MPP–2004–40
hep–ph/0404043

Cuba – a library for multidimensional numerical integration

T. Hahn
Max-Planck-Institut für Physik
Föhringer Ring 6, D–80805 Munich, Germany

Aug 28, 2014

Abstract

The Cuba library provides new implementations of four general-purpose multidimensional integration algorithms: Vegas, Suave, Divonne, and Cuhre. Suave is a new algorithm, Divonne is a known algorithm to which important details have been added, and Vegas and Cuhre are new implementations of existing algorithms with only few improvements over the original versions. All four algorithms can integrate vector integrands and have very similar Fortran, C/C++, and Mathematica interfaces.

1 Introduction

Many problems in physics (and elsewhere) involve computing an integral, and often enough this has to be done numerically, as the analytical result is known only in a limited number of cases. In one dimension, the situation is quite satisfactory: standard packages, such as Quadpack [1], reliably integrate a broad class of functions in modest CPU time. The same is unfortunately not true for multidimensional integrals.

This paper presents the Cuba library with new implementations of four algorithms for multidimensional numerical integration: Vegas, Suave, Divonne, and Cuhre. They have a C/C++, Fortran, and Mathematica interface each and are invoked in a very similar way, thus making them easily interchangeable, e.g. for comparison purposes. All four can integrate vector integrands. Cuhre is a deterministic algorithm, the others use Monte Carlo methods.

Vegas is the simplest of the four. It uses importance sampling for variance reduction, but is only in some cases competitive in terms of the number of samples needed to reach a prescribed accuracy. Nevertheless, it has a few improvements over the original algorithm [2, 3] and comes in handy for cross-checking the results of other methods.

Suave is a new algorithm which combines the advantages of two popular methods: importance sampling as done by Vegas and subregion sampling in a manner similar to Miser [4]. By dividing into subregions, Suave manages to a certain extent to get around

Vegas' difficulty to adapt its weight function to structures not aligned with the coordinate axes.

Divonne is a further development of the CERNLIB routine D151 [5]. Divonne works by stratified sampling, where the partitioning of the integration region is aided by methods from numerical optimization. A number of improvements have been added to this algorithm, the most significant being the possibility to supply knowledge about the integrand. Narrow peaks in particular are difficult to find without sampling very many points, especially in high dimensions. Often the exact or approximate location of such peaks is known from analytic considerations, however, and with such hints the desired accuracy can be reached with far fewer points.

Cuhre* employs a cubature rule for subregion estimation in a globally adaptive subdivision scheme [6]. It is hence a deterministic, not a Monte Carlo method. In each iteration, the subregion with the largest error is halved along the axis where the integrand has the largest fourth difference. Cuhre is quite powerful in moderate dimensions, and is usually the only viable method to obtain high precision, say relative accuracies much below 10^{-3}.

The new algorithms were coded from scratch in C, which is a compromise of sorts between C++ and Fortran 77, combining ease of linking to Fortran code with the availability of reasonable memory management. The declarations have been chosen such that the routines can be called from Fortran directly. The Mathematica versions are based on the same C code and use the MathLink API to communicate with Mathematica.

2 Vegas

Vegas is a Monte Carlo algorithm that uses importance sampling as a variance-reduction technique. Vegas iteratively builds up a piecewise constant weight function, represented on a rectangular grid. Each iteration consists of a sampling step followed by a refinement of the grid. The exact details of the algorithm can be found in [2, 3] and shall not be reproduced here.

Changes with respect to the original version are:

- Sobol quasi-random numbers [7] rather than pseudo-random numbers are used by default. Empirically, this seems to accelerate convergence quite a bit, most noticeably in the early stages of the integration.

  From theoretical considerations it is of course known (see e.g. [8]) that quasi-random sequences yield a convergence rate of $\mathcal{O}(\log^{n_d} n_s/n_s)$, where n_d is the number of dimensions and n_s the number of samples, which is much better than the usual $\mathcal{O}(1/\sqrt{n_s})$ for ordinary Monte Carlo. But these convergence rates are meaningful only for large n_s and so it came as a pleasant surprise that the gains are considerable already at the beginning of the sampling process. It shows that quasi-Monte Carlo methods blend well with variance-reduction techniques such as importance sampling.

*The D from the original name was dropped since the CUBA library uses double precision throughout.

Similarly, it was not clear from the outset whether the statistical standard error would furnish a suitable error estimate since quasi-random numbers are decidedly non-random in a number of respects. Yet also here empirical evidence suggests that the standard error works just as well as for pseudo-random numbers.

- The present implementation allows the number of samples to be increased in each iteration. With this one can mimic the strategy of calling Vegas with a small number of samples first to 'get the grid right' and then using an alternate entry point to perform the 'full job' on the same grid with a larger number of samples.

- The option to add simple stratified sampling on top of the importance sampling, as proposed in the appendix of [2], has not been implemented in the present version. Tests with the Vegas version from [9], which contains this feature, showed that convergence was accelerated only when the original pseudo-random numbers were used and that with quasi-random numbers convergence was in fact even slower in some cases.

Vegas' major weakness is that it uses a separable (product) weight function. As a consequence, Vegas can offer significant improvements only as far as the integrand's characteristic regions are aligned with the coordinate axes.

3 Suave

Suave (short for SUbregion-Adaptive VEgas) uses Vegas-like importance sampling combined with a globally adaptive subdivision strategy: Until the requested accuracy is reached, the region with the largest error at the time is bisected in the dimension in which the fluctuations of the integrand are reduced most. The number of new samples in each half is prorated for the fluctuation in that half.

A similar method, known as recursive stratified sampling, is implemented in Miser [4]. Miser always samples a fixed number of points, however, which is somewhat undesirable since it does not stop once the prescribed accuracy is reached.

Suave first samples the integration region in a Vegas-like step, i.e. using importance sampling with a separable weight function. It then slices the integration region in two, as Miser would do. Suave does not immediately recurse on those subregions, however, but maintains a list of all subregions and selects the region with the largest absolute error for the next cycle of sampling and subdivision. That is, Suave uses global error estimation and terminates when the requested relative or absolute accuracy is attained.

The information on the weight function collected in one Vegas step is not lost. Rather, the grid from which the weight function is computed is stretched and re-used on the subregions. A region which is the result of $m - 1$ subdivisions thus has had m Vegas iterations performed on it.

The improvements over Vegas and Miser come at a price, which is the amount of memory required to hold all the samples. Memory consumption is not really severe on

modern hardware, however. The component that scales worst is the one proportional to the number of samples, which is

$$8(n_d + n_c + 1)n_s \text{ bytes},$$

where n_d is the number of dimensions of the integral, n_c the number of components of the integrand, and n_s the number of samples. For a million samples on a scalar integrand of 10 variables, this works out to 96 megabytes – not all that enormous these days.

3.1 Description of the algorithm

As Suave is a new algorithm, the following description will be fairly detailed. For greater notational clarity, n_c-dimensional vectors are denoted with a vector arrow ($\vec{f}$) and n_d-dimensional vectors with boldface letters ($\mathbf{x}$) in the following, where n_d is the dimension of the integral and n_c the number of components of the integrand.

The essential inputs are ε_{rel} and ε_{abs}, the relative and absolute accuracies, n_s^{new}, the number of samples added in each iteration, $n_s^{\max}$, the maximum number of samples allowed, and p, a flatness parameter described below.

Suave has a main loop which calls a Vegas-like sampling step. The main loop is responsible for subdividing the subregions and maintaining the totals. The sampling step does the actual sampling on the subregions and computes the region results.

3.1.1 Main loop

1. Initialize the random-number generator and allocate a data structure for the entire integration region. Initialize its Vegas grid with equidistant bins.

2. Sample the entire integration region with n_s^{new} points. This gives an initial estimate of the integral $\vec{I}_{\text{tot}}$, the variance $\vec{\sigma}^2_{\text{tot}}$, and $\vec{\chi}^2_{\text{tot}}$.

3. Find the component c for which $r_c = \sigma_{c,\text{tot}} / \max(\varepsilon_{\text{abs}}, \varepsilon_{\text{rel}}|I_{c,\text{tot}}|)$ is maximal.

   If none of the r_c's exceeds unity, indicate success and return.

4. If the number of samples spent so far equals or exceeds $n_s^{\max}$, indicate failure and return.

5. Find the region r with the largest σ_c^2.

6. Find the dimension d which minimizes $F_c(r_L^d) + F_c(r_R^d)$, where $r_{L,R}^d$ are the left and right halves of r with respect to d. $F_c(r_{L,R}^d)$ is the fluctuation of the samples that fall into $r_{L,R}^d$ and is computed as

$$F_c(r_{L,R}^d) = \left[\left\| 1 + \tilde{F}_c(\mathbf{x}_i \in r_{L,R}^d) \right\|_p \right]^{2/3} = \left[\sum \left| 1 + \tilde{F}_c(\mathbf{x}_i \in r_{L,R}^d) \right|^p \right]^{2/(3p)}, \quad (1)$$

where all samples $\mathbf{x}_i$ that fall into the respective half are used in the norm/sum and the single-sample fluctuation $\tilde{F}_c$ is defined as

$$\tilde{F}_c(\mathbf{x}) = w(\mathbf{x}) \left|\frac{f_c(\mathbf{x}) - I_c(r)}{I_c(r)}\right| \frac{|f_c(\mathbf{x}) - I_c(r)|}{\sigma_c(r)} .$$

This empirical recipe combines the relative deviation from the region mean, $(f-I)/I$, with the χ value, $|f-I|/\sigma$, weighted by the Vegas weight w corresponding to sample $\mathbf{x}$. Note that the I_c and σ_c values of the entire region r are used.

Samples strongly contribute to F the more they lie away from the predicted mean *and* the more they lie out of the predicted error band. Tests have shown that large values of p are beneficial for 'flat' integrands, whereas small values are preferred if the integrand is 'volatile' and has high peaks. p has thus been dubbed a flatness parameter. The effect comes from the fact that with increasing p, F becomes more and more dominated by 'outliers,' i.e. points with a large $\tilde{F}$.

The power 2/3 in Eq. (1) is also used in Miser, where it is motivated as the exponent that gives the best variance reduction ([9], p. 315).

7. Refine the grid associated with r, i.e. incorporate the information gathered on the integrand in the most recent sample over r into the weight function. This is done precisely as in Vegas (see [2]), with the extension that if the integrand has more than one component, the marginal densities are computed not from f^2 but from the weighted sum[†]

$$\overline{f^2} = \sum_{c=1}^{n_c} \frac{f_c^2}{I_{c,\mathrm{tot}}^2} .$$

8. Bisect r in dimension d:

   Allocate a new region, r_L, and copy to r_L those of r's samples falling into the left half. Compute the Vegas grid for r_L by appropriately "stretching" r's grid, i.e. by interpolating all grid points of r with values less than 1/2.

   Set up r_R for the right half analogously.

9. Sample r_L with $n_L = \max\left(\frac{F_c(r_L)}{F_c(r_L)+F_c(r_R)} n_s^{\mathrm{new}}, n_s^{\mathrm{min}}\right)$ and r_R with $n_R = \max(n_s^{\mathrm{new}} - n_L, n_s^{\mathrm{min}})$ points, where $n_s^{\mathrm{min}} = 10$.

[†]It is fairly obvious that scale-invariant quantities must be used in the sum, otherwise the component with the largest absolute scale would dominate. It is less clear whether $\eta_0 = (\int f_c \,\mathrm{d}\mathbf{x})^2 = I_{c,\mathrm{tot}}^2$, $\eta_1 = (\int |f_c| \,\mathrm{d}\mathbf{x})^2$, or $\eta_2 = \int f_c^2 \,\mathrm{d}\mathbf{x}$ (or any other) make the best weights. Empirically, η_0 turns out to be both slightly superior in convergence and easier to compute than η_1 and η_2 and has thus been chosen in Suave.

A possible explanation for this is that in cases where there are large compensations within the integral, i.e. when $\int f_c \,\mathrm{d}\mathbf{x} \ll \int |f_c| \,\mathrm{d}\mathbf{x}$, it is particularly necessary for the overall accuracy that component c be sampled accurately, and thus be given more weight in $\overline{f^2}$, and this is better accomplished by dividing f_c^2 by the "small" number η_0 than by the "large" number η_1 or η_2.

10. To safeguard against underestimated errors, supplement the variances by the difference of the integral values in the following way:

$$\sigma^2_{c,\mathrm{new}}(r_{R,L}) = \sigma^2_c(r_{R,L}) \left(1 + \frac{\Delta_c}{\sqrt{\sigma^2_c(r_L) + \sigma^2_c(r_R)}}\right)^2 + \Delta^2_c$$

for each component c, where $\vec{\Delta} = \frac{1}{4}|\vec{I}(r_L) + \vec{I}(r_R) - \vec{I}(r)|$.

This acts as a penalty for regions whose integral value changes significantly by the subdivision and effectively moves them up in the order of regions to be subdivided next.

11. Update the totals: Subtract r's integral, variance, and χ^2-value from the totals and add those of r_L and r_R.

12. Discard r, put r_L and r_R in the list of regions.

13. Go to Step 3.

3.1.2 Sampling step

The function which does the actual sampling is a modified Vegas iteration. It is invoked with two arguments: r, the region to be sampled and n_m, the number of new samples.

1. Sample a set of n_m new points using the weight function given by the grid associated with r. For a region which is the result of $m-1$ subdivisions, the list of samples now consists of m sets of samples.

2. For each set of samples, compute the mean $\vec{I}_i$ and variance $\vec{\sigma}^2_i$.

3. Compute the results for the region as

$$I_c = \frac{\sum_{i=1}^m w_{i,c} I_{i,c}}{\sum_{i=1}^m w_{i,c}}, \quad \sigma^2_c = \frac{1}{\sum_{i=1}^m w_{i,c}}, \quad \chi^2_c = \frac{1}{\sigma^2_c}\left[\frac{\sum_{i=1}^m w_{i,c} I^2_{i,c}}{\sum_{i=1}^m w_{i,c}} - I^2_c\right],$$

where the inverse of the set variances are used as weights, $w_{i,c} = 1/\sigma^2_{i,c}$. This is simply Gaussian error propagation.

For greater numerical stability, χ^2_c is actually computed as

$$\chi^2_c = \sum_{i=1}^m w_{i,c} I^2_{i,c} - I_c \sum_{i=1}^m w_{i,c} I_{i,c} = \sum_{i=2}^m w_{i,c} I_{i,c}(I_{i,c} - I_{1,c}) - I_c \sum_{i=2}^m w_{i,c}(I_{i,c} - I_{1,c})\,.$$

4 Divonne

Divonne uses stratified sampling for variance reduction, that is, it partitions the integration region such that all subregions have an approximately equal value of a quantity called the spread $\vec{s}$, defined as

$$\vec{s}(r) = \frac{1}{2}V(r)\Big(\max_{\mathbf{x}\in r}\vec{f}(\mathbf{x}) - \min_{\mathbf{x}\in r}\vec{f}(\mathbf{x})\Big)\,, \tag{2}$$

where $V(r)$ is the volume of region r. What sets Divonne apart from Suave is that the minimum and maximum of the integrand are sought using methods from numerical optimization. Particularly in high dimensions, the chance that one of the previously sampled points lies in or even close to the true extremum is fairly small.

On the other hand, the numerical minimization is beset with the usual pitfalls, i.e. starting from the lowest of a (relatively small) number of sampled points, Divonne will move directly into the local minimum closest to the starting point, which may or may not be close to the absolute minimum.

Divonne is a lot more complex than Suave and Vegas but also significantly faster for many integrands. For details on the methods used in Divonne please consult the original references [5]. New features with respect to the CERNLIB version (Divonne 4) are:

- Integration is possible in dimensions 2 through 33 (not 9 as before). Going to higher dimensions is a matter of extending internal tables only.

- The possibility has been added to specify the location of possible peaks, if such are known from analytical considerations. The idea here is to help the integrator find the extrema of the integrand, and narrow peaks in particular are a challenge for the algorithm. Even if only the approximate location is known, this feature of hinting the integrator can easily cut an order of magnitude out of the number of samples needed to reach the required accuracy for complicated integrands. The points can be specified either statically, by passing a list of points at the invocation, or dynamically, through a subroutine called for each subregion.

- Often the integrand subroutine cannot sample points lying on or very close to the integration border. This can be a problem with Divonne which actively searches for the extrema of the integrand and homes in on peaks regardless of whether they lie on the border. The user may however specify a border region in which integrand values are not obtained directly, but extrapolated from two points inside the 'safe' interior.

- The present algorithm works in three phases, not two as before. Phase 1 performs the partitioning as outlined above. From the preliminary results obtained in this phase, Divonne estimates the number of samples necessary to reach the desired accuracy in phase 2, the final integration phase. Once the phase-2 sample for a particular subregion is in, a χ^2 test is used to assess whether the two sample averages are consistent with each other within their error bounds. Subregions which fail this test

move on to phase 3, the refinement phase, where they can be subdivided again or sampled a third time with more points, depending on the parameters set by the user.

- For all three phases the user has a selection of methods to obtain the integral estimate: a Korobov [10] or Sobol [7] quasi-random sample of given size, a Mersenne Twister [11] or Ranlux [12] pseudo-random sample of given size, and the cubature rules of Genz and Malik [13] of degree 7, 9, 11, and 13 that are also used in Cuhre. The latter are embedded rules and hence provide an intrinsic error estimate (that is, an error estimate not based on the spread). When this independent error estimate is available, it supersedes the spread-based error when computing the total error. Also, regions whose spread-based error exceeds the intrinsic error are selected for refinement, too.

  In spite of these novel options, the cubature rules of the original Divonne algorithm were not implemented.

Due to its complexity, the new Divonne implementation was painstakingly tested against the CERNLIB routine to make sure it produces the same results before adding the new features.

5 Cuhre

Cuhre is a deterministic algorithm which uses one of several cubature rules of polynomial degree in a globally adaptive subdivision scheme. The subdivision algorithm is similar to Suave's (see Sect. 3.1.1) and works as follows:

While the total estimated error exceeds the requested bounds:

1) choose the region with the largest estimated error,

2) bisect this region along the axis with the largest fourth difference,

3) apply the cubature rule to the two subregions,

4) merge the subregions into the list of regions and update the totals.

Details on the algorithm and on the cubature rules employed in Cuhre can be found in the original references [6]. The present implementation offers only superficial improvements, such as an interface consistent with the other Cuba routines and a slightly simpler invocation, e.g. one does not have to allocate a workspace.

In moderate dimensions Cuhre is very competitive, particularly if the integrand is well approximated by polynomials. As the dimension increases, the number of points sampled by the cubature rules rises considerably, however, and by the same token the usefulness declines. For the lower dimensions, the actual number of points that are spent per invocation of the basic integration rule are listed in the following table.

number of dimensions	4	5	6	7	8	9	10	11	12
points in degree-7 rule	65	103	161	255	417	711	1265	2335	4433
points in degree-9 rule	153	273	453	717	1105	1689	2605	4117	6745

6 Download and Compilation

The CUBA package can be downloaded from `http://feynarts.de/cuba`. The gzipped tar file unpacks into a directory `Cuba-`$m.n$. Change into this directory and type

```
./configure
make
```

This should create

`libcuba.a`	— the CUBA library,
`Vegas`, `Suave`, `Divonne`, `Cuhre`	— the MathLink executables,
`demo-c`, `demo-fortran`	— the demonstration programs,
`partview`	— the partition viewer.

CUBA can also be built in parts: "make lib" builds only the CUBA library, "make math" builds only the MathLink executables, "make demos" builds only the demo programs, and "make tools" builds only the partition viewer.

The MathLink executables require mcc, the MathLink compiler, and the partition viewer needs Qt. Compilation of the corresponding parts will be switched off by default if configure does not find these tools.

The code is C99 compliant and compiles flawlessly with the GNU C compiler, versions 2.95 and higher. Configure should take care of most C99 features. In particular if it finds that the C compiler cannot handle variable-size arrays, it will fix the array sizes at compile time. In this case, the maximum number of dimensions can be chosen with the configure option `--with-maxdim=`$n_d^{\texttt{max}}$ (default: 16) and the maximum number of components of the integrand with `--with-maxcomp=`$n_c^{\texttt{max}}$ (default: 4).

Linking Fortran or C/C++ code that uses one of the algorithms is straightforward, just add `-lcuba` (for the CUBA library) and `-lm` (for the math library) to the compiler command line, as in

```
f77 -o myexecutable mysource.f -lcuba -lm
cc -o myexecutable mysource.c -lcuba -lm
```

The `demo` subdirectory contains the source for the demonstration programs in Fortran 77, C, and Mathematica, as well as the test suite used in Sect. 9, which is also written in Mathematica.

7 User Manual

7.1 Usage in Fortran

Although written in C, the declarations have been chosen such that the routines are directly accessible from Fortran, i.e. no wrapper code is needed. In fact, Vegas, Suave, Divonne, and Cuhre can be called as if they were Fortran subroutines respectively declared as

```
   subroutine vegas(ndim, ncomp, integrand, userdata, nvec,
&    epsrel, epsabs, flags, seed, mineval, maxeval,
&    nstart, nincrease, nbatch, gridno, statefile, spin,
&    neval, fail, integral, error, prob)

   subroutine suave(ndim, ncomp, integrand, userdata, nvec,
&    epsrel, epsabs, flags, seed, mineval, maxeval,
&    nnew, flatness, statefile, spin,
&    nregions, neval, fail, integral, error, prob)

   subroutine divonne(ndim, ncomp, integrand, userdata, nvec,
&    epsrel, epsabs, flags, seed, mineval, maxeval,
&    key1, key2, key3, maxpass,
&    border, maxchisq, mindeviation,
&    ngiven, ldxgiven, xgiven, nextra, peakfinder,
&    statefile, spin,
&    nregions, neval, fail, integral, error, prob)

   subroutine cuhre(ndim, ncomp, integrand, userdata, nvec,
&    epsrel, epsabs, flags, mineval, maxeval,
&    key, statefile, spin,
&    nregions, neval, fail, integral, error, prob)
```

7.1.1 Common Arguments

- `integer ndim` $\langle in \rangle$, the number of dimensions of the integral.
- `integer ncomp` $\langle in \rangle$, the number of components of the integrand.
- `integer integrand` $\langle in \rangle$, the integrand. The external function which computes the integrand is expected to be declared as

  ```
  integer function integrand(ndim, x, ncomp, f, userdata, nvec, core)
  integer ndim, ncomp, nvec, core
  double precision x(ndim,nvec), f(ncomp,nvec)
  ```

  The integrand receives `nvec` samples in `x` and is supposed to fill the array `f` with the corresponding integrand values. Note that `nvec` indicates the actual number of points passed to the integrand here and may be smaller than the `nvec` given to the integrator.

  The return value is irrelevant unless it is -999, in the case of which the integration will be aborted immediately. This might happen if a parameterized integrand turns out not to yield sensible values for a particular parameter set (passed e.g. through `userdata`).

The worker process the integrand is running on is indicated in `core`, where `core` < 0 indicates an Accelerator, `core` $\geqslant 0$ a regular (CPU) core, and `32768` the master itself (more details in Sect. 8.2.2).

The latter three arguments, `userdata`, `nvec`, and `core` are optional and may be omitted if unused, i.e. the integrand is minimally declared (for `nvec` $= 1$) as

```
integer function integrand(ndim, x, ncomp, f)
integer ndim, ncomp
double precision x(ndim), f(ncomp)
```

- `(arbitrary type) userdata` $\langle in \rangle$, user data passed to the integrand. Unlike its counterpart in the integrand definition, this argument must be present though it may contain a dummy value, e.g. `0`.

- `integer nvec` $\langle in \rangle$, The maximum number of points to be given to the integrand routine in each invocation. Usually this is 1 but if the integrand can profit from e.g. SIMD vectorization, a larger value can be chosen.

- `double precision epsrel, epsabs` $\langle in \rangle$, the requested relative and absolute accuracies. The integrator tries to find an estimate $\hat{I}$ for the integral I which for every component c fulfills $|\hat{I}_c - I_c| \leqslant \max(\varepsilon_{\text{abs}}, \varepsilon_{\text{rel}}|I_c|)$.

- `integer flags` $\langle in \rangle$, flags governing the integration:

  - Bits 0 and 1 are taken as the verbosity level, i.e. 0 to 3, unless the `CUBAVERBOSE` environment variable contains an even higher value (used for debugging).

    Level 0 does not print any output, level 1 prints 'reasonable' information on the progress of the integration, level 2 also echoes the input parameters, and level 3 further prints the subregion results (if applicable).

  - Bit $2 = 0$, all sets of samples collected on a subregion during the various iterations or phases contribute to the final result.

    Bit $2 = 1$, only the last (largest) set of samples is used in the final result.

  - (Vegas and Suave only) Bit $3 = 0$, apply additional smoothing to the importance function, this moderately improves convergence for many integrands,

    Bit $3 = 1$, use the importance function without smoothing, this should be chosen if the integrand has sharp edges.

  - Bit $4 = 0$, delete the state file (if one is chosen) when the integration terminates successfully,

    Bit $4 = 1$, retain the state file.

  - Bits 8–31 =: `level` determines the random-number generator (see below).

  To select e.g. last samples only and verbosity level 2, pass $6 = 4 + 2$ for the flags.

- `integer seed` ⟨*in*⟩, the seed for the pseudo-random-number generator.

  The random-number generator is chosen as follows:

`seed`	`level` (bits 8–31 of `flags`)	Generator
zero	N/A	Sobol (quasi-random),
non-zero	zero	Mersenne Twister (pseudo-random),
non-zero	non-zero	Ranlux (pseudo-random).

  Ranlux implements Marsaglia and Zaman's 24-bit RCARRY algorithm with generation period p, i.e. for every 24 generated numbers used, another $p - 24$ are skipped. The luxury level is encoded in `level` as follows:

  - Level 1 ($p = 48$): very long period, passes the gap test but fails spectral test.
  - Level 2 ($p = 97$): passes all known tests, but theoretically still defective.
  - Level 3 ($p = 223$): any theoretically possible correlations have very small chance of being observed.
  - Level 4 ($p = 389$): highest possible luxury, all 24 bits chaotic.

  Levels 5–23 default to 3, values above 24 directly specify the period p. Note that Ranlux's original level 0, (mis)used for selecting Mersenne Twister in CUBA, is equivalent to `level = 24`.

- `integer mineval` ⟨*in*⟩, the minimum number of integrand evaluations required.

- `integer maxeval` ⟨*in*⟩, the (approximate) maximum number of integrand evaluations allowed.

- `character*(*) statefile` ⟨*in*⟩, a filename for storing the internal state. To not store the internal state, put `""` (empty string) or `%VAL(0)` (null pointer).

  CUBA can store its entire internal state (i.e. all the information to resume an interrupted integration) in an external file. The state file is updated after every iteration. If, on a subsequent invocation, a CUBA routine finds a file of the specified name, it loads the internal state and continues from the point it left off. Needless to say, using an existing state file with a different integrand generally leads to wrong results.

  This feature is useful mainly to define 'check-points' in long-running integrations from which the calculation can be restarted.

  Once the integration reaches the prescribed accuracy, the state file is removed, unless bit 4 of `flags` (see above) explicitly requests that it be kept.

- `integer*8 spin` ⟨*in*⟩, the 'spinning cores' pointer, which has three options:

– A value of `-1` or `%VAL(0)` (null pointer) means that the integrator completely takes care of starting and terminating child processes for the integration (if available/enabled), i.e. after the integrator returns there are no child processes running any longer. Note that a 'naive' `-1` (which is an `integer`, not an `integer*8`) is explicitly allowed.

– A zero-initialized variable `spin` instructs the integrator to start child processes for the integration but keep them running and store the 'spinning cores' pointer in `spin` on exit. Take care that in this case you have to explicitly terminate the child processes using `cubawait` later on (see Sect. 8.2.1).

– A non-zero variable `spin` means that the cores are already running as the result of some prior integration or an explicit `cubafork` call (see Sect. 8.2.1).

The actual type of `spin` is irrelevant, the variable must merely be wide enough to store a C `void *`.

- `integer nregions` $\langle out \rangle$, the actual number of subregions needed (not present in Vegas).
- `integer neval` $\langle out \rangle$, the actual number of integrand evaluations needed.
- `integer fail` $\langle out \rangle$, an error flag:
  - `fail` $= 0$, the desired accuracy was reached,
  - `fail` $= -1$, dimension out of range,
  - `fail` > 0, the accuracy goal was not met within the allowed maximum number of integrand evaluations. While Vegas, Suave, and Cuhre simply return 1, Divonne can estimate the number of points by which `maxeval` needs to be increased to reach the desired accuracy and returns this value.
- `double precision integral(ncomp)` $\langle out \rangle$, the integral of `integrand` over the unit hypercube.
- `double precision error(ncomp)` $\langle out \rangle$, the presumed absolute error of `integral`.
- `double precision prob(ncomp)` $\langle out \rangle$, the χ^2-probability (not the χ^2-value itself!) that `error` is not a reliable estimate of the true integration error[‡].

[‡]To judge the reliability of the result expressed through `prob`, remember that it is the null hypothesis that is tested by the χ^2 test, which is that `error` *is* a reliable estimate. In statistics, the null hypothesis may be rejected only if `prob` is fairly close to unity, say `prob` $> .95$.

7.1.2 Vegas-specific Arguments

- `integer nstart` $\langle in \rangle$, the number of integrand evaluations per iteration to start with.

- `integer nincrease` $\langle in \rangle$, the increase in the number of integrand evaluations per iteration.

- `integer nbatch` $\langle in \rangle$, the batch size for sampling.

  Vegas samples points not all at once, but in batches of size `nbatch`, to avoid excessive memory consumption. 1000 is a reasonable value, though it should not affect performance too much.

- `integer gridno` $\langle in \rangle$, the slot in the internal grid table.

  It may accelerate convergence to keep the grid accumulated during one integration for the next one, if the integrands are reasonably similar to each other. Vegas maintains an internal table with space for ten grids for this purpose. The slot in this grid is specified by `gridno`.

  If a grid number between 1 and 10 is selected, the grid is not discarded at the end of the integration, but stored in the respective slot of the table for a future invocation. The grid is only re-used if the dimension of the subsequent integration is the same as the one it originates from.

  In repeated invocations it may become necessary to flush a slot in memory, in which case the negative of the grid number should be set.

Vegas actually passes the integrand two more arguments, i.e. the integrand subroutine is really declared as

```
integer function integrand(ndim, x, ncomp, f, userdata, nvec, core,
  weight, iter)
integer ndim, ncomp, nvec, core, iter
double precision x(ndim,nvec), f(ncomp,nvec), weight(nvec)
```

where `weight` contains the weight of the point being sampled and `iter` the current iteration number. These extra arguments may safely be ignored, however.

7.1.3 Suave-specific Arguments

- `integer nnew` $\langle in \rangle$, the number of new integrand evaluations in each subdivision.

- `double precision flatness` $\langle in \rangle$, the parameter p in Eq. (1), i.e. the type of norm used to compute the fluctuation of a sample. This determines how prominently 'outliers,' i.e. individual samples with a large fluctuation, figure in the total fluctuation,

which in turn determines how a region is split up. As suggested by its name, `flatness` should be chosen large for 'flat' integrands and small for 'volatile' integrands with high peaks. Note that since `flatness` appears in the exponent, one should not use too large values (say, no more than a few hundred) lest terms be truncated internally to prevent overflow.

Like Vegas, Suave also passes the two optional arguments `weight` and `iter` to the integrand (see Sect. 7.1.2).

7.1.4 Divonne-specific Arguments

- `integer key1` $\langle in \rangle$, determines sampling in the partitioning phase:

  `key1` $= 7, 9, 11, 13$ selects the cubature rule of degree `key1`. Note that the degree-11 rule is available only in 3 dimensions, the degree-13 rule only in 2 dimensions.

  For other values of `key1`, a quasi-random sample of $n_1 =$ |`key1`| points is used, where the sign of `key1` determines the type of sample,

  - `key1` > 0, use a Korobov quasi-random sample,
  - `key1` < 0, use a "standard" sample (a Sobol quasi-random sample if `seed` $= 0$, otherwise a pseudo-random sample).

- `integer key2` $\langle in \rangle$, determines sampling in the final integration phase:

  `key2` $= 7, 9, 11, 13$ selects the cubature rule of degree `key2`. Note that the degree-11 rule is available only in 3 dimensions, the degree-13 rule only in 2 dimensions.

  For other values of `key2`, a quasi-random sample is used, where the sign of `key2` determines the type of sample,

  - `key2` > 0, use a Korobov quasi-random sample,
  - `key2` < 0, use a "standard" sample (see description of `key1` above),

  and $n_2 =$ |`key2`| determines the number of points,

  - $n_2 \geqslant 40$, sample n_2 points,
  - $n_2 < 40$, sample $n_2\, n_{\text{need}}$ points, where n_{need} is the number of points needed to reach the prescribed accuracy, as estimated by Divonne from the results of the partitioning phase.

- `integer key3` $\langle in \rangle$, sets the strategy for the refinement phase:

  `key3` $= 0$, do not treat the subregion any further.

  `key3` $= 1$, split the subregion up once more.

  Otherwise, the subregion is sampled a third time with `key3` specifying the sampling parameters exactly as `key2` above.

- `integer maxpass` $\langle in \rangle$, controls the thoroughness of the partitioning phase: The partitioning phase terminates when the estimated total number of integrand evaluations (partitioning plus final integration) does not decrease for `maxpass` successive iterations.

  A decrease in points generally indicates that Divonne discovered new structures of the integrand and was able to find a more effective partitioning. `maxpass` can be understood as the number of 'safety' iterations that are performed before the partition is accepted as final and counting consequently restarts at zero whenever new structures are found.

- `double precision border` $\langle in \rangle$, the width of the border of the integration region. Points falling into this border region will not be sampled directly, but will be extrapolated from two samples from the interior. Use a non-zero `border` if the integrand subroutine cannot produce values directly on the integration boundary.

- `double precision maxchisq` $\langle in \rangle$, the maximum χ^2 value a single subregion is allowed to have in the final integration phase. Regions which fail this χ^2 test and whose sample averages differ by more than `mindeviation` move on to the refinement phase.

- `double precision mindeviation` $\langle in \rangle$, a bound, given as the fraction of the requested error of the entire integral, which determines whether it is worthwhile further examining a region that failed the χ^2 test. Only if the two sampling averages obtained for the region differ by more than this bound is the region further treated.

- `integer ngiven` $\langle in \rangle$, the number of points in the `xgiven` array.

- `integer ldxgiven` $\langle in \rangle$, the leading dimension of `xgiven`, i.e. the offset between one point and the next in memory.

- `double precision xgiven(ldxgiven,ngiven)` $\langle in \rangle$, a list of points where the integrand might have peaks. Divonne will consider these points when partitioning the integration region. The idea here is to help the integrator find the extrema of the integrand in the presence of very narrow peaks. Even if only the approximate location of such peaks is known, this can considerably speed up convergence.

- `integer nextra` $\langle in \rangle$, the maximum number of extra points the peak-finder subroutine will return. If `nextra` is zero, `peakfinder` is not called and an arbitrary object may be passed in its place, e.g. just 0.

- `external peakfinder` $\langle in \rangle$, the peak-finder subroutine. This subroutine is called whenever a region is up for subdivision and is supposed to point out possible peaks lying in the region, thus acting as the dynamic counterpart of the static list of points supplied in `xgiven`. It is expected to be declared as

```
subroutine peakfinder(ndim, b, n, x)
integer ndim, n
double precision b(2,ndim)
double precision x(ldxgiven,n)
```

The bounds of the subregion are passed in the array `b`, where `b(1,`d`)` is the lower and `b(2,`d`)` the upper bound in dimension d. On entry, `n` specifies the maximum number of points that may be written to `x`. On exit, `n` must contain the actual number of points in `x`.

Divonne actually passes the integrand one more argument, i.e. the integrand subroutine is really declared as

```
subroutine integrand(ndim, x, ncomp, f, userdata, nvec, core,
  phase)
integer ndim, ncomp, nvec, core, phase
double precision x(ndim,nvec), f(ncomp,nvec)
```

The last argument, `phase`, indicates the integration phase:

- 0, sampling of the points in `xgiven`,
- 1, partitioning phase,
- 2, final integration phase,
- 3, refinement phase.

This information might be useful if the integrand takes long to compute and a sufficiently accurate approximation of the integrand is available. The actual value of the integral is only of minor importance in the partitioning phase, which is instead much more dependent on the peak structure of the integrand to find an appropriate tessellation. An approximation which reproduces the peak structure while leaving out the fine details might hence be a perfectly viable and much faster substitute when `phase` < 2.

In all other instances, `phase` can be ignored and it is entirely admissible to define the integrand without it.

7.1.5 Cuhre-specific Arguments

- `integer key` $\langle in \rangle$, chooses the basic integration rule:

  `key` $= 7, 9, 11, 13$ selects the cubature rule of degree `key`. Note that the degree-11 rule is available only in 3 dimensions, the degree-13 rule only in 2 dimensions.

  For other values, the default rule is taken, which is the degree-13 rule in 2 dimensions, the degree-11 rule in 3 dimensions, and the degree-9 rule otherwise.

7.1.6 Visualizing the Tessellation

Suave, Divonne, and Cuhre work by dividing the integration region into subregions for integration. When verbosity level 3 is selected in the flags, the actual tessellation is written out on screen and can be visualized with the partview tool. To this end, the output of the program invoking CUBA is piped through partview, e.g.

```
mycubaprogram | partview 1 2 1 3
```

opens a window with two tabs showing the 1–2 and 1–3 plane of the tessellation. The saturation of the colours corresponds to the area of the region, i.e. smaller regions are displayed in a darker shade.

7.2 Usage in C/C++

Being written in C, the algorithms can of course be used in C/C++ directly. The declarations are as follows:

```
typedef int (*integrand_t)(const int *ndim, const double x[],
  const int *ncomp, double f[], void *userdata);

typedef void (*peakfinder_t)(const int *ndim, const double b[],
  int *n, double x[]);

void Vegas(const int ndim, const int ncomp,
  integrand_t integrand, void *userdata, const int nvec,
  const double epsrel, const double epsabs,
  const int flags, const int seed,
  const int mineval, const int maxeval,
  const int nstart, const int nincrease, const int nbatch,
  const int gridno, const char *statefile, void *spin,
  int *neval, int *fail,
  double integral[], double error[], double prob[])

void Suave(const int ndim, const int ncomp,
  integrand_t integrand, void *userdata, const int nvec,
  const double epsrel, const double epsabs,
  const int flags, const int seed,
  const int mineval, const int maxeval,
  const int nnew, const double flatness,
  const char *statefile, void *spin,
  int *nregions, int *neval, int *fail,
  double integral[], double error[], double prob[])
```

```
void Divonne(const int ndim, const int ncomp,
  integrand_t integrand, void *userdata, const int nvec,
  const double epsrel, const double epsabs,
  const int flags, const int seed,
  const int mineval, const int maxeval,
  const int key1, const int key2, const int key3,
  const int maxpass, const double border,
  const double maxchisq, const double mindeviation,
  const int ngiven, const int ldxgiven, double xgiven[],
  const int nextra, peakfinder_t peakfinder,
  const char *statefile, void *spin,
  int *nregions, int *neval, int *fail,
  double integral[], double error[], double prob[])

void Cuhre(const int ndim, const int ncomp,
  integrand_t integrand, void *userdata, const int nvec,
  const double epsrel, const double epsabs,
  const int flags,
  const int mineval, const int maxeval,
  const int key, const char *statefile, void *spin,
  int *nregions, int *neval, int *fail,
  double integral[], double error[], double prob[])
```

These prototypes are contained in `cuba.h` which should (in C) or must (in C++) be included when using the CUBA routines. The arguments are as in the Fortran case, with the obvious translations, e.g. `double precision` = `double`. Note, however, the declarations of the integrand and peak-finder functions, which expect pointers to integers rather than integers. This is required for compatibility with Fortran.

The `integrand_t` type glosses over the fact that the extra `nvec` argument is routinely passed to the integrand and neither does it mention the extra arguments passed by `Vegas`, `Suave`, and `Divonne` (see Sects. 7.1.2 and 7.1.4). This is usually just what is needed for 'simple' invocations, i.e. with the 'correct' prototypes the compiler would only generate unnecessary warnings (in C) or errors (in C++). In the rare cases where the integrand actually has more arguments, an explicit typecast to `integrand_t` must be used in the invocation. In the presence of an `nvec` argument, the `x` and `f` arguments are actually two-dimensional arrays, `x[*nvec][*ndim]` and `f[*nvec][*ncomp]`.

7.3 Usage in Mathematica

The Mathematica versions are based on essentially the same C code and communicate with Mathematica via the MathLink API. When building the package, the executables `Vegas`, `Suave`, `Divonne`, and `Cuhre` are compiled for use in Mathematica. In Mathematica one first needs to load them with the `Install` function, as in

```
Install["Divonne"]
```

which makes a Mathematica function of the same name available. These functions are used almost like `NIntegrate`, only some options are different. For example,

```
Vegas[x^2/(Cos[x + y + 1] + 5), {x,0,5}, {y,0,5}]
```

integrates a scalar function, or

```
Suave[{Sin[z] Exp[-x^2 - y^2],
       Cos[z] Exp[-x^2 - y^2]}, {x,-1,1}, {y,-1,3}, {z,0,1}]
```

integrates a vector. As is evident, the integration region can be chosen different from the unit hypercube. Innermore boundaries may depend on outermore integration variables, e.g. `Cuhre[1, {x,0,1}, {y,0,x}]` gives the area of the unit triangle.

The sampling function uses `MapSample` to map the integrand over the data points. This is by default set to `Map`, but can be changed (after `Install`) e.g. to `ParallelMap` to take advantage of parallelization (see Sect. 8 for more details).

The functions return a list which contains the results for each component of the integrand in a sublist {integral estimate, estimated absolute error, χ^2 probability}. For the Suave example above this would be

```
{{1.1216, 0.000991577, 0.0000104605},
 {2.05246, 0.00146661, 0.00920716}}
```

The other parameters are specified via the following options. Default values are given on the right-hand sides of the rules.

7.3.1 Common Options

- `PrecisionGoal -> 3`, the number of digits of relative accuracy to seek, that is, $\varepsilon_{\text{rel}} = 10^{-\texttt{PrecisionGoal}}$.
- `AccuracyGoal -> 12`, the number of digits of absolute accuracy to seek, that is, $\varepsilon_{\text{abs}} = 10^{-\texttt{AccuracyGoal}}$. The integrator tries to find an estimate $\hat{I}$ for the integral I which for every component c fulfills $|\hat{I}_c - I_c| \leqslant \max(\varepsilon_{\text{abs}}, \varepsilon_{\text{rel}} I_c)$.
- `MinPoints -> 0`, the minimum number of integrand evaluations required.
- `MaxPoints -> 50000`, the (approximate) maximum number of integrand evaluations allowed.
- `Verbose -> 1`, how much information to print on intermediate results, can take values from 0 to 3.

Level 0 does not print any output, level 1 prints 'reasonable' information on the progress of the integration, level 2 also echoes the input parameters, and level 3 further prints the subregion results (if applicable). Note that the subregion boundaries in the level-3 printout refer to the unit hypercube, i.e. are possibly scaled with respect to the integration limits passed to Mathematica. This is because the underlying C code, which emits the output, is unaware of any scaling of the integration region, which is done entirely in Mathematica.

- `Final -> All` *or* `Last`, whether only the last (largest) or all sets of samples collected on a subregion during the various iterations or phases contribute to the final result.

- `PseudoRandom -> False`, whether pseudo-random numbers are used for sampling instead of Sobol quasi-random numbers. Values `True` and `0` select the Mersenne Twister algorithm, any other integer n chooses Ranlux with luxury level n (see Sect. 7.1.1).

- `PseudoRandomSeed -> Automatic`, the seed for the pseudo-random-number generator.

- `Regions -> False`, whether to return the tessellation of the integration region (thus not present in Vegas, which does not partition the integration region).

  If `Regions -> True` is chosen, a two-component list is returned, where the first element is the list of regions, and the second element is the integration result as described above. Each region is specified in the form `Region[`x_{ll}`,` x_{ur}`,` *res*`,` *df*`]`, where x_{ll} and x_{ur} are the multidimensional equivalents of the lower left and upper right corner, *res* is the integration result for the subregion, given in the same form as the total result but with the χ^2 value instead of the χ^2 probability, and *df* are the degrees of freedom corresponding to the χ^2 values.

  Cuhre cannot state a χ^2 value separately for each region, hence the χ^2 values and degrees of freedom are omitted from the `Region` information.

- `Compiled -> True`, whether to compile the integrand function before use. Note two caveats:

  - The function values still have to pass through the MathLink interface, and in the course of this are truncated to machine precision. Not compiling the integrand will thus in general not deliver more accurate results.
  - Compilation should be switched off if the compiled integrand shows unexpected behaviour. As the Mathematica online help points out, "the number of times and the order in which objects are evaluated by `Compile` may be different from ordinary Mathematica code."

7.3.2 Vegas-specific Options

- `NStart -> 1000`, the number of integrand evaluations per iteration to start with.
- `NIncrease -> 500`, the increase in the number of integrand evaluations per iteration.
- `NBatch -> 1000`, the number of points sent in one MathLink packet to be sampled by Mathematica. This setting will at most affect performance and should not normally need to be changed.
- `GridNo -> 0`, the slot in the internal grid table.

  It may accelerate convergence to keep the grid accumulated during one integration for the next one, if the integrands are reasonably similar to each other. Vegas maintains an internal table with space for ten grids for this purpose. If a `GridNo` between 1 and 10 is chosen, the grid is not discarded at the end of the integration, but stored for a future invocation. The grid is only re-used if the dimension of the subsequent integration is the same as the one it originates from. A negative grid number initializes the indicated slot before the integration (for details see Sect. 7.1.2).
- `StateFile -> ""`, the file name for storing the internal state. If a non-empty string is given here, Vegas will store its entire internal state (i.e. all the information to resume an interrupted integration) in this file after every iteration. If, on a subsequent invocation, Vegas finds a file of the specified name, it loads the internal state and continues from the point it left off. Needless to say, using an existing state file with a different integrand generally leads to wrong results.

  This feature is useful mainly to define 'check-points' in long-running integrations from which the calculation can be restarted.
- `RetainStateFile -> False`, whether the state file shall be kept even if the integration terminates normally, i.e. reaches either the prescribed accuracy or the maximum number of points.
- During the evaluation of the integrand, the global variable `$Weight` is set to the weight of the point being sampled and `$Iteration` to the current iteration number.

7.3.3 Suave-specific Options

- `NNew -> 1000`, the number of new integrand evaluations in each subdivision.
- `Flatness -> 50`, the parameter p in Eq. (1), i.e. the type of norm used to compute the fluctuation of a sample. This determines how prominently 'outliers,' i.e. individual samples with a large fluctuation, figure in the total fluctuation, which in turn determines how a region is split up. As suggested by its name, `Flatness` should be chosen large for 'flat' integrands and small for 'volatile' integrands with high peaks.

Note that since `Flatness` appears in the exponent, one should not use too large values (say, no more than a few hundred) lest terms be truncated internally to prevent overflow.

- During the evaluation of the integrand, the global variable `$Weight` is set to the weight of the point being sampled and `$Iteration` to the current iteration number.

7.3.4 Divonne-specific Options

- `Key1 -> 47`, an integer which governs sampling in the partitioning phase:

  `Key1` $= 7, 9, 11, 13$ selects the cubature rule of degree `Key1`. Note that the degree-11 rule is available only in 3 dimensions, the degree-13 rule only in 2 dimensions.

  For other values of `Key1`, a quasi-random sample of $n_1 =$ |`Key1`| points is used, where the sign of `Key1` determines the type of sample,

  - `Key1` > 0, use a Korobov quasi-random sample,
  - `Key1` < 0, use a "standard" sample (a Sobol quasi-random sample in the case `PseudoRandom -> False`, otherwise a pseudo-random sample).

- `Key2 -> 1`, an integer which governs sampling in the final integration phase:

  `Key2` $= 7, 9, 11, 13$ selects the cubature rule of degree `Key2`. Note that the degree-11 rule is available only in 3 dimensions, the degree-13 rule only in 2 dimensions.

  For other values of `Key2`, a quasi-random sample is used, where the sign of `Key2` determines the type of sample,

  - `Key2` > 0, use a Korobov quasi-random sample,
  - `Key2` < 0, use a "standard" sample (see description of `Key1` above),

  and $n_2 =$ |`Key2`| determines the number of points,

  - $n_2 \geqslant 40$, sample n_2 points,
  - $n_2 < 40$, sample $n_2\, n_{\text{need}}$ points, where n_{need} is the number of points needed to reach the prescribed accuracy, as estimated by Divonne from the results of the partitioning phase.

- `Key3 -> 1`, an integer which sets the strategy for the refinement phase:

  `Key3` $= 0$, do not treat the subregion any further.

  `Key3` $= 1$, split the subregion up once more.

  Otherwise, the subregion is sampled a third time with `Key3` specifying the sampling parameters exactly as `Key2` above.

- `MaxPass -> 5`, the number of passes after which the partitioning phase terminates. The partitioning phase terminates when the estimated total number of integrand evaluations (partitioning plus final integration) does not decrease for `MaxPass` successive iterations.

  A decrease in points generally indicates that Divonne discovered new structures of the integrand and was able to find a more effective partitioning. `MaxPass` can be understood as the number of 'safety' iterations that are performed before the partition is accepted as final and counting consequently restarts at zero whenever new structures are found.

- `Border -> 0`, the width of the border of the integration region. Points falling into this border region are not sampled directly, but are extrapolated from two samples from the interior. Use a non-zero `Border` if the integrand function cannot produce values directly on the integration boundary.

  The border width always refers to the unit hypercube, i.e. it is not rescaled if the integration region is not the unit hypercube.

- `MaxChisq -> 10`, the maximum χ^2 value a single subregion is allowed to have in the final integration phase. Regions which fail this χ^2 test and whose sample averages differ by more than `MinDeviation` move on to the refinement phase.

- `MinDeviation -> .25`, a bound, given as the fraction of the requested error of the entire integral, which determines whether it is worthwhile further examining a region that failed the χ^2 test. Only if the two sampling averages obtained for the region differ by more than this bound is the region further treated.

- `Given -> {}`, a list of points where the integrand might have peaks. A point is a list of n_d real numbers, where n_d is the dimension of the integral.

  Divonne will consider these points when partitioning the integration region. The idea here is to help the integrator find the extrema of the integrand in the presence of very narrow peaks. Even if only the approximate location of such peaks is known, this can considerably speed up convergence.

- `NExtra -> 0`, the maximum number of points that will be considered in the output of the `PeakFinder` function.

- `PeakFinder -> ({}&)`, the peak-finder function. This function is called whenever a region is up for subdivision and is supposed to point out possible peaks lying in the region, thus acting as the dynamic counterpart of the static list of points supplied with `Given`. It is invoked with two arguments, the multidimensional equivalents of the lower left and upper right corners of the region being investigated, and must return a (possibly empty) list of points. A point is a list of n_d real numbers, where n_d is the dimension of the integral.

7.3.5 Cuhre-specific Options

- `Key -> 0`, chooses the basic integration rule:

  `Key` $= 7, 9, 11, 13$ selects the cubature rule of degree `Key`. Note that the degree-11 rule is available only in 3 dimensions, the degree-13 rule only in 2 dimensions.

  For other values, the default rule is taken, which is the degree-13 rule in 2 dimensions, the degree-11 rule in 3 dimensions, and the degree-9 rule otherwise.

7.3.6 Visualizing the Tessellation

Suave, Divonne, and Cuhre work by dividing the integration region into subregions for integration. The tessellation is returned together with the integration results when the option `Regions -> True` is set. Such output can be visualized using the Mathematica program `partview.m` that comes with CUBA. The invocation is e.g.

```
result = Divonne[..., Regions -> True]
<< tools/partview.m
PartView[result, 1, 2]
```

which displays the 1–2 plane of the tessellation. The saturation of the colours corresponds to the area of the region, i.e. smaller regions are displayed in a darker shade.

7.4 Long-integer Versions

For both Fortran and C/C++ there exist versions of the integration routines that take 64-bit integers for all number-of-points-type quantities. These should be used in cases where convergence is not reached within the ordinary 32-bit integer range ($2^{31} - 1$).

The long-integer versions are distinguished by the "`ll`" prefix. Their specific invocations are, in Fortran,

```
      subroutine llvegas(ndim, ncomp, integrand, userdata, nvec,
     &  epsrel, epsabs, flags, seed, mineval, maxeval,
     &  nstart, nincrease, nbatch, gridno, statefile, spin,
     &  neval, fail, integral, error, prob)

      subroutine llsuave(ndim, ncomp, integrand, userdata, nvec,
     &  epsrel, epsabs, flags, seed, mineval, maxeval,
     &  nnew, flatness, statefile, spin,
     &  nregions, neval, fail, integral, error, prob)

      subroutine lldivonne(ndim, ncomp, integrand, userdata, nvec,
     &  epsrel, epsabs, flags, seed, mineval, maxeval,
     &  key1, key2, key3, maxpass,
```

```
&    border, maxchisq, mindeviation,
&    ngiven, ldxgiven, xgiven, nextra, peakfinder,
&    statefile, spin,
&    nregions, neval, fail, integral, error, prob)

   subroutine llcuhre(ndim, ncomp, integrand, userdata, nvec,
&    epsrel, epsabs, flags, seed, mineval, maxeval,
&    key, statefile, spin,
&    nregions, neval, fail, integral, error, prob)
```

The correspondences for C/C++ are obvious and are given explicitly in the include file `cuba.h`. The arguments are as for the normal versions except that all underlined variables are of type `integer*8` in Fortran and `long long int` in C/C++.

8 Parallelization

Numerical integration is perfectly suited for parallel execution, which can significantly speed up the computation as it generally incurs only a very small overhead. The parallelization procedure is rather different in Fortran/C/C++ and in Mathematica. We shall deal with the latter first because it needs only a short explanation. The remainder of this chapter is then devoted to the Fortran/C/C++ case.

8.1 Parallelization in Mathematica

The Mathematica version of Cuba performs its sampling through a function `MapSample`. By default this is identical to `Map`, i.e. the serial version, so to parallelize one merely needs to redefine `MapSample = ParallelMap` (after loading Cuba).

If the integrand depends on user-defined symbols or functions, their definitions must be distributed to the workers beforehand using `DistributeDefinitions` and likewise required packages must be loaded with `ParallelNeeds` instead of `Needs`; this is explained in detail in the Mathematica manual.

8.2 Parallelization in Fortran and C/C++

In Fortran and C/C++ the Cuba library can (and usually does) automatically parallelize the sampling of the integrand. It parallelizes through `fork` and `wait` which, though slightly less performant than pthreads, do not require reentrant code. (Reentrancy may not even be under full control of the programmer, for example Fortran's I/O is usually non-reentrant.) Worker processes are started and shut down only as few times as possible, however, so the performance penalty is really quite minor even for non-native fork implementations such as Cygwin's. Parallelization is not available on native Windows for lack of the `fork` function.

The communication of samples to and from the workers happens through IPC shared memory (`shmget` and colleagues), or if that is not available, through a `socketpair` (two-way pipe). Remarkably, the former's anticipated performance advantage turned out to be hardly perceptible. Possibly there are cache-coherence issues introduced by several workers writing simultaneously to the same shared-memory area.

8.2.1 Starting and stopping the workers

The workers are usually started and stopped automatically by CUBA's integration routines, but the user may choose to start them manually or keep them running after one integration and shut them down later, e.g. at the end of the program, which can be slightly more efficient. The latter mode is referred to as 'Spinning Cores' and must be employed with certain care, for running workers will not 'see' subsequent changes in the main program's data (e.g. global variables, common blocks) or code (e.g. via `dlsym`) unless special arrangements are made (e.g. shared memory).

The spinning cores are controlled through the '`spin`' argument of the CUBA integration routines (Sect. 7.1.1):

- A value of `-1` or `%VAL(0)` (in Fortran) or `NULL` (in C/C++) tells the integrator to start and shut down the workers autonomously. This is the usual case. No workers will still be running after the integrator returns. No special precautions need to be taken to communicate e.g. global data to the workers. Note that it is expressly allowed to pass a 'naive' `-1` (which is an `integer`, not an `integer*8`) in Fortran.

- Passing a zero-initialized variable for `spin` instructs the integrator to start the workers but keep them running on return and store the 'spinning cores' pointer in `spin` for future use. The spinning cores must later be terminated explicitly by `cubawait`, thus invocation would schematically look like this:

```
integer*8 spin                    void *spin = NULL;
spin = 0
call vegas(..., spin, ...)        Vegas(..., &spin, ...);
...                               ...
call cubawait(spin)               cubawait(&spin);
```

- A non-zero `spin` variable is assumed to contain a valid 'spinning cores' pointer either from a former integration or an explicit invocation of `cubafork`, as in:

```
integer*8 spin                    void *spin;
call cubafork(spin)               cubafork(&spin);
call vegas(..., spin, ...)        Vegas(..., &spin, ...);
...                               ...
call cubawait(spin)               cubawait(&spin);
```

8.2.2 Accelerators and Cores

Based on the strategy used to distribute samples, CUBA distinguishes two kinds of workers.

Workers of the first kind are referred to as 'Accelerators' even though CUBA does not actually send anything to a GPU or Accelerator in the system by itself – this can only be done by the integrand routine. The assumption behind this strategy is that the integrand evaluation is running on a device so highly parallel that the sampling time is more or less independent of the number of points, up to the number of threads p_{accel} available in hardware. CUBA tries to send exactly p_{accel} points to each core – never more, less only for the last batch. To sample e.g. 2400 points on three accelerators with $p_{\text{accel}} = 1000$, CUBA sends batches of 1000/1000/400 and not, for example, 800/800/800 or 1200/1200. The number of accelerators n_{accel} and their value of p_{accel} can be set through the environment variables

`CUBAACCEL=`n_{accel}	(default: 0)
`CUBAACCELMAX=`p_{accel}	(default: 1000)

or, superseding the environment, an explicit

`call cubaaccel(`n_{accel}`,` p_{accel}`)`

CPU-bound workers are just called 'Cores'. Their distribution strategy is different in that all available cores are used and points are distributed evenly. In the example above, the batches would be 800/800/800 thus. Each core receives at least 10 points, or else fewer cores are used. If no more than 10 points are requested in total, CUBA uses no workers at all but lets the master sample those few points. This happens during the partitioning phase of Divonne, for instance, where only single points are evaluated in the minimum/maximum search. Conversely, if the division of points by cores does not come out even, the remaining few points ($< n_{\text{cores}}$) are simply added to the existing batches, to avoid an extra batch because of rounding. Sampling 2001 points on two cores with $p_{\text{cores}} = 1000$ will hence give two batches 1001/1000 and not three batches 1000/1000/1.

Although there is typically no hardware limit, a maximum number of points per core, p_{cores}, can be prescribed for Cores, too. Unless the integrand is known to evaluate equally fast at all points, a moderate number for p_{cores} (10000, say) may actually increase performance because it effectively load-levels the sampling. For, a batch always goes to the next free core so it doesn't matter much if one core is tied up with a batch that takes longer.

The number of cores n_{cores} and the value of p_{cores} can be set analogously through the environment variables

`CUBACORES=`n_{cores}	(default: no. of idle cores)
`CUBACORESMAX=`p_{cores}	(default: 10000)

If `CUBACORES` is unset, the idle cores on the present system are taken (total cores minus load average), which means that a program calling a CUBA routine will by default automatically parallelize on the available cores. Again, the environment can be overruled with an explicit

`call cubacores(`n_{cores}`,` p_{cores}`)`

Using the environment has the advantage, though, that changing the number of cores to use does not require a re-compile, which is particularly useful if one wants to run the program on several computers (with potentially different numbers of cores) simultaneously, say in a batch queue.

The integrand function may use the '`core`' argument (Sect. 7.1.1) to distinguish Accelerators (`core` < 0) and Cores (`core` $\geqslant 0$). The special value `core` $= 32768$ (2^{15}) indicates that the master itself is doing the sampling.

8.2.3 Worker initialization

User subroutines for (de)initialization may be registered with

```
call cubainit(initfun, initarg)          Fortran
call cubaexit(exitfun, exitarg)

cubainit(initfun, initarg);              C/C++
cubaexit(exitfun, exitarg);
```

and will be executed in every process before and after sampling. Passing a null pointer (`%VAL(0)` in Fortran, `NULL` in C/C++) as the first argument unregisters either subroutine.

The init/exit functions are actually called as

```
call initfun(initarg, core)              Fortran
call exitfun(exitarg, core)

initfun(initarg, &core);                 C/C++
exitfun(exitarg, &core);
```

where `initarg` and `exitarg` are the user arguments given with the registration (arbitrary in Fortran, `void *` in C/C++) and `core` indicates the core the function is being executed on, with (as before) `core` < 0 for Accelerators, `core` $\geqslant 0$ for Cores, and `core` $= 32768$ for the master.

On worker processes, the functions are respectively executed after `fork` and before `wait`, independently of whether the worker actually receives any samples. The master executes them only when actual sampling is done. For Accelerators, the init and exit functions are typically used to set up the device for the integrand evaluations, which for many devices must be done per process, i.e. after the `fork`.

8.2.4 Concurrency issues

By creating a new process image, `fork` circumvents all memory concurrency, to wit: each worker modifies only its own copy of the parent's memory and never overwrites any other's data. The programmer should be aware of a few potential problems nevertheless:

- Communicating back results other than the intended output from the integrand to the main program is not straightforward because, by the same token, a worker cannot overwrite any common data of the master, it will only modify its own copy.

  Data exchange between workers is likewise not directly possible. For example, if one worker stores an intermediate result in a common block, this will not be seen by the other workers.

  Possible solutions include using shared memory (`shmget` etc., see App. A) and writing the output to file (but see next item below).

- `fork` does not guard against competing use of other common resources. For example, if the integrand function writes to a file (debug output, say), there is no telling in which order the lines will end up in the file, or even if they will end up as complete lines at all. Buffered output should be avoided at the very least; better still, every worker should write the output to its own file, e.g. with a filename that includes the process id, as in:

```
character*32 filename
integer pid
data pid /0/
if( pid .eq. 0 ) then
  pid = getpid()
  write(filename,'("output.",I5.5)') pid
  open(unit=4711, file=filename)
endif
```

- Fortran users are advised to flush (or close) any open files before calling Cuba, i.e. `call flush(`*unit*`)`. The reason is that the child processes inherit all file buffers, and *each* of them will write out the buffer content at exit. Cuba preemptively flushes the system buffers already (`fflush(NULL)`) but has no control over Fortran's buffers.

For debugging, or if a malfunction due to concurrency issues is suspected, a program should be tested in serial mode first, e.g. by setting `CUBACORES` = 0 (Sect. 8.2.2).

8.2.5 Vectorization

Vectorization means evaluating the integrand function for several points at once. This is also known as Single Instruction Multiple Data (SIMD) paradigm and is different from

ordinary parallelization where independent threads are executed concurrently. It is usually possible to employ vectorization on top of parallelization.

Vector instructions are commonly available in hardware, e.g. on x86 platforms under acronyms such as SSE or AVX. Language support varies: Fortran 90's syntax naturally embeds vector operations. Many C/C++ compilers offer auto-vectorization options, some have extensions for vector data types (usually for a limited set of mathematical functions), and even hardware-specific access to the CPU's vector instructions. And then there are vectorized libraries of numerical functions available.

Cuba cannot automatically vectorize the integrand function, of course, but it does pass (up to) `nvec` points per integrand call (Sect. 7.1.1). This value need not correspond to the hardware vector length – computing several points in one call can also make sense e.g. if the computations have significant intermediate results in common. The actual number of points passed is indicated through the corresponding `nvec` argument of the integrand.

A note for disambiguation: The `nbatch` argument of Vegas is related in purpose but not identical to `nvec`. It internally partitions the sampling done by Vegas but has no bearing on the number of points given to the integrand. On the other hand, it it pointless to choose `nvec` > `nbatch` for Vegas.

9 Tests and Comparisons

Four integration routines may seem three too many, but as the following tests show, all have their strengths and weaknesses. Fine-tuning the algorithm parameters can also significantly affect performance.

In the following, the test suite of Genz [14] is used. Rather than testing individual integrands, Genz proposes the following six families of integrands:

$$
\begin{aligned}
&\text{1. Oscillatory:} && f_1(\mathbf{x}) = \cos(\mathbf{c}\cdot\mathbf{x} + 2\pi w_1)\,,\\
&\text{2. Product peak:} && f_2(\mathbf{x}) = \prod_{i=1}^{n_d} \frac{1}{(x_i - w_i)^2 + c_i^{-2}}\,,\\
&\text{3. Corner peak:} && f_3(\mathbf{x}) = \frac{1}{(1 + \mathbf{c}\cdot\mathbf{x})^{n_d+1}}\,,\\
&\text{4. Gaussian:} && f_4(\mathbf{x}) = \exp(-\mathbf{c}^2(\mathbf{x} - \mathbf{w})^2)\,,\\
&\text{5. } C^0\text{-continuous:} && f_5(\mathbf{x}) = \exp(-\mathbf{c}\cdot|\mathbf{x} - \mathbf{w}|)\,,\\
&\text{6. Discontinuous:} && f_6(\mathbf{x}) = \begin{cases} 0 & \text{for } x_1 > w_1 \vee x_2 > w_2\,,\\ \exp(\mathbf{c}\cdot\mathbf{x}) & \text{otherwise.}\end{cases}
\end{aligned}
\tag{3}
$$

Parameters designated by w are non-affective, they vary e.g. the location of peaks, but should in principle not affect the difficulty of the integral. Parameters designated by c are

affective and in a sense "define" the difficulty of the integral, e.g. the width of peaks are of this kind. The c_i are positive and the difficulty increases with $\|\mathbf{c}\|_1 = \sum_{i=1}^{n_d} c_i$.

The testing procedure is thus: Choose uniform random numbers from $[0, 1)$ for the c_i and w_i. Renormalize $\mathbf{c}$ for a given difficulty. Run the algorithms with the integrands thus determined. Repeat this procedure 20 times and take the average.

For comparison, Mathematica's `NIntegrate` function was included in the test. Unfortunately, when a maximum number of samples is prescribed, `NIntegrate` invariably uses non-adaptive methods, by default the Halton–Hammersley–Wozniakowski quasi-Monte Carlo algorithm. The comparison may thus seem not quite balanced, but this is not entirely true: Lacking an upper bound on the number of integrand evaluations, `NIntegrate`'s adaptive method in some cases 'locks up' (spends an inordinate amount of time and samples) and the user can at most abort a running calculation, but not extract a preliminary result. The adaptive method could reasonably be used only for some of the integrand families in the test, and it was felt that such a selection should not be done, as the comparisons should in the first place give an idea about the *average* performance of the integration methods, without any fine-tuning.

Table 1 gives the results of the tests as described above. This comparison chart should be interpreted with care, however, and serves only as a rough measure of the performance of the integration methods. Many integrands appearing in actual calculations bear few or no similarities with the integrand families tested here, and neither have the integration parameters been tuned to 'get the most' out of each method.

The Mathematica code of the test suite is included in the downloadable Cuba package.

10 Summary

The Cuba library offers a choice of four independent routines for multidimensional numerical integration: Vegas, Suave, Divonne, and Cuhre. They work by very different methods, summarized in the following table:

Routine	Basic integration method	Algorithm type	Variance reduction
Vegas	Sobol quasi-random sample	Monte Carlo	importance sampling
	or pseudo-random sample	Monte Carlo	
Suave	Sobol quasi-random sample	Monte Carlo	globally adaptive subdivision
	or pseudo-random sample	Monte Carlo	
Divonne	Korobov quasi-random sample	Monte Carlo	stratified sampling,
	or Sobol quasi-random sample	Monte Carlo	aided by methods from
	or pseudo-random sample	Monte Carlo	numerical optimization
	or cubature rules	deterministic	
Cuhre	cubature rules	deterministic	globally adaptive subdivision

$n_d = 5$

j	Vegas	Suave	Divonne	Cuhre	NIntegrate
1	162000 ± 0	127300 ± 32371	21313 ± 11039	819 ± 0	218281 ± 0
2	11750 ± 1795	13500 ± 1539	17353 ± 3743	56238 ± 40917	218281 ± 0
3	16125 ± 2411	11500 ± 1000	17208 ± 2517	1174 ± 444	218281 ± 0
4	56975 ± 11372	20100 ± 4745	19636 ± 6159	22577 ± 31424	218281 ± 0
5	14600 ± 3085	15250 ± 2337	21675 ± 4697	150423 ± 0	218281 ± 0
6	19750 ± 4999	23850 ± 2700	39694 ± 14001	1884 ± 215	218281 ± 0

$n_d = 8$

j	Vegas	Suave	Divonne	Cuhre	NIntegrate
1	153325 ± 20274	124350 ± 35467	28463 ± 31646	3315 ± 0	212939 ± 13557
2	12650 ± 1987	21050 ± 4594	22030 ± 3041	91826 ± 58513	218281 ± 0
3	24325 ± 3753	29350 ± 3588	67104 ± 16906	18785 ± 22354	218281 ± 0
4	38575 ± 16169	29250 ± 8873	24849 ± 5015	62322 ± 44328	218281 ± 0
5	15150 ± 2616	25500 ± 6444	32885 ± 5945	151385 ± 0	218281 ± 0
6	18875 ± 2512	40900 ± 7196	116744 ± 32533	9724 ± 9151	218281 ± 0

$n_d = 10$

j	Vegas	Suave	Divonne	Cuhre	NIntegrate
1	156050 ± 21549	129800 ± 30595	32176 ± 30424	7815 ± 0	214596 ± 16481
2	14175 ± 2672	24800 ± 5464	25684 ± 7582	144056 ± 25983	218281 ± 0
3	30275 ± 6296	51150 ± 15608	139737 ± 18505	109150 ± 58224	218281 ± 0
4	29475 ± 10277	34050 ± 10200	27385 ± 8498	105763 ± 49789	218281 ± 0
5	16150 ± 2791	31400 ± 7715	44393 ± 18654	153695 ± 0	218281 ± 0
6	22100 ± 3085	74900 ± 32203	136508 ± 17067	73200 ± 64621	218281 ± 0

Test parameters:

- number of dimensions: $n_d = 5, 8, 10$,
- requested relative accuracy: $\varepsilon_{\text{rel}} = 10^{-3}$,
- maximum number of samples: $n_s^{\max} = 150000$,
- integrand difficulties:

Integrand family j	1	2	3	4	5	6
$\|\mathbf{c}_j\|_1$	6.0	18.0	2.2	15.2	16.1	16.4

Table 1: The number of samples used, averaged from 20 randomly chosen integrands from each integrand family j defined in Eq. (3). Values in the vicinity of $n_s^{\max}$ generally indicate failure to converge. `NIntegrate` seems not to be able to stop at around the limit of `MaxPoints ->` $n_s^{\texttt{max}}$, but always samples considerably more points.

All four have a C/C++, Fortran, and Mathematica interface and can integrate vector integrands. Their invocation is very similar, so it is easy to substitute one method by another for cross-checking. For further safeguarding, the output is supplemented by a χ^2 probability which quantifies the reliability of the error estimate.

The source code is available from `http://feynarts.de/cuba` and compiles with gcc, the GNU C compiler. The C functions can be called from Fortran directly, so there is no need for adapter code. Similarly, linking Fortran code with the library is straightforward and requires no extra tools.

The routines in the CUBA library have all been carefully tested, but it would of course be folly to believe they are completely error-free. The author welcomes any kind of feedback, in particular bug and performance reports, at hahn@feynarts.de.

Acknowledgements

I thank A. Hoang for involving me in a discussion out of which the concept of the Mathematica interface was born and T. Fritzsche, M. Rauch, and A.M. de la Ossa for testing. B. Chokoufe implemented check-pointing (state file) for Suave, Divonne, and Cuhre.

A Shared Memory in Fortran

IPC shared memory is not natively available in Fortran, but it is not difficult to make it available using two small C functions `shmalloc` and `shmfree`:

```
#include <sys/shm.h>
#include <assert.h>

typedef long long int memindex;
typedef struct { void *addr; int id; } shminfo;

void shmalloc_(shminfo *base, memindex *i, const int *n, const int *size) {
  base->id = shmget(IPC_PRIVATE, *size*(*n + 1) - 1, IPC_CREAT | 0600);
  assert(base->id != -1);
  base->addr = shmat(base->id, NULL, 0);
  assert(base->addr != (void *)-1);
  *i = ((char *)(base->addr + *size - 1) - (char *)base)/(long)*size;
}

void shmfree_(shminfo *base) {
  shmdt(base->addr);
  shmctl(base->id, IPC_RMID, NULL);
}
```

The function `shmalloc` allocates (suitably aligned) `n` elements of size `size` and returns a mock index into `base`, through which the memory is addressed in Fortran. The array `base` must be of the desired type and large enough to store the struct `shminfo`, e.g. two doubles wide. Be careful to invoke `shmfree` after use, for the memory will not automatically be freed upon exit but stay allocated until the next reboot (or explicit removal with `ipcs`).

The following test program demonstrates how to use `shmalloc` and `shmfree`:

```
program test
implicit none
integer*8 i
double precision base(2)

call shmalloc(base, i, 100, 8)     ! allocate 100 doubles

base(i) = 1                        ! now use the memory
...
base(i+99) = 100

call shmfree(base)                 ! don't forget to free it
end
```

References

[1] R. Piessens, E. de Doncker, C. Überhuber, D. Kahaner, QUADPACK – a subroutine package for automatic integration, Springer-Verlag, 1983.

[2] G.P. Lepage, *J. Comp. Phys.* **27** (1978) 192.

[3] G.P. Lepage, Report CLNS-80/447, Cornell Univ., Ithaca, N.Y., 1980.

[4] W.H. Press, G.R. Farrar, *Comp. in Phys.* **4** (1990) 190.

[5] J.H. Friedman, M.H. Wright, *ACM Trans. Math. Software* **7** (1981) 76;
J.H. Friedman, M.H. Wright, SLAC Report CGTM-193-REV, CGTM-193, 1981.

[6] J. Berntsen, T. Espelid, A. Genz, *ACM Trans. Math. Software* **17** (1991) 437;
J. Berntsen, T. Espelid, A. Genz, *ACM Trans. Math. Software* **17** (1991) 452;
TOMS algorithm 698.

[7] P. Bratley, B.L. Fox, *ACM Trans. Math. Software* **14** (1988) 88;
TOMS algorithm 659.

[8] H. Niederreiter, Random number generation and quasi-Monte Carlo methods, SIAM, 1992.

[9] W.H. Press, S.A. Teukolsky, W.T. Vetterling, B.P. Flannery, Numerical recipes in Fortran, 2nd edition, Cambridge University Press, 1992.

[10] N.M. Korobov, Number theoretic methods in approximate analysis (in Russian), Fizmatgiz, Moscow, 1963.

A comprehensive English reference on the topic of good lattice points (of which the Korobov points are a special case) is H.L. Keng, W. Yuan, Applications of number theory to numerical analysis, Springer-Verlag, 1981.

[11] M. Matsumoto, T. Nishimura, *ACM Trans. Modeling Comp. Simulation* **8** (1998) 3. See also `http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html`.

[12] M. Lüscher, *Comp. Phys. Commun.* **79** (1994) 100;
F. James, *Comp. Phys. Commun.* **79** (1994) 111.

[13] A. Genz, A. Malik, *SIAM J. Numer. Anal.* **20** (1983) 580.

[14] A. Genz, A package for testing multiple integration subroutines, in: P. Keast, G. Fairweather (eds.), Numerical Integration, Kluwer, Dordrecht, 1986.