limit. In the EOM scheme these adjustments for the individual branches has to be done by the user. On the other hand, in some rare cases, the user may want to configure the `Foam` objects for each branch individually. In the EOM scheme it can be done, for each `Foam` object separately. In the PAID scheme it cannot be done, because all cells have the same properties, the cell split algorithm is the same, cell geometry is common, etc. In most cases, the PAID method will be preferred, because it is easier to organize.

In the following we shall concentrate on the PAID scheme. In this case, the normalization integral is provided by the `Foam` at the end of the exploration phase, and it includes the sum over discrete variable

$$R' = \sum_{i=1}^{N} \int \rho'_i(x) dx^n. \tag{7}$$

We also have the usual relation between the average weight and the integral

$$R = \sum_{i=1}^{N} \int \rho(x)dx^n = R' \langle w \rangle. \tag{8}$$

The above method extends trivially to the case of several discrete variables. As already stressed, the relative probabilities of the discrete components $p_i \sim R'_i = \int \rho'_i(x)dx^n$ in the MC generation are automatically adjusted by the `Foam` algorithm, such that the maximum weight or the total invariance is minimized. The arranging for that in the user program in the EOM scheme would require an extra programming effort, while in `Foam` this comes for free.

2.11.3 Multi-layer method

There is an alternative PAID-type method of dealing with the problem of the discrete variable, which generates points according to $\rho(x_1, \ldots, x_n, i)$, $i = 1, 2, \ldots N$. It will produce the same distribution but will differ from PAID in the MC efficiency, in terms of the maximum weight or variance. One may simply generate with help of `Foam` the n-dimensional auxiliary distribution

$$\bar{\rho}(x_1, \ldots, x_n) = \sum_{j=1}^{N} \rho(x_1, \ldots, x_n, j) \tag{9}$$

and next, for each generated x, chose randomly discrete variable i according to the probability

$$p_i(x) = \rho(x, i) / \sum_{j=1}^{n} \rho(x, j). \tag{10}$$

Let us call it PAID*, or a multi-layer method. This method is slightly less convenient to implement, as is clearly seen for $n = 0$, where the user effectively has to generate the discrete variable $i = 1, 2, \ldots, N$ according to the above probability by himself, by means of creating an inverse cumulative distribution, mapping random number into i, etc., while in the standard PAID scenario this all job is done by the `Foam` program[14]. Furthermore, in PAID method each component distributions $\rho(x, j)$ may have a "cleaner" structure of the singularities then the sum. Consequently, in the PAID method `Foam` will probably find it easier to learn the shape of each component distribution than of the sum in PAID*. These two kinds of equivalent multibranching algorithms like PAID and PAID* are described and analysed in Ref. [14]. PAID* method is used in the KKMC generator of Ref. [13] to generate index i numbering type of the final state quark or lepton.

---

[14] The mapping $x_{n+1} \rightarrow i$ is a simple arithmetic operation.

2.11.4 Multibranching and mapping

However, the most important reason setting up `Foam` according to PAID scenario, with the separate foam build-up for each component distributions $\rho(x, j)$, is that for each component one may apply *individually adjusted mapping* of variables, which makes every component distribution much less singular. The combination of the mapping and multi-branching is one of the most powerful known methods of improving MC efficiency [15, 14]. How it can be actually realized with help of `Foam`, depends on the properties of the distribution $\rho(x)$ to be generated. In the case when we have an explicit sum over many components

$$\rho(x_1, \ldots, x_n) = \sum_{j=1}^{N} \rho(x_1, \ldots, x_n, j), \tag{11}$$

each of the components being positive, with distinctly different and well known structure of the singularities, we would recommend the use `Foam` in the PAID scheme. Knowing the structure of singularities, we may be able to introduce mapping in each component separately, which compensates for these singularities with the Jacobian factor. In such a case `Foam` is provided with the following distribution:

$$\rho(y_1, \ldots, y_n, j) = \rho(x_1^{(j)}(y), \ldots, x_n^{(j)}(y), j) \left| \frac{\partial x^{(j)}(y)}{\partial y} \right|, \quad j = 1, 2, \ldots, N, \tag{12}$$

understanding that the translation of the discrete index j into a continuous variable y_{n+1}, is done in the usual way. The foam of cells is, of course, build-up in the y-variables, different for each j-th branch. The user is fully responsible for the proper mapping $x^{(j)}(y)$, $j = 1, 2, \ldots, N$, and the calculation the Jacobian factor in every component (branch). In the user subprogram providing the ρ-distribution the variable y_{n+1} will be translated first into index j and then, depending on the value of j, a given type of a mapping will be applied. For the outside part of the code the index j can be made available, or it may be hidden (erased from the record), depending on the needs of a specific application.

In some cases, however, we do not have at our disposal an unique split of the $\rho(x)$ into well defined positive components like in eq. (12), but rather only a rough idea about the leading singularities. That means, one is able to construct the distribution

$$\rho(x_1, \ldots, x_n) \sim \bar{\rho}(x_1, \ldots, x_n) = \sum_{j=1}^{N} \bar{\rho}(x_1, \ldots, x_n, j), \tag{13}$$

where $\bar{\rho}(x, j)$ have the same type of the leading singularities as $\rho(x)$, and we know the normalization of singularities in $\rho(x)$ up to a constant factor; that is for x in the neighbourhood of the j-th singular point, a line or a (hyper)plane, only $\bar{\rho}(x, j)$ really matters, that is $\rho(x) \simeq C_j \bar{\rho}(x, j)$ where C_j is not known a priori[15].

---

[15]This definition is not very precise, it roughly means that each component is approximately a product of the singular factors and cannot be reduced into sum of such.

In addition, let us assume, that we are able to compensate for the singularities in each $\bar{\rho}(x, j)$ *exactly* by dedicated mapping specific to singularities in the j-th branch. In other words, the mapping is *ideal* in each branch:

$$\left| \frac{\partial x^{(j)}(y)}{\partial y} \right| = \frac{\bar{R}_j}{\bar{\rho}(x, j)}. \tag{14}$$

The above means also, that we know analytically the exact values of the integrals[16]: $\bar{R}_j = \int \bar{\rho}(x, j) dx^n$.

In such a case we may employ the algorithm of `Foam` successfully by means of defining the "branching ratio"

$$b_j(y_1, \ldots, y_n) = \bar{\rho}(y_1, \ldots, y_n, j) / \bar{\rho}(y_1, \ldots, y_n), \quad \sum b_i(y) = 1, \tag{15}$$

constructing the distribution to be digested by `Foam` as

$$\rho(y_1, \ldots, y_n, j) = b_j(x) \rho(x_1^{(j)}(y), \ldots, x_n^{(j)}(y)) \left| \frac{\partial x^{(j)}(y)}{\partial y} \right| = \frac{\bar{R}_j \, \rho(x(y))}{\sum_l \bar{\rho}(x(y), l)}, \tag{16}$$

and proceeding as in the PAID scheme described previously.

The role of the function $b_j(x)$ is to isolate out from $\rho(x)$ "a layer" including just one known type of singularity. In order to see how this method works, let us consider the j-th singularity being a $\delta^{(n)}(x - a)$ shape (narrow Gaussian peak etc.) of the size ϵ. Then, in the neighbourhood of the singularity $b_j(x) = 1$ and $\rho(x) \simeq C_j \bar{\rho}(x, j)$, while further away from the singularity position $\rho(x) \sim \epsilon^n$, and is negligible. The `Foam` program will, of course, include the C_j factor properly in the normalization, and build up the foam of cell everywhere, close to a singularity and far away. It will do it, however, not in the x variables but in the y-variables. Now, the mapping $x \to y$ (specific to j-th branch) will expand the singularity neighbourhood to size of $\mathcal{O}(1)$, while the y-image of the remaining space will be shrinker down to a size of $\mathcal{O}(\epsilon)$. This can be a source of the following pitfall to be remembered: in the shrinked y-domain of $\mathcal{O}(\epsilon)$, in the places where the other singularities $i \neq j$ are placed, $\rho(y, j)$ may get narrow spikes or dips of the height of $\mathcal{O}(1)$, such that their integral contribution will be negligible, of $\mathcal{O}(\epsilon^n)$. Nevertheless, the `Foam` algorithm may find it difficult to locate these structures, and this may lead to a small but finite bias of the generated distributions and calculated integrals. One should keep this in mind and perform special tests (MC runs with a maximum number of cells, and high MC statistics) in order to check that this effect is not present.

The above method is quite similar to that of Ref. [15]. One difference is that in method of Ref. [15] there are several iteration with the aim at adjusting the relative normalization of the components $\bar{\rho}(x_1, \ldots, x_n, j)$ to $\rho(x)$. Our scheme could be effectively regarded as a method of Ref. [15] with just one iteration; that is the first step being the foam build-up, and the second step (1st iteration) being the MC simulation. One iteration is sufficient in the limit of vanishing overlap of the components $\bar{\rho}(x_1, \ldots, x_n, j)$ in the entire $\bar{\rho}(x)$. While

---

[16]In fact, we could normalize $\bar{\rho}(x, j)$ to unity, $\bar{R}_j = 1$, if we wanted.

in the method of Ref. [15] a better adjustment is provided by the next iterations, in `Foam` the cellular adaptive method provides an extra mileage. One cannot therefore say which one is better in general – it depends on the distribution $\rho(x)$.

In fact, in the PAID scheme with the mapping, extra iterations are also possible. It can be done as follows: (a) read R_j from all N "leading cells" after the foam build-up, (b) rescale $\bar{\rho}(y_1, \ldots, y_n, j) \to (R_j/\bar{R}_j)\bar{\rho}(y_1, \ldots, y_n, j)$ and (c) repeat the foam build-up[17] for the new branching ratios $b_j(x)$ in eq. (16). The above procedure can be repeated. Whether such an iteration is profitable it depends on the particular distribution – we expect that in most cases it is not necessary, due to adaptive capabilities of Foam.

Last not least, let us also consider the case of a sum of integrals with different dimensionality, or in other words, the distribution in which the number n_i of the continuous variables $x_1, ..., x_{n_i}$ depends on a certain discrete "master variable" $i = 1, ..., N$ (for example $n_i = i$)

$$R = \sum_{i=1}^{N} \int \rho_i(x_1, ..., x_{n_i}) \tag{17}$$

`Foam` can deal with this case too. The simplest solution is to find the maximum dimension $n_{\max}$ and add extra dummy variables on which ρ_i does not depend, such that formally all sub-distributions have the same dimension $n_{\max}$. In this way one is back in a situation described earlier, and may apply the PAID method, with or without the additional mapping. The slight drawback of this solution is that in the present implementation of `Foam` we cannot inhibit the unnecessary cell divisions across the directions of the newly introduced dummy variables – simply because they are not the same in all branches[18]. Because of that, this kind of a problem can be in some cases dealt with more efficiently using EOM scenario, with a separate `Foam` object for each branch.

For additional practical examples on how to realize multibranching with `Foam`, see Section 6.

3 Cell split algorithm and geometry

As already indicated, our algorithm of the cell split covers two strategies: (A) minimization of the maximum weight $w_{\max}$ and (B) minimization of the variance σ, where both $w_{\max}$ and σ are calculated in the Monte Carlo generation, using the MC weight $w = \rho/\rho'$. The distribution ρ' is the result of the exploration (it is constant over each cell) and is frozen at the end of exploration. During the subsequent MC event generation, events are generated according to $\rho'(x)$. Its integral $R' = \int \rho'(x)d^n x$ has to be known *exactly* before the start of the MC generation. The integral $R = \int \rho(x)d^n x$ is obtained up to a statistical error at the end MC event generation from the usual relation to the average weight: $R = R'\langle w \rangle_{\rho'}$. The average $\langle ... \rangle_{\rho'}$ is over events generated according to ρ'.

---

[17]In the programming with `Foam` it is possible to erase all cells from memory and rebuild them.

[18]A more sophisticated procedure of inhibiting the division could be implemented in `Foam`, if there is a strong demand for that.

There is another important ingredient in the algorithm of the cell split: in addition to the auxiliary distributions $\rho'(x)$ we also define another distribution $\rho_{loss}(x)$ related to integrand $\rho(x)$. The important role of the distribution $\rho_{loss}(x)$ is to guide the build-up of the foam of cells; the function $R_{loss} = \int \rho_{loss} d^n x$ is minimized in the process – its value is decreasing step by step, at each the cell split. Obviously, both $\rho_{loss}(x)$ and $\rho'(x)$, are evolving step by step during the foam build-up. Once the division process is finished, the distribution $\rho_{loss}(x)$ is not used anymore; MC events are generated with $\rho'(x)$. Nevertheless, $\rho_{loss}(x)$ is strongly related to the properties of the weight distribution in the MC generation phase.

(A) In the case when our ultimate aim is to minimize $w_{\max}$ we define

$$
\begin{aligned}
\rho'(x) &\equiv \max_{y \in Cell_I} \rho(y), \quad \text{for} \quad x \in Cell_I, \\
R_{loss} &= \int d^n x \, [\rho'(x) - \rho(x)] = \int d^n x \, \rho_{loss}(x).
\end{aligned}
\tag{18}
$$

The distribution ρ_{loss} is the difference between the "ceiling distribution" ρ' and the actual distribution ρ from which it is derived. The rejection rate in final MC run is just proportional to the integral over the loss distribution $\rho_{loss}(x)$ by construction, i.e. the rejection rate$= R_{loss}/R$. (This justifies the name "loss".) The distribution $\rho_{loss}(x)$ has also a clear geometrical meaning, see below.

(B) In the case when we do not care so much about the maximum weight and the rejection rate but rather we want to minimize the ratio of the variance to average of the weight, $\sigma/\langle w \rangle$, in the final MC generation, then we are led to the following definition:

$$
\begin{aligned}
\rho'(x) &\equiv \sqrt{\langle \rho^2 \rangle_I}, \quad \text{for} \quad x \in Cell_I, \\
\rho_{loss}(x) &\equiv \sqrt{\langle \rho^2 \rangle_I} - \langle \rho \rangle_I, \quad \text{for} \quad x \in Cell_I.
\end{aligned}
\tag{19}
$$

The average $\langle ... \rangle_I$ is over the I-th cell; see Appendix A for a detailed derivation of the above prescription. The ratio $\sigma/\langle w \rangle$ in the final MC generation is a monotonous ascending function of the $R_{loss} = \int \rho_{loss}(x) dx^n$, see Appendix A. Consequently, minimization of R_{loss} is equivalent to minimization of $\sigma/\langle w \rangle$.

3.1 Rules governing binary split of a cell

The basic rule governing the development of the foam of cells are the following:

(a) For the next cell to be split we chose a cell with the biggest[19] R_{loss}.

(b) Position/direction of a plane dividing a parent cell into two daughter cells $\omega \to \omega' + \omega''$ is chosen to get the largest possible decrease $\Delta R_{loss} = R_{loss}^{\omega} - R_{loss}^{\omega'} - R_{loss}^{\omega''}$.

---

[19]In the `Foam` code there is also an option of choosing randomly the next cell to be split, according to probability proportional to R_{loss}, instead of a cell with the largest R_{loss}.

How the split of a given cell into two daughter cells in step (b) is done in practice? The method relies upon a small MC exercise within a cell, in which a few hundreds of events are generated with a flat distribution. They are weighted with ρ and projected onto k (hyperrectangular case) or $n(n+1)/2$ (simplical case) *edges* of the cells. In the mixed case of cell being the Cartesian product of a k-dimensional hyperrectangle and n-dimensional simplex, there are $k + n(n+1)/2$ projections/edges. Resulting histograms are analysed and the best "division edge" and "division hyperplane position" are found – this one for which the estimate (forecast) of the ΔR_{loss} is the biggest. In the actual `Foam` algorithm, each new born cell is immediately explored, its R_{loss}, R and R' are calculated, and the best candidate of the direction and position of the dividing plane are established and memorized, as the attributes of the cell, see below for details. In this way, every newly created cell is ready for an immediate binary division.

3.2   Geometry of binary split of a cell
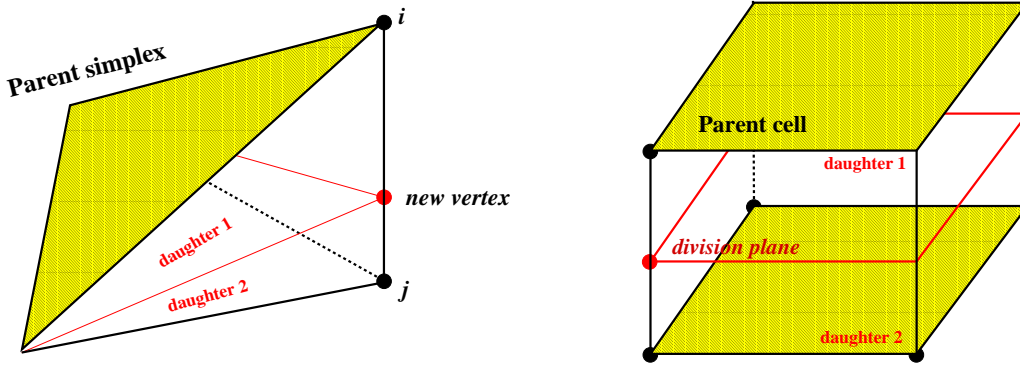
Figure 5: Geometry of the split of the 3-dimensional cell being simplex or hyperrectangle.

In Fig. 5 we show a 3-dimensional cell being a simplex or a hyperrectangle and we visualize the geometry of their split.

Let us describe first the split of the n-dimensional simplical parent cell into two daughter cells. In the case of the simplical cell [1] we need to know which of $n(n+1)/2$ edges defined by any pair of vertices of a given simplex is used in the split. Suppose that it is an edge defined by a pair of indices (i,j), $i \neq j$, where $i, j = 1, 2, ..., N_V$, of the vertices $(\vec{V}_{K_i}, \vec{V}_{K_j})$, see Sect. 2.4 for the method of numbering of the vertices. A new vertex V_{N_V+1} is put somewhere on the line (edge) in between the two vertices

$$V_{N_V+1} = \lambda \vec{V}_{K_i} + (1 - \lambda_{div})\vec{V}_{K_j}, \ \ 0 < \lambda_{div} < 1, \tag{20}$$

where the division parameters λ_{div} is determined using an elaborate procedure described later in this section, and the number of vertices is updated $N_V \rightarrow N_V + 1$. With the new

23

vertex two daughter simplices are formed with the following two lists of vertices (their pointers):

$$(K_1, K_2, ..., K_{i-1}, (N_V + 1), K_{i+1}, ..., K_{j-1}, K_j, K_{j+1}, ..., K_n, K_{n+1}),$$
$$(K_1, K_2, ..., K_{i-1}, K_i, K_{i+1}, ..., K_{j-1}, (N_V + 1), K_{j+1}, ..., K_n, K_{n+1}). \tag{21}$$

For the k-dimensional hyperrectangular cell defined with a pair of the vectors $(\vec{q}, \vec{h})$ we decide first about the direction of the division plane. Assuming that the division plane is perpendicular to i-th direction the two daughter cells (a) and (b) are defined with the two pairs of the new vectors as follows:

$$\vec{q}^{(a)} = (q_1, q_2, \ldots, q_k), \quad \vec{h}^{(a)} = (h_1, h_2, \ldots, h_{i-1}, h_i \lambda_{div}, h_{i+1}, \ldots, h_k),$$
$$\vec{q}^{(b)} = (q_1, \ldots, q_{i-1}, q_i + h_i \lambda_{div}, h_{i+1}, \ldots, q_k), \quad \vec{h}^{(b)} = (h_1, \ldots, h_{i-1}, h_i(1 - \lambda_{div}), h_{i+1}, \ldots, h_k). \tag{22}$$

The 3-dimensional case of the simplical and hyperrectangular cell split made in this way is illustrated in Fig. 5.
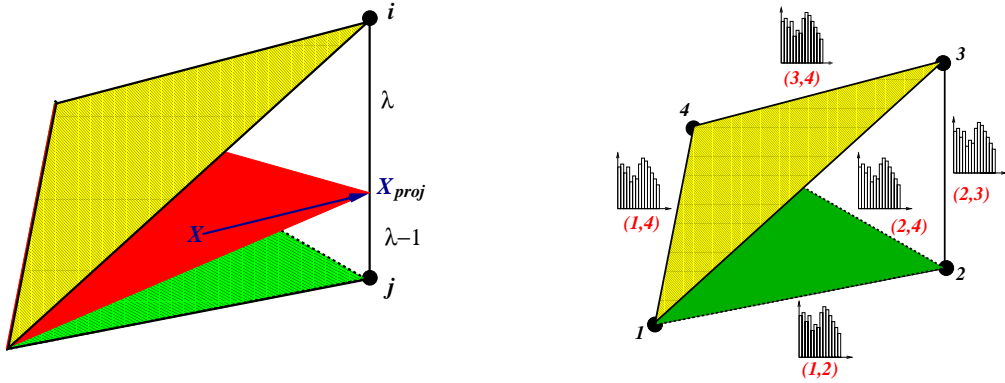
Figure 6: Geometry of the split of the 3-dimensional simplex cell.

3.3 Projecting points into an edge

Before we describe the determination of the division edge and of the division parameter λ_{div} let us still discuss certain geometric aspect of the Foam algorithm – that is how we project a point $\vec{x}$ inside a cell onto one of the edges of the cell. In the case of a simplex the edges are numbered by the pair of indices (i, j), $i > j$, which number edges spanned by a pair of vertices[20] $(\vec{V}_{K_i}, \vec{V}_{K_j})$, while in the case of the hyperrectangle the i-th edge is spanned by the pair of vectors $\vec{q}$ and $\vec{q}_+ = (q_1, \ldots, q_{i-1}, q_i + h_i, \ldots, q_n)$. The point x inside a cell is projected into the edge and parametrized using parameter $\lambda \in (0, 1)$.

---

[20] As explained in Sect. 2.4, numbering of vertices is done using pointers K_i to the elements of the array of vertices.

Figure 7: Geometry of the split of the 3-dimensional hyperrectangular cell.

Parameter λ will be used to define an auxiliary projection $d\rho/d\lambda$ for each edge in the following subsection. In particular we have to know how to evaluate λ in an efficient way. In the case of a simplex cell we have:

$$\vec{x}_{proj} = \lambda\vec{V}_{K_i} + (1 - \lambda_{ij})\vec{V}_{K_j}, \ \ 0 < \lambda_{ij} < 1, \tag{23}$$

where

$$
\begin{aligned}
\lambda_{ij}(\vec{x}) &= \frac{|\mathrm{Det}_i|}{|\mathrm{Det}_i| + |\mathrm{Det}_j|}, \\
\mathrm{Det}_i &= \mathrm{Det}(\vec{r}_1, \ldots, \vec{r}_{i-1}, \vec{r}_{i+1}, \ldots, \vec{r}_n, \vec{r}_{n+1}), \\
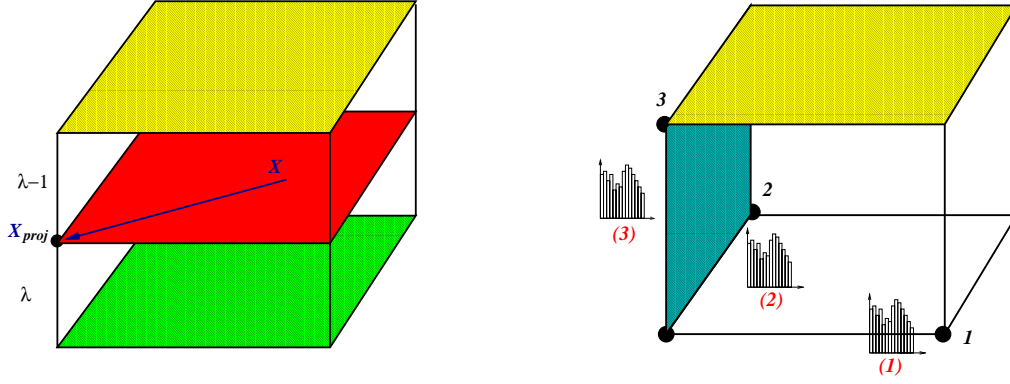\mathrm{Det}_j &= \mathrm{Det}(\vec{r}_1, \ldots, \vec{r}_{j-1}, \vec{r}_{j+1}, \ldots, \vec{r}_n, \vec{r}_{n+1}), \qquad \vec{r}_j = \vec{V}_{K_j} - \vec{x},
\end{aligned}
\tag{24}
$$

and $\mathrm{Det}(x_1, x_2, ..., x_n)$ is determinant. The case of a hyperrectangular cell is much simpler:

$$\lambda_i = (x_i - q_i)/h_i. \tag{25}$$

Obviously, due to time consuming evaluation of the determinants at higher dimensions, the above projection procedure will be much slower for simplices than for hyperrectangles.

In Fig. 6 we illustrate projection procedure into six edges for the 3-dimensional simplex and in Fig. 7 the case of the three edges of the 3-dimensional hyperrectangular cell.

3.4 Determination of an optimal division edge and of λ_{div}

Our aim is to find out which division plane, that is cutting through which edge, provides the best gain of the total integral R_{loss}, summed over two daughters, as compared to the parent cell. In order to do that, first we analyse all possible division planes, for all edges, and find out the best one, in terms of the gain in R_{loss}. In other words, we go through all edges (k edges for hyperrectangle and/or $n(n + 1)/2$ for simplex), for each edge we find out the best parameter λ_{div} and the corresponding best gain in R_{loss}. Then,
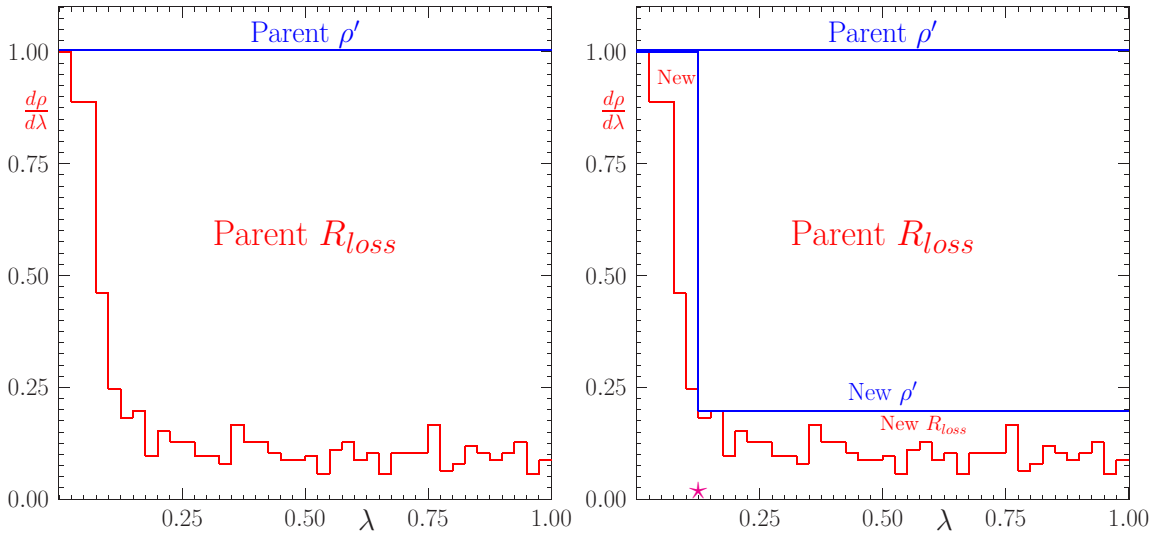
25

Figure 8: Projection histogram. The case of optimizing the maximum weight.

we compare between the gains in R_{loss} for all edges and define the optimal edge as the one with the best gain in R_{loss}. The procedure of finding the best λ_{div} is essentially the same for simplical and hyperrectangular cells – on the other hand, there is a difference in the algorithm of finding the best λ_{div} between the cases of optimization of the maximum weight and of the variance, see the following discussion.

3.4.1   Optimization of the maximum weight – choosing λ_{div}

Let us consider first the case of finding the best λ_{div} for R_{loss} corresponding to *optimization of the maximum weight*. The 1-dimensional case is a good starting point. The cell in this case is just an interval $(q, q + h)$ and $\lambda = (x - q)/h$. In the left part of the Fig. 8 we see a histogram with N_b bins, made of 1000 events generated inside a cell (interval) using the weight $w = \rho(x)$, that is the histogram represents approximately the distribution $d\rho/d\lambda$, $\lambda \in (0, 1)$. This distribution (histogram) peaks close to lower edge. The function $\rho'(x) = \max_{x \in Cell} \rho(x)$ is constant over the cell and is depicted as an upper horizontal line marked "Parent ρ'". The contribution of this particular (parent) cell to $R_{loss} = \int_{cell} \rho_{loss} dx = \int_{cell} (\rho'(x) - \rho(x)) dx$ is easily recognized as an area between the line marked "Parent ρ'" and the histogram line. If we have stopped the exploration at this stage, with this parent cell, then in the MC run points would be generated with the flat "Parent ρ'" and the weight would be $w = \rho(x)/\rho$. Turning weighted events into unweighted by means accepting $r < w$ events and rejecting $r < w$, where $0 \leq r \leq 1$ is an uniform random number, would correspond to generating points (λ, r) within rectangle below "Parent ρ'" line, accepting all points which are below the histogram line and rejecting points above the histogram line in the area marked "Parent R_{loss}". This justifies the subscript "loss".

The best cell division is found by examining all $N_b - 1$ end-points $\lambda = q + ih/N_b$,

$i = 1, 2, ..., N_b - 1$ of the bins in the histogram, as a possible candidate for the division point (plane in two and more dimensions) between the two daughter cells, and choosing the best one. In the right part of the Fig. 8 we have marked such a candidate division point with a star. For a given division point, we determine for two daughter cells the new "ceiling function" ρ'; in Fig. 8 it is line marked "New ρ'". For each daughter cell we evaluate R_{loss}. The summary R_{loss} for both daughter cells is easily recognized as an area between the line marked "New ρ'" and the histogram. Of course, we get automatically the new total R_{loss} smaller than for the original parent cell! This procedure is repeated for all possible $j = 1, 2, ..., N_b - 1$ division points and each time we record the net gain in $\Delta_j R_{loss} = R_{loss,parent} - R_{loss,daughter1} - R_{loss,daughter2}$. For the actual best division point we chose the division point with the largest gain $\Delta_j R_{loss}$. In Fig. 8 the star marks the best division point.
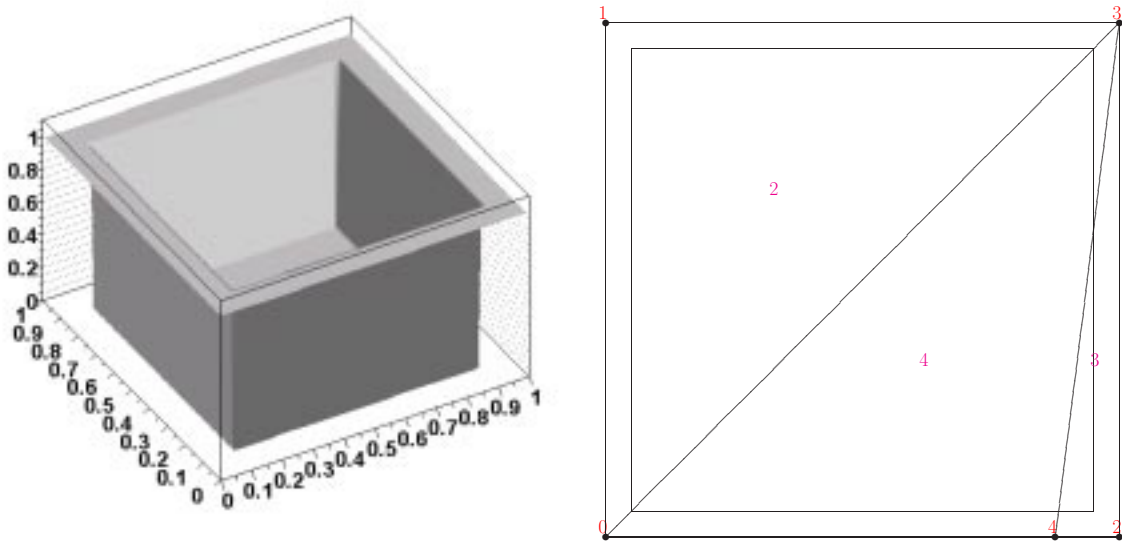
Figure 9: Two dimensional $\rho(\vec{x})$ (left) and the geometry of the first three simplical cells (right). Inside the area marked by dashed line $\rho(\vec{x}) = 0$.

Let us now consider the 2-dimensional distribution $\rho(x)$ depicted in the left part of Fig. 9, which is nonzero within the narrow strip along four edges of the rectangle. In the simplical mode Foam divides starting n-dimensional hyperrectangle into $n!$ simplices – in this case into 2 triangles. We concentrate on the division procedure of the lower triangle, see the right part of Fig. 9. In the exploration of this triangular cell we use 1000 MC points and project them onto 3 edges. The corresponding 3 histograms are shown in Fig. 10, where the middle histogram represents the projection onto the diagonal – this is why it features two peaks distinct at the ends. In all three plots we have also drawn the curve for $\rho'(x)$ for the best hypothetical split. The most promising split in terms of the gain in R_{loss} turn outs to be related to the middle plot and is marked in the right part of
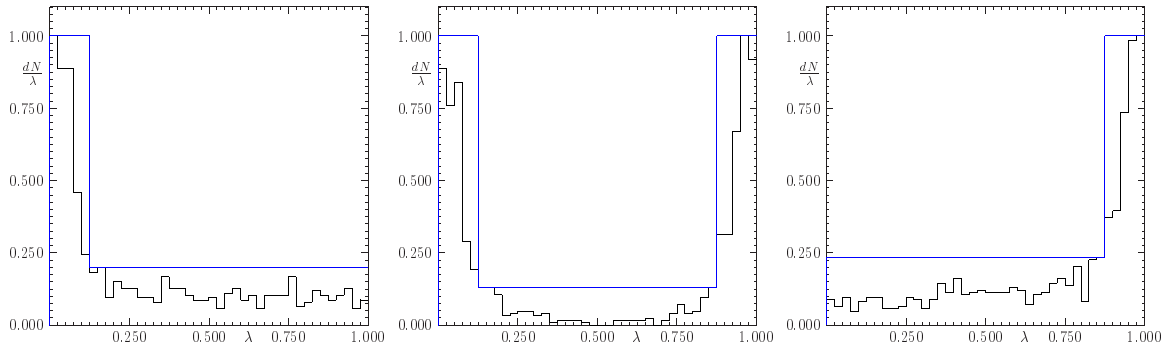
Figure 10: Distributions used in construction of λ_{div} in the case of optimizing the maximum weight.

Fig. 9. The reader may notice that the $\rho'(x)$ in the middle plot of Fig. 10 is not of the type discussed above, because it has two discontinuities instead of one. This is because in Foam we have introduced certain refinement of the algorithm of finding an optimal λ_{div}. One may easily notice that the algorithm described above could not locate correctly the drop in the distribution $d\rho/d\lambda$ of the middle plot, because there are two equally strong peaks at the end of this distribution[21]. In the improved version of the algorithm the search of the optimal λ_{div} uses all pairs of the bin edges $(\lambda_i, \lambda_j) = (q + ih/N_b, \ q + jh/N_b)$, $0 \leq i < j \leq N_b$. For every pair (i, j) a new ceiling function $\rho'(x)$ is determined such that it is unchanged outside the subinterval (λ_i, λ_j) and is "majorizing" the histogram bins inside this subinterval. Once we find out the best pair (i, j) in terms of R_{loss}, then we take either λ_i or λ_j as a division point λ_{div} (at least one of them is not equal 0 or 1). In the case of two or more peaks in $d\rho/d\lambda$ the resulting division point λ_{div} happens to be close to one of the edges of the gap between the two peaks. This feature prevents the Foam algorithm (at least partly) from placing a new division plane across a void in the multidimensional distribution $\rho(x)$. In other words such a void will "repel" the division planes from the voids. In the case of the double peak structure of the middle plot of Fig. 10, the improved algorithm will of course allocate big value of R_{loss} to a new cell (interval) which includes the gap. In the next step of the foam build-up this cell (interval) will have big chance to be split, and for this split the position of the split point will be located at the second edge of the gap. This is exactly what we need for an efficient foam evolution.

3.4.2 Optimization of the variance – choosing λ_{div}

Let us consider now the case of finding out the best λ_{div} for R_{loss} corresponding to *optimization of the variance*. The strategy is again to choose λ_{div} minimizing R_{loss}. In Fig. 11 we illustrate our algorithm on the example of the three projections of the triangular cell. The three projections correspond to a triangular cell in two dimensions. (We do not specify $\rho(x)$, as it is irrelevant for the purpose of our explanation.) In the upper row of three plots in Fig. 11 we show as a horizontal line the value of the distribution $\rho_{loss} = \sqrt{\langle \rho^2 \rangle}$ (it

---

[21]The algorithm would pick up λ_{div} in a random point between the two peaks.
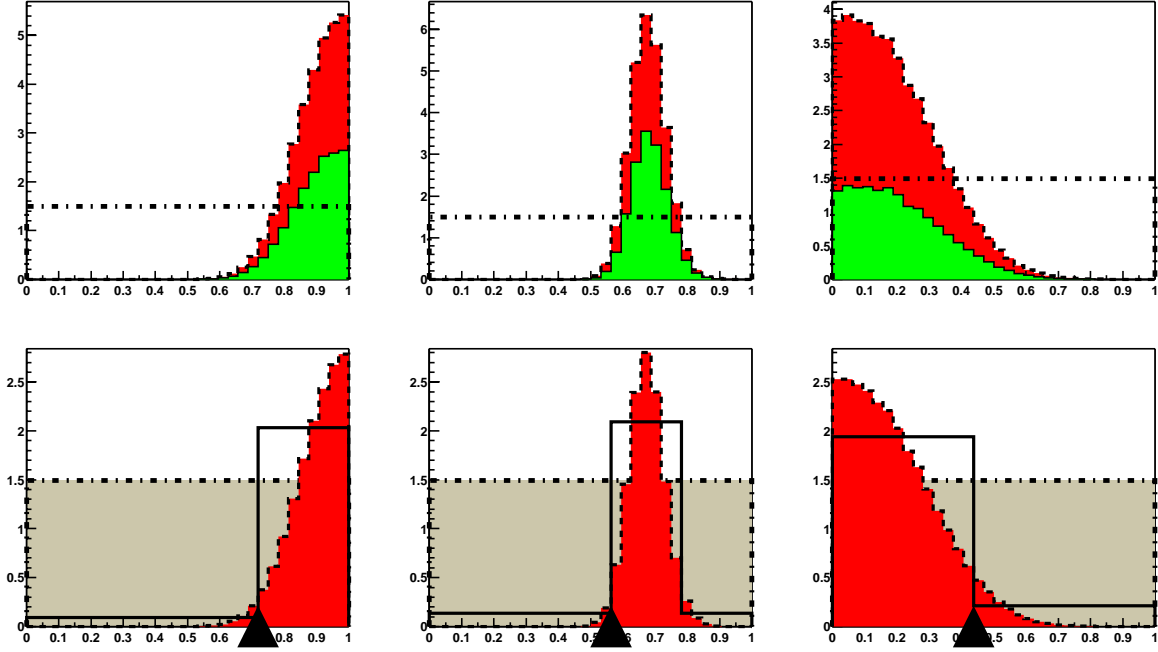
Figure 11: Distributions used in construction of λ_{div} in case of optimizing the variance.

is the same for all 3 projections). The solid histogram is the distribution $d\rho/d\lambda$ and the dashed histogram is the distribution $d\rho_{loss}/d\lambda$ calculated bin by bin using $\sqrt{\langle\rho^2\rangle}$, treating every bin as a separate cell. The properly normalized difference of the above two distributions $\sqrt{\langle\rho^2\rangle} - \langle\rho\rangle$ is plotted separately as the dashed histograms in the lower row of the three plots in Fig. 11. The horizontal line for the total ρ_{loss} is shown once again there. The histogram of $d\rho_{loss}/d\lambda$ gives us an idea where the biggest source of the variance is located and our aim is to "trap" it properly with an intelligent choice of λ_{div}. We follow a similar algorithm as in the case of the maximum weight minimization, namely we loop over pairs of the bin edges $(\lambda_i, \lambda_j) = (q + ih/N_b, q + jh/N_b)$, $0 \le i < j \le N_b$, and for every pair we calculate R_{loss} inside the interval (λ_i, λ_j) and outside this interval. We find out which (i, j) provides the biggest gain $\Delta_{ij}R_{loss} = R_{loss,parent} - R_{loss,Inside} - R_{loss,Outside}$. In the lower row of the plots in Fig. 11 we show as a solid line the distribution of R_{loss} for the best pair (i, j). Depending on the peak structure, at least one of the division point, of the optimal pair (λ_i, λ_j) is different from zero or one, and we take this one as λ_{div}. In Fig. 11 the chosen λ_{div} are marked with the black triangles. The above procedure is done for each edge and the best $\Delta_{ij}R_{loss}$ is used as a guide to define an edge for which the next cell division will be executed. The information about the best edge and the best division point λ_{div} is recorded in the cell object. As seen in Fig. 11, λ_{div} tends to fall at the position where $d\rho_{loss}/d\lambda$ drops or increases sharply. Note that since the division point is always at the edge of the bin, it is therefore a rational number, $\lambda_{div} = j/N_b$. This has interesting consequences, since the number of the bins N_b is fixed, it is therefore

29

enough to memorize this integer index j (2 Bytes) together with the integer index of the division cell edge (also 2 Bytes) as an attributes of the cell, in order to define fully and uniquely the geometry of the division of the cell! See Section 2.6 for more details how this is exploited to save computer memory needed to encode the entire foam of cells.
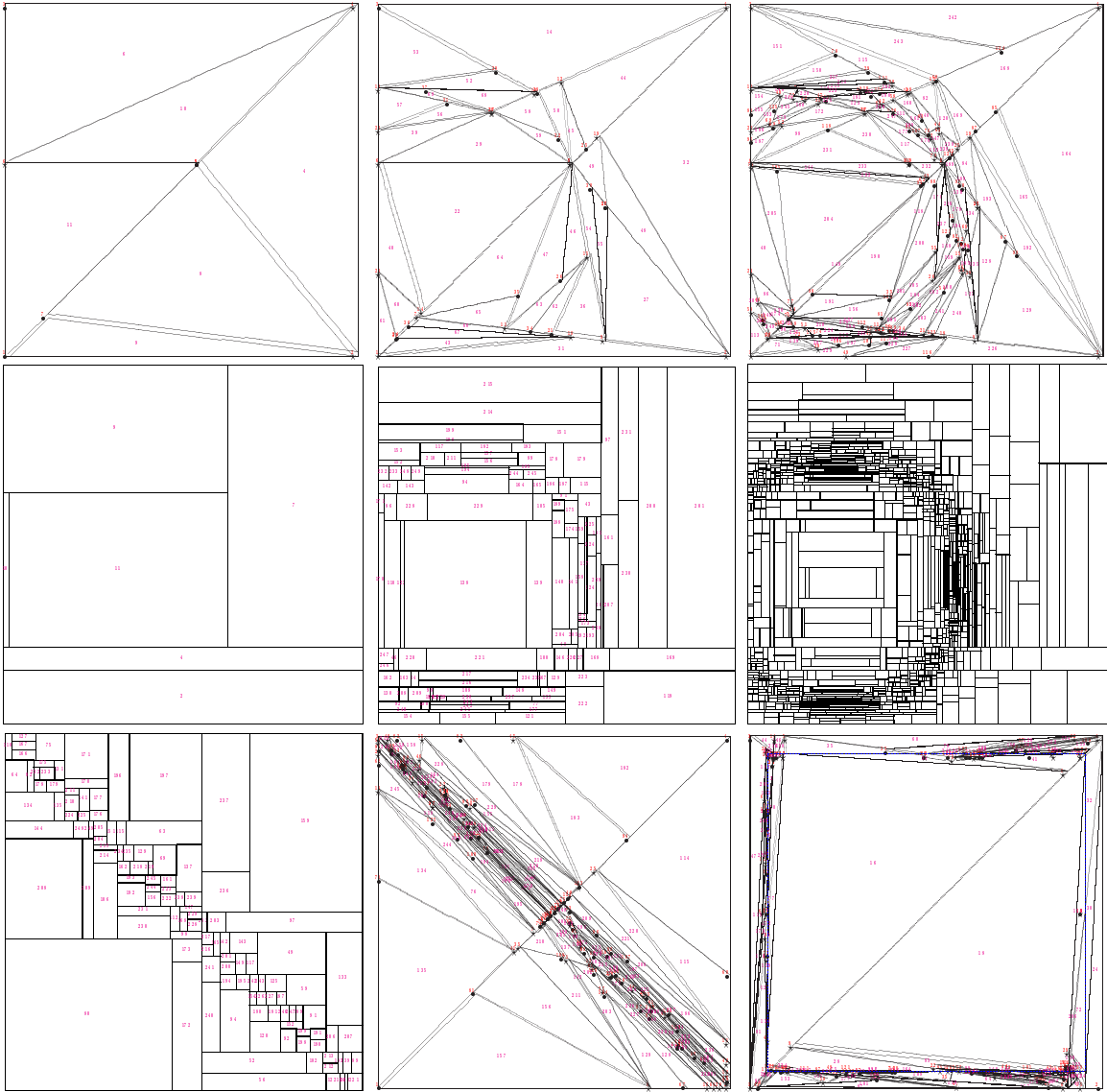
Figure 12: Examples of the 2-dimensional foam. Number of cells from 10 to 2500.

3.4.3   Concluding remarks on the cell division algorithm

The algorithm of the split of the cell is the important and most sophisticated part of the new `Foam`. Let us therefore add a couple final remarks:

30
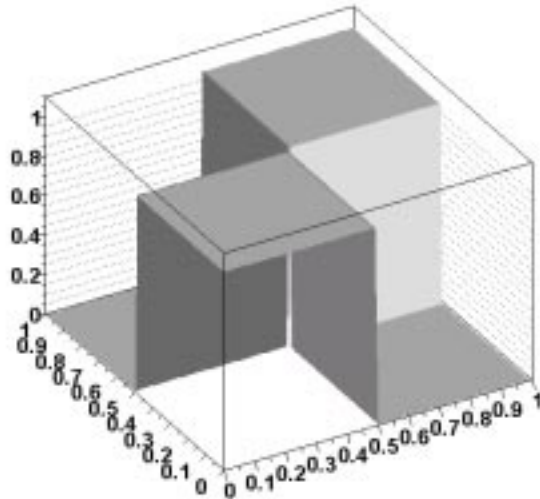
Figure 13: Example of $\rho(x)$ for which the Foam algorithm of cell division fails.

- The new, much improved procedure of the choice of the division plane is the most significant difference[22] with respect to Foam 1.x of Ref. [1].

- The choice of the edge based on the histograms for each edge makes sense if we use histograms with at least 2-4 bins and at least 100 MC events per cell. This might be a serious limitation for these $\rho(x)$, which require a lot of CPU time per function call.

Finally, in Fig. 12 we show examples of the evolution of the foam of the cell as the number of the cells grows gradually. The case of the 2 dimensions is easily visualized and we do it in Fig. 12 for triangular and rectangular cells. In the upper six plots $\rho(x)$ feature a circular ridge, in the two bottom plots is concentrated along antidiagonal $x_1 + x_2 = 1$, and the last one corresponds to $\rho(x)$ of Fig 9.

3.5   Limitations

We are aware that the present procedure of selecting next cell for the split and the procedure of defining division plane, although quite sophisticated, is not a perfect one and has certain shortcomings. Some of them can be probably removed, but some of them are inherent. In Fig. 13 we show a surprisingly simple example of a function for which our method of finding a good division point λ_{div} fails. It fails simply because both projections of $\rho(x)$ onto two edges of the rectangle are just flat and our procedure will pick up some λ_{div} randomly within $(0, 1)$, while the most economic division point is in

---

[22]I would like to thank A. Para for discussion which ignited this new development.

the middle $\lambda_{div} = 1/2$. On the other hand, although Foam algorithm gets disoriented for the first division, it will recover and correct for the "falstart" in the next divisions rather quickly. It will eliminate the two voids from its area of the interest.

Let us notice that the distribution of Fig. 13 violates maximally strongly the "principle of factorizability" $\rho(\vec{x}) = \prod_1^n \rho_i(x_i)$, the principle on which VEGAS family programs are built [3–5]. Contrary to VEGAS the problem with factorizability in Foam is not a general one, but is limited to a single cell and usually goes away after the cell split. Nevertheless, the algorithm of Foam analyzing projections on all edges in a single cell is relying on the "principle of factorizability".

Class	Short description
TFOAM_INTEGRAND	Abstract class (interface) for the Foam integrand distribution $\rho(x)$
TFVECT	Class of vectors with dynamic allocation of the components. Used in TFOAM and TFCELL
TFMATRIX	Square matrices, used for simplical geometry in the Foam
TFPARTITION	Auxiliary small class for looping over partitions and permutations
TFCELL	Class of objects presenting single cell used in TFOAM (Cartesian product of the simplex and hyperrectangle)
TFOAM	Main class of Foam. The entire MC simulator
TPSEMAR	Marsaglia et.al. random number generator [16].
TFHST	Simple class of one-dimensional histograms. Used only in the Foam version without ROOT
TFMAXWT	Monitors MC weight, measures performance of the MC run
TFDISTR	Collections of distributions $\rho(x)$ for testing Foam

Table 1: Description of C++ classes of Foam.

4   The Foam code

Presently, the C++ version of the Foam code is more advanced than the Fortran77 version. (We do not plan to develop F77 code any further.) In this section we shall describe mainly the C++ code.

The code of the Foam version 1.x was originally written in Fortran77 with popular language extensions, like long variable names etc. It was already written in an object-oriented style, as much as it was possible. In particular the main classes TFOAM and TFCELL of the present C++ version were already present in certain form. The important shortcoming of the F77 version is the lack of dynamic allocation of the memory. Otherwise, it has most of the functionality of the C++ version, see latter this section for list of limitations.

TFOAM member	Short description
float m_Version^g	Actual version of the Foam (like 2.34)
char m_Date[40]	Release date of the Foam
char m_Name[128]	Name of a given instance of the TFOAM class
int m_nDim^s	Dimension of the simplical subspace
int m_kDim^s	Dimension of the hyperrectangular subspace
int m_TotDim^g	Total dimension = m_nDim+m_kDim
int m_nCells^s	Maximum number of cells
int m_vMax	Maximum number of vertices (calculated)
int m_LastVe	Actual index of the last vertex
int m_RNmax	Maximum number of random numbers generated at once
int m_OptDrive^s	Type of optimization =1,2 for variance or maximum weight reduction
int m_OptEdge^s	Decides whether vertices are included in the cell MC exploration
int m_OptPeek^s	Type of cell peek =0,1,2 for maximum, random, random2
int m_OptOrd^s	Root cell is simplex for OptOrd=1, hyperrectangle for OptOrd=2
int m_OptMCell^s	=1 economic memory for hyperrectangles is on; =0 off
int m_Chat^s	=0,1,2 chat level in output; =1 for normal output
int m_OptDebug	=1, additional histogram (dip-switch)
int m_OptCu1st	=1, numbering starts with hyperrectangle (dip-switch)
int m_OptRej	=0 for weighted events; =1 for unweighted events in MC generation
int m_nBin^s	No. of bins in edge-histogram for cell MC exploration
int m_nSampl^s	No. of MC events, when dividing (exploring) cell
int m_EvPerBin^s	Maximum number of effective ($w = 1$) events per bin
int m_nProj	Number of projection edges (calculated)

Table 2: Data members of the TFOAM class. Associated setters and getters marked as super-
scripts s and g.

4.1 Description of C++ classes

In Table 1 we list all classes of the Foam package. The main class is TFOAM, which is
the MC simulator itself. It is served by the class TFCELL of the cell objects, and three
auxiliary classes TFVECT, TFMATRIX and TFPARTITION. The other classes are not related
directly to Foam algorithm – they are utilities used by Foam: random number generator
class TPSEMAR [16] and the histograming class TFHIST. The class TFDIST provides a
menu of the distributions for testing Foam. In the following we shall describe in a more
details the key classes TFOAM and TFCELL.

4.2 TFOAM class

TFOAM is the main class. Every new instance of this class (properly initialized) is another
independent Foam event generator. In Tables 2 and 3 we provide full list of data members
of the class TFOAM and their short description. As seen in these tables, we have added
prefix "m_" to all names of the data members, such that in the code they differ visually

TFOAM member	Short description
Provision for multi-branching	
int *m_MaskDiv	![m_nProj] Dynamic mask for cell division
int *m_InhiDiv	![m_kDim] Flags inhibiting cell division, h-rectang. subspace
int m_OptPRD	Option switch for predefined division, for quick check
TFVECT **m_XdivPRD	!Lists of division values encoded in one vector per direction
Geometry of cells	
int m_NoAct	Number of active cells
int m_LastCe	Index of the last cell
TFCELL **m_Cells	[m_nCells] Array of ALL cells
TFVECT **m_VerX	[m_vMax] Array of pointers to vertex vectors
Monte Carlo generation	
double m_MaxWtRej;	Maximum weight in rejection for getting $w = 1$ events
TFMAXWT *m_MCMonit;	Monitor of the MC weight for measuring MC efficiency
TFCELL **m_CellsAct	!Array of pointers to active cells, constructed at the end of foam build-up
double *m_PrimAcu	!Array of cumulative $\sum_{i=1}^{k} R'_i$, for cell index generation
TObjArray *m_HistEdg	Histograms of w, one for each edge, with ROOT
TObjArray *m_HistDbg	Histograms for debug (m_OptDebug=1), with ROOT
TH1D *m_HistWt;	Histograms of MC weight, with ROOT
TFHST **m_HistEdg	Array of pointers to histograms, without ROOT
TFHST *m_HistWt;	Histograms of MC weight, without ROOT
double *m_MCvect^g	[m_TotDim] Generated MC vector for the outside user
double m_MCwt^g	MC weight
double *m_Rvec	[m_RNmax] Random number vector from r.n. generator, up to m_TotDim+1 maximum elements
Externals	
TFOAM_INTEGRAND *m_Rho$^g s$	The distribution ρ to be generated/integrated
TPSEMAR *m_PseRan$^g s$	Generator of the uniform pseudorandom numbers
Statistics and MC results	
long m_nCalls^g	Number of function calls
long m_nEffev	Total no. of effective $w = 1$ events in build-up
double m_SumWt, m_SumWt2	Sum of weight w and squares w^2
double m_NevGen	No. of MC events
double m_WtMax, m_WtMin	Maximum/Minimum weight (absolute)
double m_Prime^g	Primary integral R', ($R = R\langle w\rangle$)
double m_MCresult	True integral R from the cell exploration MC
double m_MCerror	and its error
Working space for cell exploration	
double *m_Lambda	[m_nDim] Internal parameters of the simplex: $\sum \lambda_i < 1$
double *m_Alpha	[m_kDim] Internal parameters of the h-rectang.: $0 < \alpha_i < 1$

Table 3: Data members of the class TFOAM. Cont.

`TFOAM` method	Short description
Constructors and destructors	
TFOAM()	Default constructor (for ROOT streamer)
TFOAM(const char*)	User constructor
$\tilde{\text{T}}$FOAM()	Explicit destructor
TFOAM(const TFOAM&)	Copy Constructor NOT USED
TFOAM& operator=(const TFOAM&)	Substitution NOT USED
Initialization, foam build-up	
void Initialize(TPSEMAR*, TFOAM_INTEGRAND*)	Initialization, allocation of memory and the foam build up
void InitVertices(void)	Initializes first vertices of the root cell
void InitCells(void)	Initializes first $n!$ cells in h-rect. root cell
void Grow(void)	Adds new cells to foam, until buffer is full
int Divide(TFCELL *)	Divides cell into two daughters
void Explore(TFCELL *Cell)	MC exploration of cell main subprogram
void Carver(int&,double&,double&)	Determines the best edge, $w_{\max}$-reduction
void Varedu(double[ ],int&,double&,double&)	Determines the best edge, σ-reduction
long PeekMax(void)	Chooses one active cell, used in `Grow`
TFCELL* PeekRan(void)	Chooses randomly one active cell, in `Grow`
void MakeLambda(void)	Generates random point inside simplex
void MakeAlpha(void)	Generates rand. point inside h-rectangle
int CellFill(int, TFCELL*, int*,TFVECT*,TFVECT*)	Fills next cell and return its index
void MakeActiveList(void)	Creates table of all active cells
Generation	
void MakeEvent(void)	Makes (generates) single MC event
void GetMCvect(double *)	Provides generated random MC vector
void GetMCwt(double &)	Provides MC weight
double MCgenerate(double *MCvect)	All the above in single method
void GenerCell(TFCELL *&)	Chooses one cell with probability $\sim R'_j$
void GenerCel2(TFCELL *&)	Chooses one cell with probability $\sim R'_j$
Finalization, reinitialization	
void Finalize(double&, double&)	Prints summary of MC integration
void GetIntegMC(double&, double&)	Provides MC integral
void GetIntNorm(double&, double&)	Provides normalization
void GetWtParams(const double, double&, double&, double&)	Provides MC weight parameters
void LinkCells(void)	Restores pointers after restoring from disk
Debug	
void CheckAll(const int)	Checks correctness of the data structure
void PrintCells(void)	Prints all cells
void PrintVertices(void)	Prints all vertices
void LaTexPlot2dim(char*)	Makes LaTeX file for drawing 2-dim. foam
void RootPlot2dim(char*)	Makes C++ code for drawing 2-dim. foam

Table 4: Methods of `TFOAM` class.

from the other variables (it is a recommended practice in the C++ coding).

Generally, one may notices that many data members, could be declared (and allocated) as the local variables in the procedures, instead of being data members. For example, vector m_MCvect transporting random numbers out from random number generator m_PseMar could be declared locally at every place where m_PseMar is called. We opted for a more "static" structure of the data, with more than necessary of the data members in the class, at the expense of the human readability of the code, in order to: (a) facilitate the implementation persistency with ROOT (b) gain in the execution speed (c) facilitate the translation to other languages.

Most of the methods (procedures) of the class TFOAM are listed in the Table 4. We omitted in this table "setters" and "getters", which provide access to some data members, and simple inline functions, like sqr for squaring double variable. Data members which are served by the setters and getters are marked in Tables 2 and 3 by the superscripts "*s*" or/and "*g*".

Let us now characterize briefly the role of most important methods of the class TFOAM in the Foam algorithm.
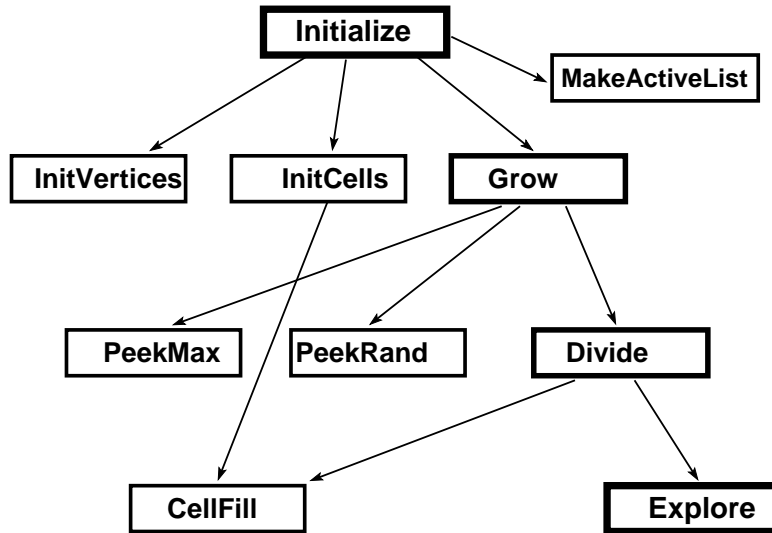
Figure 14: Calling sequence of the Foam procedures during the foam build-up (initialization).

4.2.1   Procedures for Foam initialization and foam build-up

The constructor TFOAM(const char*) is for creating an object of the class TFOAM. Its parameter is the name given by the user to an object. The principal role of this constructor is to initialize data members to its default values – no memory allocation is done at this stage. After resetting all kind of steering parameters of the Foam to preferred values (using

setters) user is calling `Initialize` method, which builds up the foam of cells. The two methods `InitVertices` and `InitCells` allocate arrays of vertices and cells (pointers) with empty cells. The empty cells are allocated/filled using `CellFill`. Next comes procedure `Grow` which loops over cells, picking up most promising cell for the split, either randomly using `PeekRand` or deterministically using `Peekmax`. The chosen cell is split using `Divide`. It is, however, the procedure `Explore` called by `Divide` (and by `InitCells` for the root cell) which does the most important job in the foam build-up – it performs a small MC run for each newly allocated daughter cell. It calculates how profitable will be the future split of the cell and defines the optimal cell division geometry with the help of `Carver` or `Varedu` procedures, for maximum weight or variance optimization respectively. All essential results of the exploration are written into the explored cell object. At the very end of the foam build-up `MakeActiveList` is invoked to create list of pointers to all active cells, for the purpose of quick access during the MC generation. The procedure `Explore` uses two procedures `MakeLambda` and `MakeAlpha`, which generate randomly (uniformly) coordinates of the MC points inside a given cell. The above sequence of the procedure calls is depicted in Fig. 14.

4.2.2   Procedures for MC generation

The MC generation of a single MC event is done by invoking `MakeEvent`, which is choosing randomly a cell with the help of procedure[23] `GenerCell2` and, next, the internal coordinates of the point within the cell using `MakeLambda` and/or `MakeAlpha`. The absolute coordinates of the MC event are calculated and stored in the data member double-precision vector `m_MCvect`. MC weight is calculated using an external procedure providing the density distribution $\rho(x)$, which is represented by the pointer `m_Rho`. A class to which the object `m_Rho` belongs must inherit from the abstract class `TFOAM_INTEGRAND`. The MC event (double-precision vector) and its weight is available through getters `GetMCvect` and `GetMCwt`. Note that the variables of the hyperrectangular subspace come first in the `m_MCvect`, before variables of the simplical subspace.

User may alternatively call `MCgenerate` which invokes `MakeEvent` and provides MC event and its weight, all at the same time.

4.2.3   Procedures for finalization and debug

The use of the method `Finalize` is not mandatory. It prints statistics and calculates the estimate of the integral using the average weight from the MC run. The amount of printed information depends on the values of `m_chat`. For the normalization of the plots and integrals user needs to know the exact value of $R' = \int \rho'(x)dx$, which is provided by the method `GetIntNorm` or `Finalize`. The actual value of the integrand from the MC series is provided by `GetIntegMC`. Note that for the convenience of the user `GetIntNorm` provides R' or MC estimate of $R = \int \rho(x)dx$, depending on whether MC run was with weighted events or $w = 1$ events.

---

[23]Method `GenerCell1` exists, but is not used.

Another useful finalization frocedure

```
GetWtParams(const double eps, double &AveWt, double &WtMax, double &Sigma)
```

provides three parameters characterizing the MC weight distribution: the average weight `AveWt`, the "inteligent" maximum weight `WtMax`= $w_{\max}^{\varepsilon}$ for a given value of `eps`= ε (see Sect. 6 for its definition) and the variance `sigma`= σ. In particular, in case of unweighted events, $w_{\max}^{\varepsilon}$ can be used as an input for the next MC run.

The `Foam` program is invoking procedure is `CheckAll`, which checks correctness of the pointers in the doubly linked tree of cells (this can take time for large N_c). It can sometimes be useful for the debugging purpose. Another two methods `PrintVertices` and `PrintCells` can be used at any stage of the calculation in order to print the list of all cells and vertices. In the case of the two-dimensions there is a possibility to view the geometry of the cells with a 2-dimensional plot, which is either a LaTeX file produced by `LaTexPlot2dim`, or a ROOT file produced by `RootPlot2dim`.

TCELL member	Short description
"Static" members, the same for all cells!	
short int m_kDim	Dimension of hyperrectangular subspace
short int m_nDim	Dimension of simplical subspace
short int m_OptMCell	Option of economic memory for usage (hyperrectangular subspace)
short int m_OptCu1st	=1, Numbering of dims starts with hyperrectangles; =0 simplices
int m_nVert	No. of vertices in the simplex = m_nDim+1
TFCELL **m_Cell0	! Pointer of the root cell
TFVECT **m_Vert0	! Pointer of the vertex list
Linked tree organization	
int m_Serial	Serial number (index in m_Cell0)
int m_Status	Status (active or inactive)
int m_Parent	Pointer to parent cell
int m_Daught0	Pointer to daughter 1
int m_Daught1	Pointer to daughter 2
The best split geometry from the MC exploration	
double m_Xdiv	Factor λ of the cell split
int m_Best	The best edge candidate for the cell split
Integrals of all kinds	
double m_Volume	Cartesian Volume of this cell
double m_Integral	Integral over cell (estimate from exploration)
double m_Drive	Driver integral R_{loss} for cell build-up
double m_Primary	Primary integral R' for MC generation
Geometry of the cell	
int *m_Verts	[m_nVert] Pointer to array of vertices in simplical subspace
TFVECT *m_Posi	Pointer to position vector, hyperrectangular subspace
TFVECT *m_Size	Pointer to size vector, hyperrectangular subspace

Table 5: Data members of the class TFCELL.

4.3 TFCELL class

TFCELL is the important class of objects representing single cell of the foam. Data members of the class are listed in Table 5.

Most of the methods of the TFCELL class are setters and getters. The non-trivial methods are GetHcub and GetHSize, which calculate the absolute position and size of hyperrectangles in the algorithm of Section 2.6 and MakeVolume which calculates the Cartesian volume of the cell. In the simplical subspace volume is a determinant of a square matrix of the class TFMATRIX.

4.4 Persistency with help of ROOT

C++ language does not provide any built-in mechanism for persistency of the classes. For this purpose we use ROOT package [10], with help of its "automatic streamers". ROOT is a useful C++ library for histograming, organizing large database of identical objects of the type used in high energy physics experiments. It also provides an efficient input/output, with compressing capabilities.

Providing full persistency of any type of C++ classes, preserving all structure of the pointers is probably impossible to realize in general. ROOT can do it, even for pointers, provided the code is organized in a special way. (No static variable, explicit integer indices instead of pointers in some places). As a whole, this solution is not very elegant, but relatively simple and works correctly. In Tables 2 and 3 we have in the beginning of the description certain characteristic marks which are directives for persistency mechanism of ROOT, see manual of ROOT [10] for more details.

One has to remember, when reading TFOAM class object from the disk, that the method LinkCells() has to be invoked in order to reconstruct fully all pointers in the doubly linked tree of cells. Moreover, any object of the class TFOAM restored form the disk file will have its internal object for the random number generator and distribution function. There is a method which provides access (pointer) to these objects, if necessary. The relevant fragment of the code may look as follows:

```
 TPSEMAR *RNGen= FoamX->GetPseRan();        //get pointer of RN generator
 TFDISTR *RHO  = (TFDISTR*)FoamX->GetRho(); //get pointer of distribution
```

It might be useful if, for instance, we want to reinitialize the random number generator used by the TFOAM class object, which has been read from the disk-file.

Foam can be used with or without ROOT. In the code all parts of the code dependent on ROOT enclosed in the pair of preprocessor commands #ifdef ROOT_DEF ... #endif, where ROOT_DEF variable is defined centrally in the header file ROOT_DEF.h. Eliminating ROOT requires removing this variable and modifying makefile accordingly (the TFHST class has to be linked). Version without ROOT does not feature persistency, and is employing its own simple histograming class TFHST instead of the ROOT class TH1D. ROOT helps also to create documentation of the Foam in the html format. We recommend to use a version tied up with the ROOT.

4.5   Fortran77 version and its limitations

We also provide users with the Fortran77 versions of the Foam. They are two of them at the development level 2.02 (May 2001) of the algorithm. First one, in which cells can be simplical, hyperrectangular and the Cartesian product of the two. This version is limited to dimension five for the simplical subspace and not very useful for large dimensions ($n \geq 5$) in the hyperrectangular space, because it does not feature the memory saving algorithm of Section 2.6. Another version called `MCell` (standing for Mega-Cell) features only hyperrectangular cells, on the other hand, it includes the memory saving algorithm described in Section 2.6. We recommend the reader to use the version `MCell`.

Both these version cannot have dynamic memory allocation; they have a maximum dimensions of the integration/simulation subspaces (simplical and hyperrectangular) hardcoded in the source code. Any change of these maximum dimensions requires recompilation of the code.

Present versions in Fortran77 are substantially improved with respect the original version of ref. [1]. For the option of minimizing the maximum weight they have exactly the same algorithm (of the cell split) as the C++ version. They feature, however, an older more primitive version of the algorithm of finding the best cell division for the variance reduction.

The structure of the programs, naming of procedures and variables, configuration parameters and their meaning are very similar in F77 and C++ versions. Some differences in the usage will be indicated in the next Sections.

4.6   Future development

In the following we indicate some of the possible future developments of the `Foam` package. As already indicated we do not plan to develope Fortran77 version any further. On the other hand, it would be interesting to upgrade the existing `Foam` version 1.x to in JAVA to the level of the present version 2.x.

As for the C++ version, it would be a logical development to derive class of pseudo-random number generators `TPSEMAR` from the common abstract class, and in this way to define a universal interface for a library of the number generators. We intend to collect library of a few random number generators with a universal interface (or find one) for the use in `Foam` and applications based on it.

Concerning version of `Foam` adapted to parallel processing, as in Refs. [5, 7–9], we do not have plans in this direction in the immediate future. Here, we have of course in mind the use of the true CPU parallelism in the foam of cells build-up. One has to remember that in the high energy physics applications, which are our main objective, the foam of cell build-up will be always a tiny fraction of the total CPU time. The main fraction will be the subsequent MC simulation in which, as the vast experience with the PC-farms in CERN and FNAL shows, one may organize the MC simulation with the low-level of the parallelism, with many simulators started with different random seeds, running in parallel but not communicating – another specialized job is combining all results at the very end

of the run. However, the first practical examples of the true parallelism in the massive MC simulation for the purpose of the high energy physics experiments has already appeared recently [17].

As already stressed, the main algorithm of `Foam` is already rather stable and the main emphasis in its the future development will be on the effort of making it more user friendly, and better adapted to the use as a part of bigger MC projects. In particular its provisions for multibranching will become more sophisticated, as more feedback comes from the real life applications.

5 Usage of the `Foam`

5.1 `Foam` distribution directory of the C++ version

The `Foam` package is distributed together with the demonstration main programs and some utilities in form of about 20 files in a single UNIX directory `FOAM-export-v2.05`. Demonstration runs can be executed using standard `make` commands as follows:

```
make Demo-run
make DemoPers
make Demo-map
make DemoNR-run
```

The essential fragments of the output form `make Demo-run` are shown in Appendix B. The compilation and linking procedure is encoded in the `Makefile`, which has to be checked by the user if it conforms the local operating system. In particular, if ROOT is used, then certain paths and environmental variables in the `Makefile` have to be adjusted. The use of the ROOT is decided by the presence of the variable `ROOT_DEF` in `ROOT_DEF.h` file. Without ROOT user should execute `make DemoNR-run`.

The essential part of the `Foam`, that is class `TFOAM`, `TFCELL` and a few auxiliary classes are located in the files `TFOAM.cxx` and `TFOAM.h`. This is the "core" of the `Foam` source. The source code of the other utility classes `TFHST`, `TFMAXWT`, `TPSEMAR` and `TFDISTR` are in separate files. The main programs are in files `Demo.cxx` and `DemoPers.cxx`. They should serve as a useful templates for the user's own application based on Foam.

There is also one Fortran77 source code `circe2.f`[24], which contains certain testing distribution linked to `TFDISTR`. The `Makefile` provides, therefore, also a useful example of linking C++ and F77 codes.

There are also two output files `output-Demo.linux` and `output-DemoNR.linux`, which the reader may use to check whether he is able to reproduce these benchmark output results.

---

[24]We thank Thorsten Ohl for providing us a preliminary version of this code [18].

5.2   Simple example of an application

The very simple example of the use of the `Foam` may look as follows:

```
//    *** Initialization ***
double MCwt;
TFDISTR *Density1 = new TFDISTR(FunType); // Create integrand function
TPSEMAR *PseRan   = new TPSEMAR();        // Create random numb. generator
TFOAM   *FoamX    = new TFOAM("FoamX");   // Create Simulator
FoamX->SetkDim( 3);                       // Set dimension, h-rect.
FoamX->Initialize(PseRan,  Density1 );    // Initialize simulator
//    *** MC Generation ***
TFHST *hst_Wt = new TFHST(0.0,1.25, 25);  // Create weight histogram
double *MCvect =new double[3];            // Monte Carlo event
for(long loop=0; loop<1000000; loop++){
   MCwt = FoamX->MCgenerate(double *MCvect); // Generate MC event
   hst_Wt->Fill(MCwt,1.0);                   // Fill weight histogram
}
//    *** Finalization ***
double IntNorm, Errel;
FoamX->Finalize( IntNorm, Errel);  // Print statistics, get normalization
double MCresult, MCerror, AveWt, WtMax, Sigma;
FoamX->GetIntegMC( MCresult, MCerror);        // get MC integral
double eps = 0.0005;
FoamX->GetWtParams(eps, AveWt, WtMax, Sigma); // get MC wt parameters
hst_Wt->Print();                      // Print weight histogram
```

The user has to provide the distribution function belonging to the class which has to inherit from the following abstract class:

```
class TFOAM_INTEGRAND{ // Abstract class of distributios for Foam
 public:
  TFOAM_INTEGRAND() { };
  virtual ~TFOAM_INTEGRAND() { };
  virtual double Density(int ndim, double*) = 0;
};
```

In the above example the distribution `*Density1` belongs to the class `TFDISTR`, which is provided in the `Foam` distribution directory.

5.3   Configuring the `Foam`

`Foam` has fourteen principal configuration parameters plus parameters inhibiting and/or predefining division geometry in the cell split.

Param.	Value	Meaning
nDim	0*	Dimension of simplical sub-space
kDim	0*	Dimension of hyperrectangular sub-space
nCells	1000*	Maximum number of Cells,
nSampl	200*	No. of MC events in the cell MC exploration
nBin	8*	No. of bins in edge-histogram in cell exploration
OptRej	0*	OptRej=0, weighted; =1, $w = 1$ MC events
OptDrive	2*	Maximum weight reduction,
	1	or variance reduction
OptPeek	0*	Next cell for split with maximum R'_I (PeekMax),
	1	or randomly with probability $\sim R'_I$ (PeekRan)
OptEdge	0*	Vertices are NOT included in the cell MC exploration,
	1	or vertices are included in the cell MC exploration
OptOrd	0*	Root cell is hyperrectangular in simplical subspace
	1	or root cell is simplex in simplical subspace
OptMCell	1*	Economic memory algorithm in hyperrectangular subspace is ON,
	0	or economic memory algorithm in hyperrectangular subspace is OFF
EvPerBin	25*	Maximum no. of eff $w = 1$ events/bin,
	0	or counting of no. eff events/bin is inactive
Chat	1*	=0,1,2 is the "chat level" in the standard output
MaxWtRej	1.1*	Maximum weight used to get $w = 1$ MC events

Table 6: Fourteen principal configuration parameters and switches of the Foam program. The default values are marked with the star superscript.

5.3.1 Principal configuration parameters

All of the principal parameters listed in Table 6 are set to meaningful default values (see Table), hence, the beginning user may stay ignorant about their role for some time, and learn gradually later on how to exploit them in order to improve the efficiency of the Foam. All these parameters are data members of the TFOAM class, see Table. 2. If the user wants to redefine all of them, then the relevant piece of code will look as follows:

```
FoamX->SetnDim(    nDim);
FoamX->SetkDim(    kDim);
FoamX->SetnCells(  nCells);
FoamX->SetnSampl(  nSampl);
FoamX->SetnBin(    nBin);
FoamX->SetOptRej(  OptRej);
FoamX->SetOptDrive( OptDrive);
FoamX->SetOptPeek(  OptPeek);
FoamX->SetOptEdge(  OptEdge);
FoamX->SetOptOrd(   OptOrd);
FoamX->SetOptMCell( OptMCell);
FoamX->SetEvPerBin( EvPerBin);
```

```
FoamX->SetMaxWtRej( MaxWtRej);
FoamX->SetChat(     Chat);
```

In practical applications one will redefine a subset of them. The minimum requirement is that the user sets nonzero value of `nDim` or `kDim` such that the total dimension `nDim+kDim` is a non-zero positive number.

5.3.2   Inhibiting cell division in certain directions

If user of `Foam` decides to inhibit division in certain variable in the hyperrectangular subspace, then it can be done with the method `SetInhiDiv(int iDim, int InhiDiv)` of the class `TFOAM`, where `iDim` is the dimension index for which inhibition is done and `InhiDiv` is the inhibition tag. This method should be used before invoking `Initialize`, after setting `nDim` and/or `kDim`. The relevant code may look as follows:

```
FoamX->SetInhiDiv(0, 1); //Inhibit division of x_1
FoamX->SetInhiDiv(1, 1); //Inhibit division of x_2
```

The allowed values are `InhiDiv=0,1` and the default value is `InhiDiv=0`. Note that numbering of dimensions with `iDim` starts from zero and variables of the hyperrectangular subspace always come first, before the simplical ones.

5.3.3   Setting predefined cell division geometry

We may predefine divisions of the root cell in certain variable in the hyperrectangular subspace using method `SetXdivPRD(int iDim, int len, double xDiv[])`. The relevant piece of the user code may look as follows:

```
double xDiv[3];
xDiv[0]=0.30; xDiv[1]=0.40; xDiv[2]=0.65;
FoamX->SetXdivPRD(0, 3, xDiv);
```

Again, this should be done before invoking `Initialize`, after setting `nDim` and/or `kDim`.

5.4   Persistency

Persistency of the `Foam` classes is arranged using "default streamers" of the ROOT [10] package. Writing `TFOAM` class object into a disk file `rmain.root` can be done with the single `Write` as follows:

```
TFile RootFile("rmain.root","RECREATE","histograms");
...
FoamX->Write("FoamX"); //Writing Foam on the disk, TESTING PERSISTENCY!
...
RootFile.Write();
RootFile.Close();
```

The instruction `FoamX->Write("FoamX")` can be put at any place of the code after the instruction `FoamX->Initialize(...)`, see example of the user code shown in Section 5.2.

Next, in another program `TFOAM` class object can be read from the disk file `rmain.root` as follows:

```
TFile fileA("rmain.root");  // connect disk file
fileA.cd();
fileA.ls();                        // optional printout
fileA.Map();                       // optional printout
fileA.ShowStreamerInfo();          // optional printout
fileA.GetListOfKeys()->Print();    // optional printout
TFOAM  *FoamX = (TFOAM*)fileA.Get("FoamX"); // find object
FoamX->LinkCells();  // restore pointers of the binary tree of cells
FoamX->CheckAll(1);                // optional x-check of pointers
```

and at this point `FoamX` object is ready to generate MC events, as in the MC generation part of the code shown in Section 5.2.

5.5   Fortran77 versions

The distribution directory `FoamF77-2.02-export` contains `README` file, two demonstration main programs `DemoFoam.f` and `DemoMCell.f` to be compiled and run with the help of commands

```
make DemoFoam
make DemoMCell
```

encoded in the `Makefile`. The outputs from the above runs can be compared with the benchmark outputs `output-DemoFoam-linux` and `output-DemoMCell-linux`.

The basic `Foam` source files are: `FoamA.f` with header file `FoamA.h` and `MCellA.f` with header file `MCellA.h`.

For the description of the input (configuration) parameters see comments in FoamA.f and MCellA.f respectively. The names of the configuration variables are the same as in C++ version, except `nCells` which is renamed to `nBuff`. Their values and the meaning are the same.

Demonstration main programs `DemoFoam.f` and `DemoMCell.f` can serve as templates for the user application programs.

The testing main program uses histograming package `GLK` of the `KKMC` program [13], which user may replace with any other histograming package.

6   Numerical studies and example applications

In the following subsection we examine MC efficiency of the `Foam` in a series of numerical exercises. In some of them we shall also show examples of the `Foam` application with the distributions relevant for everyday practice in the high energy physics.

nDim	kDim	nCalls	nCells	nSampl	$w^\varepsilon_{\max}/\langle w\rangle$	$\sigma/\langle w\rangle$	$\Delta_{statist.}R$	R
0	1	203719	1000	333	0.99148	0.014585	1.031e-05	0.99999782
0	1	206192	1000	1000	0.99147	0.014752	1.043e-05	0.99999962
0	1	206192	1000	3333	0.99147	0.014752	1.043e-05	0.99999962
0	1	206192	1000	10000	0.99147	0.014752	1.043e-05	0.99999962
0	1	206192	1000	33333	0.99147	0.014752	1.043e-05	0.99999962
0	3	275421	1000	333	0.50538	0.54088	0.000382	0.99986832
0	3	435112	1000	1000	0.50886	0.54033	0.000382	1.00017104
0	3	663493	1000	3333	0.49922	0.55721	0.000394	0.99934710
0	3	834094	1000	10000	0.50359	0.54674	0.000386	1.00056316
0	3	1015157	1000	33333	0.51091	0.54035	0.000382	0.99983999
0	3	2312759	10000	333	0.72346	0.27758	0.000196	0.99977045
0	3	2675820	10000	1000	0.72677	0.27504	0.000194	0.99995080
0	3	3054404	10000	3333	0.72199	0.27710	0.000195	1.00013270
0	3	3333479	10000	10000	0.72200	0.27720	0.000196	1.00008994
0	3	3575366	10000	33333	0.72243	0.27786	0.000196	0.99997875
0	4	3825046	10000	1000	0.50363	0.51168	0.000361	1.00013082
0	4	6559430	10000	10000	0.50297	0.51001	0.000360	0.99960319
2	2	4493961	10000	1000	0.43076	0.63185	0.000446	1.00072564
2	2	9374351	10000	10000	0.44922	0.60669	0.000429	1.00013171
4	0	6642202	10000	1000	0.21029	1.19420	0.000844	1.00072248
4	0	12337748	10000	10000	0.20817	1.20067	0.000849	1.00020405
0	6	2311881	1000	3333	0.04199	2.12091	0.001499	0.99856206
0	6	5542146	1000	10000	0.03847	2.38588	0.001687	0.99912901
0	6	12844256	1000	33333	0.03279	2.61028	0.001845	0.99799089
0	6	12737314	10000	3333	0.15385	1.15211	0.000814	1.00039754
0	6	24134694	10000	10000	0.15313	1.19596	0.000845	0.99945766
0	6	42827237	10000	33333	0.14168	1.22627	0.000867	0.99954178
0	6	42808972	100000	1000	0.30910	0.71250	0.000503	0.99972833
0	6	61803017	100000	3333	0.30805	0.71462	0.000505	1.00002674
0	6	92531875	100000	10000	0.30905	0.71423	0.000505	0.99985093
0	9	78325890	100000	1000	0.03718	1.64608	0.001163	0.99367339
0	9	167710365	100000	3333	0.05247	1.73063	0.001223	1.00109792
0	9	353943409	100000	10000	0.05196	1.80538	0.001276	1.00196909
0	9	272162624	400000	1000	0.08490	1.30193	0.000920	1.00065580
0	9	495260998	400000	3333	0.09174	1.35307	0.000956	0.99884358
0	9	924011087	400000	10000	0.08853	1.38579	0.000979	1.00052122
0	12	261911066	100000	3333	0	5.83954	0.004129	0.97304842
0	12	671460574	100000	10000	0.00640	3.85823	0.002728	0.98878698
0	12	913072065	400000	3333	0.01285	2.73991	0.001937	0.98688299
0	12	2117963809	400000	10000	0.01235	2.92642	0.002069	0.99301117

Table 7: Numerical results of Foam with the maximum weight reduction. Variable nCalls is the total number of the function calls in the foam build-up.

nDim	kDim	nCalls	nCells	nSampl	$w^\varepsilon_{\max}/\langle w\rangle$	$\sigma/\langle w\rangle$	$\Delta_{statist.}R$	R
0	4	3855289	10000	1000	0.27659	0.31944	0.000225	1.00025027
0	4	7760907	10000	10000	0.30313	0.31483	0.000222	0.99978048
2	2	4589024	10000	1000	0.23086	0.38050	0.000269	0.99967167
2	2	8696153	10000	10000	0.24696	0.37153	0.000262	0.99959278
4	0	6157799	10000	1000	0.08498	0.92314	0.000652	1.00006553
4	0	10547749	10000	10000	0.09881	0.89859	0.000635	1.00024727

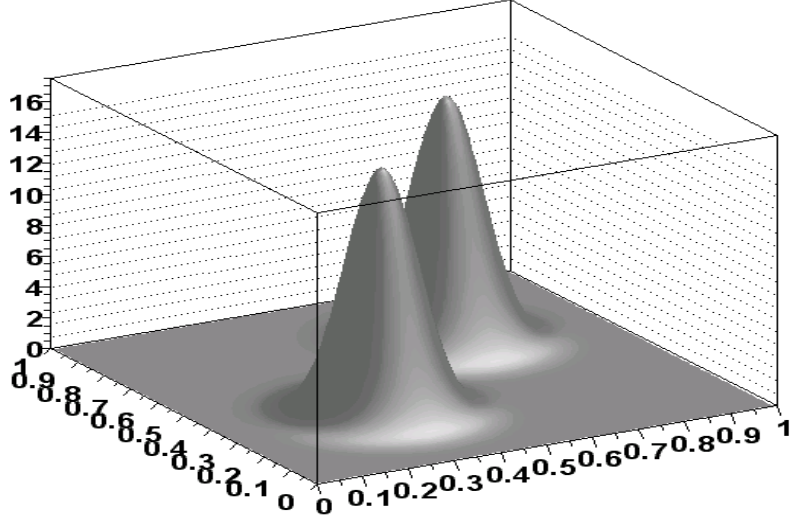Table 8: Numerical results of Foam with the variance reduction.

Figure 15: Testing distribution $\rho_{camel}(x)$ of Ref. [3] in two dimensions.

6.1 Dependence of the Foam efficiency on the configuration parameters

As a first numerical exercise we examine the dependence of the Foam efficiency on the most important (input) configuration parameters, including the dimension of the space.

In Table 7 we collect results from many MC runs for various dimensions, number of cells and number of MC events in single cell exploration, varying also the type of the cells. We use always the same test distribution $\rho_{camel}(x)$ of Ref. [3] which features two relatively narrow gaussian peaks placed on the diagonal. The 2-dimensional version of this distribution is shown in Fig. 15. We have used the non-default values nBin=4 and EvPerBin=50 and the default values for the other configuration parameters. In this table all tests were done for the default option of reduction of the maximum weight, OptDrive=2 (for results with OptDrive=1 see next table). The efficiency $w^\varepsilon_{\max}/\langle w\rangle$ is calculated using

maximum weight $w_{\max}^\varepsilon$ defined as in[25] Ref. [1] for $\varepsilon = 0.0005$. The maximum weight $w_{\max}^\varepsilon$ is calculated with help of a small auxiliary class `TFMAXWT`.

In Table 7 the efficiency of the MC run measured in terms of $w_{\max}/\langle w \rangle$ and $\sigma/\langle w \rangle$. The value of the integral $R \pm \Delta_{statist.}R$, shown in last four columns, was obtained from the MC run in which the total number of the MC events was 2×10^6. The value of the integral R is well known, it is equal one, within 10^{-5}.

The following observations based on the results of Table 7 can be made:

- Looking at the results for total dimension $n = 4$ we see that the hyperrectangular cells clearly provide better MC efficiency than simlical ones. All other results are for hyperrectangular cells.

- All of results are consistent with the observation that the MC efficiency depends critically on the number of cells. In particular, see results for $n = 6$, the increase of `nSamp` (no. of MC events in cell exploration) beyond certain value does not improve the efficiency at all.

- In the case of the very inefficient `Foam`, see $n = 12$ with $\sigma/\langle w \rangle \sim 6$, the estimate of the MC statistical error can be misleading. We see an indication, that one should not trust runs with $\sigma/\langle w \rangle > 3$.

- For this particular testing function the dimension $n = 12$ requires a minimum of 400k cells and the resulting efficiency of order of 1% is barely acceptable[26].

In Table 8 we repeat the exercise of Table 7 for the option of the variance reduction `OptDrive=1` at four dimensions. As compared to Table 7 we see net improvement in the variance and deterioration of the $w_{\max}^\varepsilon$. This agrees with the expectations.

Functions at 2-dimens.	Foam 1.01	Simpl.	H-Rect.	VEGAS
$\rho_a(x)$ (diagonal ridge)	0.93	0.93	0.86	0.03
$\rho_b(x)$ (circular ridge)	0.82	0.82	0.82	0.16
$\rho_c(x)$ (edge of square)	0.57	1.00	1.00	0.53
Functions at 3-dimens.	Foam 1.01	Simpl.	H-Rect.	VEGAS
$\rho_a(x)$ (thin diagonal)	0.67	0.74	0.66	0.002
$\rho_b(x)$ (thin sphere)	0.36	0.47	0.53	0.11
$\rho_c(x)$ (surface of cube)	0.37	0.95	1.00	0.30

Table 9: Efficiencies $\langle w \rangle / w_{\max}^\varepsilon$ for $\varepsilon = 0.0005$. Functions $\rho_x(x)$ are the same as in Ref. [1]. Results from `Foam` are for 5000 cells (2500 active cells) and cell exploration is done for a modest 200 MC events/cell.

---

[25]The ε-dependent maximum weight is defined such that events with $w > w_{\max}^\varepsilon$ contribute ε-fraction to the total integral. It is numerically more stable in the numerical evaluation than the one defined as the biggest weight in the MC run.

[26]However, we still get the correct value of the integral within 0.2%.

	k	n	N_c	N_s	N_b	$\frac{N_{eff}}{bin}$	nCall	$\frac{\sigma}{\langle w \rangle}$	$\frac{\langle w \rangle}{w^\varepsilon_{\max}}$	$R \pm \Delta R$	$\Delta R/R$
1	2	0	1K	1K	4	25	900K	1168.8	0.0	5.40726±1.99871	0.36963
1	0	2	1K	1K	4	25	169K	0.2272	0.6149	3.14121±0.00022	$7.1 \cdot 10^{-5}$
2	0	2	1K	1K	4	25	215K	0.2962	0.7754	3.14118±0.00029	$9.3 \cdot 10^{-5}$
1	0	2	5K	1K	4	25	656K	0.0639	0.8421	3.14159±0.00006	$2.0 \cdot 10^{-5}$
1	0	2	10K	1K	4	25	1174K	0.0487	0.8877	3.14156±0.00005	$1.5 \cdot 10^{-5}$
1	0	2	1K	10K	4	25	849K	0.1479	0.5920	3.14118±0.00014	$4.6 \cdot 10^{-5}$
1	0	2	5K	10K	4	25	1457K	0.0606	0.8354	3.14150±0.00006	$1.9 \cdot 10^{-5}$
1	0	2	1K	2K	8	25	621K	0.0606	0.8354	3.14195±0.00026	$8.4 \cdot 10^{-5}$
1	0	2	1K	8K	8	100	1671K	0.1048	0.6652	3.14168±0.00010	$3.3 \cdot 10^{-5}$

Table 10: Numerical results of Foam for 2-dimensional distribution of eq. (26) for $\mu = 10^{-6}$. Variation of the configuration parameters: k =kDim, n =nDim, N_c =nCells (no. of function calls), N_b =nBin, $\frac{N_{eff}}{bin}$ =EvPerBin. In first column we mark the type of the weight optimization OptDrive=1,2, for variance or maximum weight reduction. The value of the integral R and its statistical error ΔR are from MC run of $N_{MC} = 10^7$ events. $w^\varepsilon_{\max}$ is for $\varepsilon = 0.0005$. nCalls is the total number of the function calls in the foam build-up.

6.2 Comparison with Foam 1.x and classic VEGAS

In Table 9 we update the comparison of the Foam and VEGAS of Ref. [1], adding results for the new hyperrectangular option. The simplical results are now clearly improved with respect to Ref. [1], because of the better cell division algorithm. Generally, hyperrectangular cell mode provides as good efficiency as simplical one. However, one should keep in mind that Foam with hyperrectangular cells is factor two or more faster in the execution.

6.3 Example of sharply peaked distribution

In Table 10 we examine the dependence of the MC efficiency/error on the various input configuration parameters of the Foam. All these numerical results are for the distribution

$$\rho_g(x) = \frac{\mu x_2}{(x_1 + x_2 - 1)^2 + \mu^2}, \tag{26}$$

which, for $\mu = 10^{-6}$, has a very sharp ridge across the diagonal $x_1 + x_2 = 1$. This distribution is taken from Ref. [19] and is related to the photon distribution at high energy electron-positron colliders.

What can we learn from the results in Table 10? First of all, in first line, we see a spectacular failure of the Foam with rectangular cells[27]. The value of the integral is wrong by factor two and statistical error is underestimated. This illustrates the problem of the the lack of "angular mobility" of the rectangular cells indicated in Section 2.3. Rectangles are unable to align with the singularity along the diagonal. This we illustrate in the left plot of Fig. 16, for rectangular 1000 cells, where we see clearly "blind spots". In the right

---

[27]Setting the no. of the MC events in a single rectangle exploration to 10^4 cures the problem partly.
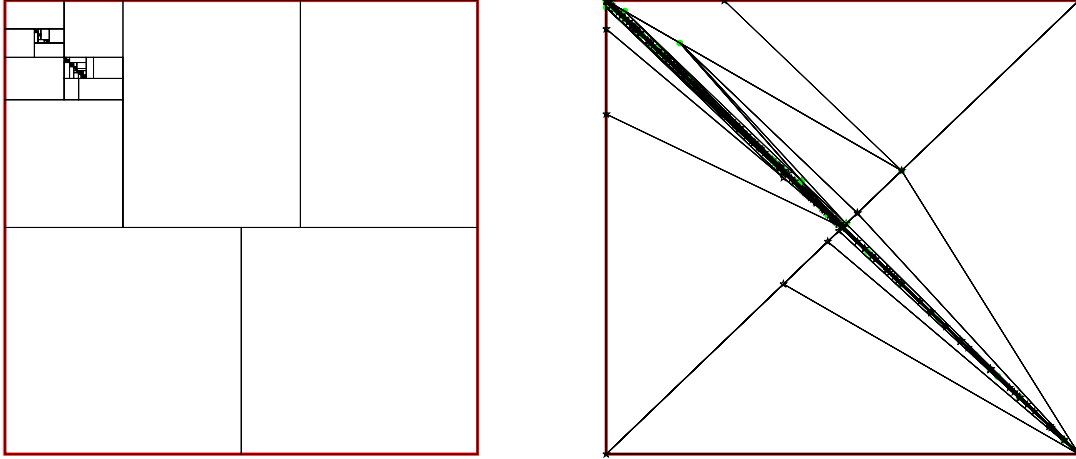
Figure 16: Rectangular and triangular foam of 1000 cells for the distribution of eq. (26).

plot the triangular foam of cells is clearly aligning with the diagonal ridge. In the rows 2-3 of Table 10 we see the reasonable numerical results for the triangular foam. They are for the maximum weight reduction and variance reduction options respectively; the other configuration parameters are rather close to the default ones. In rows 4-5 we are playing with the increase of the cell number and in the rows 6-7 with the number of the MC events used in the cell exploration. Finally in rows 8-9 we change binning of the histograms used in the MC cell exploration. As we see, the most profitable in terms of the MC efficiency/precision is the increase of the number of the cells, however, adjusting other parameters can also help. In all cases we show the number of the calls of the distribution `nCalls` in the foam build-up. In the best result of the line 5 with 10000 triangular cells we have obtained 5 digit precision for about 10^7 function calls[28].

Summarizing, we see from the above exercise, that the user of the `Foam` has a possibility to adjust several configuration parameters, such that the MC efficiency for a given distribution is improved quite significantly.

6.4   Decay of τ lepton into 3 pions

In Table 11 we collect numerical results for an example of the `Foam` application to the very practical problem of the MC simulation of the decay $\tau \to \nu\pi^-\pi^+\pi^-$, according to the distribution (matrix element squared) taken from the MC program `TAUOLA` [20, 21].

The amplitude of the decay process contains two distinct parts due two Feynman diagrams, see Fig. 17, which have peaks due to a_1 resonance and ρ resonance. There are two peaks due to ρ resonances partly overlapping in the integration space, such that

---

[28]In ref. [19] the same precision for the same function was attained for about 10^8 function calls.

MAPPING		k	n	N_c	N_s	N_b	$\frac{N_{eff}}{bin}$	nCall	$\frac{\sigma}{\langle w \rangle}$	$\frac{\langle w \rangle}{w^\varepsilon_{\max}}$	$\Delta R/R$	SIZE
(a1)	OFF	8	0	1	1K	4	25	518	2.1555	0.038	0.00481	15KB
(a2)	ON	9	0	1	1K	4	25	218	1.1391	0.115	0.00254	15KB
(b1)	OFF	8	0	20	1K	4	25	5767	1.1847	0.130	0.00264	15KB
(b2)	ON	9	0	20	1K	4	25	3548	0.7626	0.206	0.00170	15KB
(c0)	OFF	0	8	1000	1K	4	25	487K	1.6603	0.085	0.00371	54KB
(c1)	OFF	8	0	1000	1K	4	25	145K	0.5298	0.359	0.00118	53KB
(c2)	ON	9	0	1000	1K	4	25	125K	0.7626	0.394	0.00104	53KB
(d1)	OFF	8	0	5000	1K	4	25	528K	0.4330	0.438	0.00096	209KB
(d2)	ON	9	0	5000	1K	4	25	596K	0.4037	0.467	0.00090	209KB

Table 11: Numerical results from Foam simulation/integration for the decay process $\tau \to \nu\pi^-\pi^+\pi^-$, according to matrix element squared of the TAUOLA Monte Carlo [20, 21] program. All MC averages are for 200K events generated after Foam initialization with the configuration parameters given in the table. Notation and the meaning of the quantities are the same as in previous tables. The size of the ROOT disk-file, in which the Foam object was written is indicated in the last column.
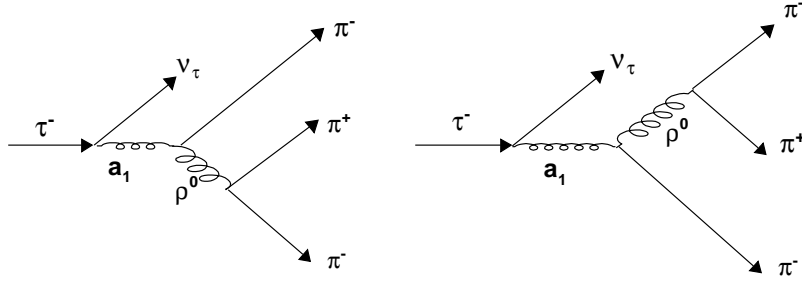

Figure 17: Feynman diagrams for τ decay into 3 pions.

the actual shape of the differential distribution is rather complicated. We took for this exercise subroutine DPHTRE of TAUOLA in which nine random numbers[29] are replaced by the nine variables of the Foam. The 4-particle phase space is 8-dimensional. The ninth variable is due to two branches in the phase space parametrization of TAUOLA, and in case of the Foam as well (the method is similar to that of Section 2.11). In cases of "no mapping and no multibranching" we are back to eight dimensions. The variables x_1 and x_2 of the Foam represent (up to a linear transformation) the two effective masses of 3π and 2π system. The next four variables $x_i, i = 3, 4, 5, 6$ are polar variables $\cos\theta$ and ϕ of the pions in the rest frame of the 3π and 2π systems – this is a completely standard

---
[29]Including two random numbers of the subroutine SPHERA and two (Euler) angles corresponding to the overall rotation of the entire event.
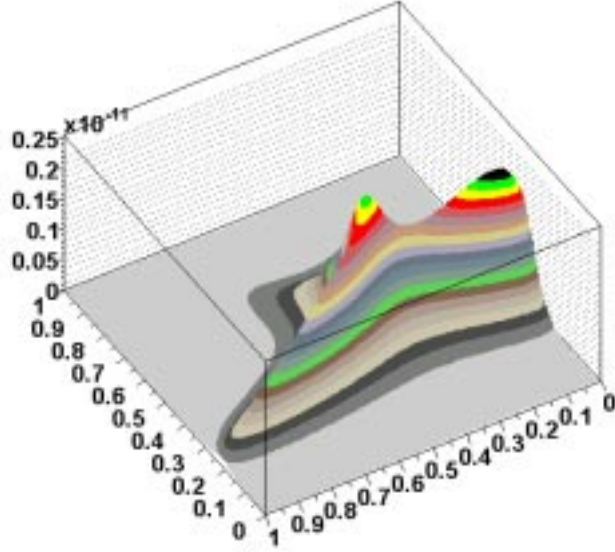
Figure 18: The distribution of $\tau \to \nu 3\pi$ decay as a function of x_1 and x_2 (without mapping), fixing the other variables to some values.

phase space parametrization, see ref. [21], and also Ref. [22]. The variables x_7 and x_8 are reserved for the overall rotation, and the last one x_9 is mapped into branch index, see Section 2.11. Variables x_7 and x_8 are inhibited (no cell split in them), because the distribution does not depend on them. Variable x_9 (if present) has a predefined division value equal 0.5 and is inhibited for the division (see Section 2.11). In Fig. 18 we show the decay distribution as a function of x_1 and x_2 (without mapping), fixing the other variables to some values. The distribution is clearly a non-trivial one.

In Table 11 we show MC efficiency of the Foam for gradually increasing number of the hyper-rectangular cells, with the mapping compensating for the for Breit-Wigner peaks of the a_1 and ρ resonances (as in Ref. [21]) and without. One example with simplical cells is also included.

The most striking result in Table 11 is the comparison of lines (a2) and (b1): the Foam algorithm *with only 20 cells* is performing equally well as the doubly-branched mapping compensating for the resonance peaks of the a_1 and ρ. When going to higher number of cells, the MC efficiency in the cases with and without mapping becomes almost the same. This is expected, because Foam also does the mapping compensating for the resonances on its own. From the row (c0) we see also that the simplical mode of the Foam is clearly under-performing. We think that foam with 1000 cells, see row (c1) in Table 11, is an economic solution for this problem[30]. (Mapping due to resonances is not really necessary

---

[30]The net profit with respect to TAUOLA [20, 21] would be three times faster program, and more importantly, a significantly simpler code.
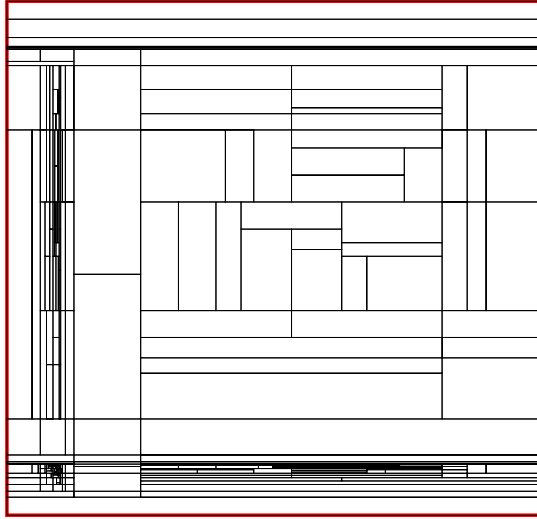
52

in this case.)

Figure 19: `Foam of rectangular cells for the electron-positron beamstrahlung spectrum of circe2` [18] with 500 cells.

6.5   Beamstrahlung spectrum

Fig. 19 shows foam of rectangular cells for the 2-dimensional beamstrahlung spectrum $D(z_1, z_2)$ of the electron-positron collider [23] at 500GeV, encoded in the program `circe2` of Ref. [24, 18]. It should be stressed that this spectrum is not known analytically but rather through a numerical fit to results of the machine simulation or (in the future) from an experiment. In order to avoid (integrable) infinite singularities at $z_i = 1$ in $D(z_1, z_2)$ we use in Fig. 19 variables $t_i = (1 - z_i)^{0.1}$, $i = 1, 2$. For this exercise we used foam of 500 cells getting the MC efficiency $\sigma/\langle w \rangle = 0.41$ and $\langle w \rangle/w^{\varepsilon}_{\max} = 0.64$, (for $\varepsilon = 0.0005$); enough for practical application (can be easily improved by adding more cells).

Generating $D(z_1, z_2)$ is not really so very much important and difficult problem. A more interesting problem is to generate the distribution $D(z_1, z_2)\sigma(sz_1 z_2)$, where $\sigma(s)$ is the cross section of some physics process, which may have a strong singularity of its own, like resonance or threshold factor. Such a problem was already treated with help of `Foam` program in KKMC [13] program and the study of Ref. [25].

7 Conclusions

The author hopes that this new adaptive tool for constructing efficient MC programs will find its way to many applications in high energy physics and beyond. Thhe main points on the new `Foam` algorithm and the program are the following:

- `Foam` is a versatile adaptive general purpose Monte Carlo similator.

- `Foam` algorithm is based on the cellular division of the integration domain.

- Geometry of the "foam of cells" is rather simple, simplical or hyperrectangular cells are constructed in the process of a binary split.

- It works in principle for arbitrary distribution – no assumption of factorizability as in `VEGAS` of Ref. [3].

- `Foam` is reducing maximum weight of the weight distribution, it can therefore provide unweighted events. The variance reduction, useful for the integration and generating weighted events, is also available.

- Memory-efficient coding of cells allows to build up to $\sim 10^6$ cells in the computer memory of a typical desktop computer.

- The rules for picking up next cell for the division and the division geometry starts to be relatively sophisticated (projection on edges etc.) This costs CPU time which becomes the main barrier towards higher MC efficiency.

- `Foam` can deal efficiently with strongly peaked distribution up to ~ 12 dimensions, with todays desktop computers.

8 Acknowledgements

A   Variance optimization

Suppose we have already constructed the cells $\omega_1, \omega_1, \ldots, \omega_N$ and within each cell we have defined the function $\rho'(x)$ constant over the cell, $\rho'(x) = \rho'_I$. The integral $R'_I = \int_{\omega_i} \rho'(x) dx^n = \rho'_I V_I$ is known, because the volume of the cell V_I is known. The function $\rho(x)$ is not, in general, constant over the cell and the weight $w = \rho(x)/\rho'(x)$ is used to determine its integral in the usual way: $R = R'\langle w \rangle_{\rho'} = \sum_I \langle w \rangle_{\rho'_I} R_I$, where the average

$$\langle a \rangle_{\rho'_I} = \frac{1}{R'_I} \int_{\omega_I} \rho'_I(x) a(x) dx^n$$

is defined for the I-th cell alone.

The question is now the following: preserving the geometry of the cells, can we get smaller variance, simply, by changing the probabilities of the generation of the cells? Rescaling $\rho'_I \to \rho''_I = \lambda_I \rho'_I$ does not affect the integral R, because the change of the normalization of R' and $\langle w \rangle_{\rho'}$ is cancelling. It is convenient to assume that the above rescaling preserves $R' = R''$ and the total average weight $\langle w \rangle_{\rho'} = \langle w \rangle_{\rho''}$, that is λ_I obey the constraint $\sum_I R'_I = \sum_I R'_I \lambda_I = const$. With the above constraint in mind we now ask: for what values of λ_I the dispersion of the total weight $\sigma^2 = \langle w^2 \rangle_{\rho''} - \langle w \rangle^2_{\rho''}$ is minimal[31]. Since by construction $\langle w \rangle_{\rho''}$ is independent of λ_I, we may look only for a minimum of $\langle w^2 \rangle_{\rho''}$. With the standard methods we get a (local) minimum condition:

$$\frac{\partial}{\partial \lambda_I} \left\{ \langle w^2 \rangle_{\rho''} + \Lambda R'' \right\} = \frac{\partial}{\partial \lambda_I} \left\{ \frac{1}{R'} \sum_I \int_{\omega_I} \left(\frac{\rho(x)}{\rho'_I \lambda_I} \right)^2 \rho'_I \lambda_I dx^n + \Lambda \left(\sum_I R'_I \lambda_I \right) \right\} = 0 \quad (27)$$

where Λ is the Lagrange multiplier. The solution of the minimum condition

$$\frac{R'_I}{R'} \langle w^2 \rangle_{\rho'_I} \frac{1}{\lambda_I^2} - \Lambda R'_I = 0 \quad (28)$$

is simply $\lambda_I \simeq const \times \sqrt{\langle w^2 \rangle_{\rho'_I}}$, or more precisely

$$\lambda_I = \frac{\sqrt{\langle w^2 \rangle_{\rho'_I}}}{\sum_J \frac{R'_J}{R'} \sqrt{\langle w^2 \rangle_{\rho'_J}}}. \quad (29)$$

The value of $\langle w^2 \rangle_{\min} \equiv \langle w^2 \rangle_{\rho''}$ calculated at the minimum, that is for λ_I of eq. (29), is also rather simple

$$\langle w^2 \rangle_{\min} = \sum_I \frac{R'_I}{R'} \langle w^2 \rangle_{\rho'_I} \frac{1}{\lambda_I^2} = \left(\sum_J \frac{R'_J}{R'} \sqrt{\langle w^2 \rangle_{\rho'_J}} \right)^2 = \left(\left\langle \sqrt{\langle w^2 \rangle_{\rho'_J}} \right\rangle_{\rho'} \right)^2. \quad (30)$$

---

[31]This problem was, of course, often considered in the past, see for instance Refs. [3, 15]. We outline here the solution for the sake of completeness and convenience of the reader.

Let us note that the functional $\sqrt{\langle w^2 \rangle_{\min}}$, which we intend to minimize in the process of the evolution of the foam (cell split) is a simple sum of contributions from all cells. Consequently, when working out details of the split of a given cell we may calculate the gain in terms of the total dispersion independently of other cells, see also below. This very convenient feature is exploited in the algorithm of the foam build-up.

Adopting (temporarily) the following normalization conventions: $\rho'_I = 1$, $w = \rho(x)$ and $R'_I = V_I$, $\lambda_I \simeq \sqrt{\langle \rho^2 \rangle_I}$, we get

$$\rho''_I = \sqrt{\langle \rho^2 \rangle_I}, \qquad R''_I = V_I \sqrt{\langle \rho^2 \rangle_I}, \tag{31}$$

where the average $\langle \ldots \rangle_I$ is understood as defined for points uniformly distributed within the I-th cell.

In the `Foam` algorithm we do not go, of course, through a judicious adjustment of the relative importance of the cells, which minimizes the variance as described above, but instead, we simply declare that the distribution $\rho'(x)$ is defined by the optimum solution of eq. (31). Once we have done it, for the MC weight w defined with respect to such a new $\rho'(x)$, we find out that $\langle w^2 \rangle_I = 1$ for each cell, and also for all cells, $\langle w^2 \rangle = 1$. What is then minimized in the process of the cell division is the ratio of the variance to the average weight[32]

$$\frac{\sigma^2}{\langle w \rangle^2} = \left(\frac{R'}{R} \right)^2 - 1. \tag{32}$$

This quantity is not so convenient to optimize in the process of the cell split (foam evolution) and we rather chose to minimize a closely related 'linearized" quantity

$$R_{loss} = R \left(\sqrt{\frac{\sigma^2}{\langle w \rangle^2} + 1} - 1 \right) = \sum_I V_I (\sqrt{\langle w^2 \rangle_I} - \langle w \rangle_I) = \int \rho_{loss}(x) dx^n = R' - R, \tag{33}$$

which is a sum over all cells and is a monotonous ascending function of $\sigma/\langle w \rangle$. In the process of the cell division $\omega_I \to \omega_{Ia} \oplus \omega_{Ib}$ we decrease $\sigma/\langle w \rangle$ step by step, by playing with the geometry of the cell split such that the gain in the total $R_{loss} = R' - R$ due to a given cell split is as big as possible. It is convenient that the contributions from the other cells to R_{loss} are unchanged. In this way every cell split will lead to a smaller and smaller $\sigma/\langle w \rangle$ in the final MC run.

---

[32]We exploit here the relation $\langle w \rangle = R/R'$, and we should keep in mind that R is constant during the variance minimization.

B   Output of the demonstration program in C++

```
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
F                                                                          F
F                  *****************************************               F
F                  ******     TFOAM::Initialize     ******               F
F                  *****************************************               F
F                                            FoamX                         F
F     Version =              2.05    =           Release date:  2002.02.13 F
F        nDim =          0 =  Dimension of the simplical sub-space         F
F        kDim =          2 =  Dimension of the hyper-cubical sub-space     F
F      nCells =       1000 =  Requested number of Cells (half of them active) F
F      nSampl =        500 =  No of MC events in exploration of a cell cell F
F        nBin =          8 =  No of bins in histograms, MC exploration of cell F
F    EvPerBin =         25 =  Maximum No effective_events/bin, MC exploration F
F    OptDrive =          2 =  Type of Driver   =1,2 for Sigma,WtMax        F
F     OptEdge =          0 =  Decides whether vertices are included in the MC F
F     OptPeek =          0 =  Type of the cell Peek =0,1 for maximum, random F
F      OptOrd =          0 =  Root cell hyp-cub. or simplex, =0,1           F
F    OptMCell =          1 =  MegaCell option, slim memory for hyp-cubes   F
F    OptDebug =          1 =  Additional debug histogram, SetDirectory(1)  F
F    OptCu1st =          1 =  Numbering of dimensions  starts with h-cubic  F
F      OptRej =          0 =  MC rejection on/off for OptRej=0,1            F
F    MaxWtRej =            2     =       Maximum wt in rejection for wt=1 evts F
F                                                                          F
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF


FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
F                                                                          F
F                  ***  TFOAM::Initialize FINISHED!!!   ***               F
F     nCalls =    246799 =            Total number of function calls        F
F     XPrime =    1.2748942   =   Primary total integral                    F
F     XDiver =    0.27441356  =   Driver  total integral                    F
F    MCresult =   1.0004806   =   Estimate of the true MC Integral          F
F                                                                          F
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
```

```
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
F                                                                          F
F                 *****************************************                F
F                 ******      TFOAM::Finalize      ******                 F
F                 *****************************************                F
F     NevGen =        2000 = Number of generated events in the MC generation F
F     LastVe =          -1 = Number of vertices (only for simplical option) F
F     nCalls =      248799 = Total number of function calls                F
F               ----------------------------------------                   F
F      AveWt =   0.78836949  =    Average MC weight                        F
F      WtMin = 1.3022085e-06 =    Minimum MC weight (absolute)             F
F      WtMax =   1.0227719   =    Maximum MC weight (absolute)             F
F               ----------------------------------------                   F
F      XPrime =   1.2748942  =    Primary total integral, R_prime          F
F      XDiver =   0.27441356 =    Driver  total integral, R_loss           F
F               ----------------------------------------                   F
F      IntMC =    1.0050877 +-   0.0052357675  = Result of the MC Integral F
F    MCerelat = 0.0052092644 =   Relative error of the MC intgral          F
F    <w>/WtMax = 0.80037512  =          MC efficiency, acceptance rate     F
F    Sigma/<w> = 0.23296539  =          MC efficiency, variance/ave_wt     F
F       WtMax =       0.985  =          WtMax(esp= 0.0005)                  F
F       Sigma =   0.1836628  =          variance of MC weight              F
F                                                                          F
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
```

References

[1] S. Jadach, Comput. Phys. Commun. 130 (2000) 244, physics/9910004.

[2] M. Ciesla and M. Slusarczyk, M.Sc thesis at the Institute of the Computer Science, Jagiellonian University, Kraków, Poland, 2000.

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

[4] S. Kawabata, Comp. Phys. Commun. 88 (1995) 309.

[5] T. Ohl, Comput. Phys. Commun. 120 (1999) 13, hep-ph/9806432.

[6] G.I. Manankova, A.F. Tatarchenko and F.V. Tkachov, Milxy way: How much better than vegas can one integrate in many dimensions?, A Contribution to AINHEP-95, Pisa, Italy, Apr 3-8, 1995 (extended version).

[7] E. de Doncker and A. Gupta, Parallel Computing 24 (1998) 1223.

[8] E. de Doncker, A. Gupta and R. Zanny, Journal of Computational and Applied Mathematics 112 (1999).

[9] E. de Doncker, K. Kaugars and A. Gupta, User manual of parint1.1, http://www.cs.wmich.edu/~parint/.

[10] R. Brun and F. Rademakers, Root - an object oriented data analysis framework, Proceedings AIHENP'96 Workshop, Lausanne, Sep. 1996, Nucl. Inst., Meth. in Phys. Res. A 389 (1997) 81-86, See also http://root.cern.ch/.

[11] S. Jadach and W. Płaczek, Comput. Phys. Commun. 72 (1992) 221.

[12] M. Boonekamp, (2001), hep-ph/0111213.

[13] S. Jadach, B.F.L. Ward and Z. Was, Comput. Phys. Commun. 130 (2000) 260, hep-ph/9912214.

[14] S. Jadach, (1999), physics/9906056.

[15] R. Kleiss and R. Pittau, Comput. Phys. Commun. 83 (1994) 141, hep-ph/9405257.

[16] G. Marsaglia, B. Narasimhan and A. Zaman, Comput. Phys. Commun. 60 (1990) 345.

[17] S. Jadach et al., Comput. Phys. Commun. 140 (2001) 475, hep-ph/0104049.

[18] T. Ohl, Circe version 2.0: Beam spectra for simulating linear collider and photon collider physics, (to be published, code and parametrisations provided by the author in advance), 2002, WUE-ITP-2002-006.

[19] E. de Doncker, R. Zanny and K. Kaugars, Integrand and performance analysis with parint/parvis, http://www.cs.wmich.edu/∼parint/, 2000.

[20] S. Jadach et al., Comput. Phys. Commun. 76 (1993) 361.

[21] M. Jezabek et al., Comput. Phys. Commun. 70 (1992) 69.

[22] M. Skrzypek and Z. Was, Comput. Phys. Commun. 125 (2000) 8, hep-ph/9904385.

[23] ECFA/DESY LC Physics Working Group, J.A. Aguilar-Saavedra et al., (2001), hep-ph/0106315.

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

[25] G. Blair, Particle masses and widths via threshold scans at the linear collider, APS/DPF/DPB Summer Study on the Future of Particle Physics (Snowmass 2001), Snowmass, Colorado, 30 June - 21 July 2001.