Edinburgh 2002/17

SINGINT: Automatic numerical integration of singular integrands

Nikolas Kauer

School of Physics, University of Edinburgh, Edinburgh EH9 3JZ, UK

Abstract

We explore the combination of deterministic and Monte Carlo methods to facilitate efficient automatic numerical computation of multidimensional integrals with singular integrands. Two adaptive algorithms are presented that employ recursion and are runtime and memory optimised, respectively. SINGINT, a C implementation of the algorithms, is introduced and its utilisation in the calculation of particle scattering amplitudes is exemplified.

Key words: Adaptive numerical integration, multidimensional quadrature, singular integrands, integration rules, Monte Carlo integration, recursion.

Email address: `nkauer@ph.ed.ac.uk` (Nikolas Kauer).

Program summary

Title of program: SINGINT
Program obtainable from: `http://www.ph.ed.ac.uk/~nkauer/singint/`
Program requirements: SINGINT can be built with any compatible C and Fortran compilers. It requires GNU Scientific Library[1] 1.2. The program has been tested with GNU Compiler Collection[2] 3.1 on Red Hat Linux 7.1.
Programming language: C, F77
No. of bytes in distributed program: 26224
Distribution format: gzip-compressed tar file

Nature of physical problem:
Efficient, robust and automatic numerical computation of multidimensional integrals with singular integrands, for example occurring in the calculation of particle scattering amplitudes in perturbative field theory.

Method of solution:
Runtime- or memory-optimised algorithms that combine integration rule and Monte Carlo techniques and employ recursion.

Restrictions on the complexity of the problem:
`singint_rec`, the implementation of the recursive algorithm, is suitable for integrands of up to 12 dimensions. `singint_cut`, the implementation of the global algorithm, is currently restricted to two dimensions. An extension of the algorithm to more than two dimensions is straightforward and will be implemented in a future version.

[1] `http://sources.redhat.com/gsl/`
[2] `http://gcc.gnu.org`

1 Introduction

While the practitioner can rely on highly efficient and robust automatic routines for the numerical integration of 1-dimensional integrals [1], for multidimensional integrals the situation is more complicated and no mature universal tool has emerged yet.

To guide our discussion we start by recalling the key features of a superior numerical integration routine: It should be efficient, robust and automatic. Efficient means that results of the desired precision are obtained using as little resources as possible in terms of processor time and memory. Robust means that the routine yields accurate results for a variety of integrands of interest, for example, not only smooth ones, but also discontinuous or singular integrands. Automatic implies that the routine requires minimal information about the integrand, or, more precisely, that a "black box" that returns the value of the integrand for any point in the integration volume is all that is required. To devise one routine that excels in all three categories is extremely challenging, but a number of complementary approaches have been developed to satisfy more limited expectations.

Two basic approaches to numerical integration are deterministic integration with integration rules and Monte Carlo integration [2]. Both methods have strengths and weaknesses. Integration rules, on the one hand, yield precise results with relatively few integrand evaluations, but they are not too robust and work best for very smooth functions. Monte Carlo methods, on the other hand, impose few requirements on the integrand, but are known to converge slowly. To bridge the gap, adaptive procedures have been developed, that select and evaluate suitable subregions separately, thus making deterministic integration more robust and Monte Carlo integration more efficient.

As mentioned above, the use of Monte Carlo techniques is suggestive when multidimensional, square-integrable functions with singularities need to be integrated. But, when fast integration or high precision are mandatory pure Monte Carlo methods are not well suited. To enable rapid convergence, integration rule based approaches for singular integrands have been explored in the literature [3]. As a result, highly efficient integrators exist that exploit extrapolation methods and can be used if the location of all singularities is known in analytic form. However, such procedures are not automatic and not applicable if the location of singularities is unknown or difficult to determine.

In this article we therefore present two *automatic* procedures for the integration of low-dimensional singular integrands that aim to retain the performance advantage of deterministic integration methods. This is achieved by the combined use of integration rules and Monte Carlo sampling. To the best of our knowledge, this combination has not previously been proposed in the literature.

A second objective of this article is to explore the applicability of recursion to adaptive integration. Numerical algorithms have traditionally been implemented in Fortran 77, which does not allow recursion. [3] One might therefore conjecture that recursive techniques have not been explored by practitioners to the desirable extent. To avoid being limited by such technicalities, SINGINT is implemented in C, which allows to conveniently program with recursion. [4]

The article is organised as follows: In Sections 2 and 3, we present two efficient algorithms for the automatic numerical integration of singular integrands, which combine integration rule and Monte Carlo techniques. The first algorithm is runtime optimised, while the second is fully recursive and has small memory requirements. In Section 4, we introduce the interface of our implementation and discuss important aspects of SINGINT's use based on a short example. In Section 5, we give numerical examples and discuss a practical application in theoretical particle physics. The article closes with conclusions.

2 Global Algorithm

2.1 Method

The global algorithm we suggest decomposes the singular integrand $f(x)$ into a bounded function $b_c(x)$ and a singular rest $s_c(x)$ by introducing a cut parameter $c > 0$:

$$b_c(x) := \begin{cases} +c & \text{if } f(x) > c \\ f(x) & \text{if } -c \leq f(x) \leq c \\ -c & \text{if } f(x) < -c \end{cases} \tag{1}$$

$$s_c(x) := f(x) - b_c(x) \tag{2}$$

The bounded component $b_c(x)$ can then be integrated relying on automatic deterministic methods for maximum efficiency. To integrate the singular com-

[3] Recursive behaviour can and has nevertheless been implemented in Fortran 77 through simulated recursion, which requires unintuitive code. We also note that this deficiency has been remedied in the Fortran 90 standard.
[4] Our choice was also influenced by the fact that the Fortran compiler of the widely-used GNU compiler collection does not support recursion.

ponent $s_c(x)$ we revert to Monte Carlo integration, an approach that is well suited for irregular or singular integrands and requires no analytic knowledge about the structure of the integrand, thus being automatic, too. These advantages are, however, offset by significantly slower convergence relative to deterministic methods. With regard to our integrand decomposition, this interplay suggests the existence of an optimal range for the cut parameter c. Below that range, unnecessarily large, non-singular regions are Monte Carlo integrated, causing slow overall convergence. Above that range, the deterministic routine has to integrate exceedingly steep peaks in the singular regions, necessitating a high number of function evaluations, which again leads to slow overall convergence. For cut parameters in the optimal range, however, the two effects balance and efficiency is maximal. After both components have been computed, the final result is obtained by adding up the integrals over $b_c(x)$ and $s_c(x)$. It is obviously independent of the cut parameter c.

2.2 Optimisation

We employ an optimised approach to the Monte Carlo integration of $s_c(x)$ that requires a 2-dimensional grid covering the integration region. During the integration of $b_c(x)$, some evaluations of $f(x)$ may return values above or below the cuts, i.e. $s_c(x) \neq 0$. In this case the grid cell that contains x is saved. In a second step the integral over $s_c(x)$ is calculated by using all cells with $s_c(x) \neq 0$ that were detected in the previous step as seed cells. Note that the number of seed cells found will typically be much smaller than the total number of cells as long as the cut parameter c is chosen sufficiently large. $s_c(x)$ is integrated by using crude Monte Carlo integration to evaluate all contributing cells using a recursive algorithm that starts with seed cells. For Monte Carlo integration in low dimensions (less than ca. 15 dimensions according to the analysis in Ref. [4]) convergence can be improved significantly by using quasi-random numbers rather than pseudo-random numbers. We therefore base the sampling in our implementation on the low-discrepancy sequence of Ref. [5]. If the result of a particular cell is finite, all neighbouring cells are also scheduled for evaluation. This procedure is applied recursively until the region where $s_c(x)$ is nonvanishing is covered. The recursive algorithm we employ is given below as Algorithm 1 (using orientations to identify neighbouring cells). Note that other seed cells encountered during execution are removed from the list of remaining seed cells. The recursion stops when a cell evaluates to zero and is therefore at the boundary of the integration region for $s_c(x)$. We refer to such cells as “border” cells.

The recursion spreads in a “starlight-like” fashion and is therefore only suited to cover convex areas. This limitation can be overcome with the following extension: Apply the algorithm to one seed cell. Then make every encountered

Algorithm 1. recursive coverage of convex grid area

```
if valid cell then
   process cell
   if cell is start cell then
      process cells N, NW, W, SW, S, SE, E, NE
   else
      if cell type is N then
         process cell N
      else if cell type is NW then
         process cells N, NW, W
      else if cell type is W then
         process cell W
      else if cell type is SW then
         process cells W, SW, S
      else if cell type is S then
         process cell S
      else if cell type is SE then
         process cells S, SE, E
      else if cell type is E then
         process cell E
      else if cell type is NE then
         process cell E, NE, N
      end if
   end if
end if
```

border cell a start cell and apply the algorithm again. (The border cells of the secondary runs are not made start cells.) This extension makes the procedure more robust, but made little or no difference in our test runs. To rule out repeated evaluations of the same cell, the status of each cell is tagged with one of the following labels: unknown, finite, zero. Initially all cells are tagged "unknown". Cells tagged "finite" have already been evaluated and taken into account, while cells tagged "zero" have already been identified as border cells. Note that tagging each cell introduces a global element in the recursion, which is permissible since the algorithm used to evaluate $b_c(x)$ is also global in nature (see below). Once the enhanced algorithm has been run for all seed cells the integration of $s_c(x)$ is complete.

To optimise the integration of $b_c(x)$ we utilize DCUHRE, a very efficient and robust integrator for multidimensional integrals with *bounded* integrands developed by Berntsen *et al.* [6]. Its adaptive algorithm applies integration rules to determine the integral and error over non-uniform subvolumes until the desired total error is achieved. The algorithm is global, i.e. results for all subvolumes are retained, to be able to select at any stage the subvolume with the largest error contribution for further refinement. This approach is optimal in terms of runtime but expensive in terms of memory. We return to this issue in Section 3.

3 Recursive Algorithm

Due to its global nature the approach described in Section 2 requires a potentially large amount of memory, and the question arises if a viable "local" alternative with small memory footprint can be found. To that end, we propose a second, fully recursive approach. Assume the integral I_0 over a hypercube with volume V_0 is to be determined with precision ΔI_0. Starting with volume V_0 the following procedure is applied recursively:

(1) A value I and error estimate ΔI for the integral in the cell of volume V is obtained by applying
   a) an integration rule (ca. 200 integrand evaluations are necessary for a degree 13 integration rule [6])
   b) basic Monte Carlo integration with the same number of function evaluations as in a)

   If both results are compatible within errors, the one with the lower error estimate is selected, otherwise the result with the larger error is selected.

(2) The tolerable error[5] in the cell is $\Delta I_{max} := \Delta I_0 \sqrt{V/V_0}$.

(3) If $\Delta I \leq \Delta I_{max}$, no further action is necessary. If $\Delta I > \Delta I_{max}$, the cell is divided into n subcells of equal volume, and the integrals I_i in the subcells are determined as in (1).

(4) If $\Delta I_{div} < \Delta I/\sqrt{n}$ with $\Delta I_{div} := [\sum_{i=1}^{n} (\Delta I_i)^2]^{1/2}$, the procedure is applied recursively to the subcells.

(5) If $\Delta I_{div} \geq \Delta I/\sqrt{n}$, further subdivision is not advantageous, and I is Monte Carlo sampled until $\Delta I \leq \Delta I_{max}$.

This procedure clearly shows that in general the algorithm that controls the subdivision of the integration volume is conceptually distinct from the algorithm used to integrate individual subcells. To let several complementary integration methods compete in the latter adds flexibility and should typically increase efficiency. Our experience in the context of the practical application described in Section 5.2 confirms this hypothesis.

[5] This condition guarantees that the overall error is at most ΔI_0.

4 Using SINGINT

To get an overview of SINGINT's interface, the header file `singint.h` is a good starting point:

```
typedef void Integrand(const int* ndim, const double x[],
                       const int* numfun, double funvls[]);

typedef void Integrator(const int ndim, const double a[], const double b[],
                        Integrand* integrand, const double desired_abserr,
                        const double desired_relerr, const int maxpts,
                        double* result, double* abserr);

Integrator singint_mc;

Integrator singint_ir;

Integrator singint_rec;

Integrator singint_cut;
extern double singint_cut_parameter;
void singint_cut_callback(const int* ndim, const double x[],
                          const int* numfun, double funvls[]);
```

All integrand routines have to implement the function signature `Integrand`, which essentially takes a point in the integration volume specified by array `x` with `*ndim` elements as input and returns the integrand value at this point in array element `funvls[0]`. In order to avoid an additional, performance reducing integrand wrapper when applying integration rules, pointers are used to satisfy Fortran calling conventions and the dummy argument `*numfun` (restricted to 1) is introduced to create compatibility with DCUHRE.

The integrand routine is one of the arguments of the `Integrator` signature, which all integrator functions implement. A list of all arguments with type and short description follows:

`ndim: const int` number of dimensions of integral
`a[]: const double` lower limits of hypercubical integration volume
`b[]: const double` upper limits of hypercubical integration volume
`integrand: Integrand*` (pointer to) integrand function
`desired_abserr: const double` desired absolute error of integration result
`desired_relerr: const double` desired relative error of integration result
`maxpts: const int` maximum number of integrand evaluations ($\geq$ minimum given by `IR_MINPOINTSPERCELL`)
`result: double*` integral estimate
`abserr: double*` estimated absolute error of `result`

The global algorithm (with cut parameter) and the recursive algorithm defined above are implemented through the functions `singint_cut` and `singint_rec`, respectively. For convenience, the underlying techniques, i.e. Monte Carlo and integration rule-based integration, are implemented in integrators `singint_mc` and `singint_ir`, respectively. Note that `singint_ir` is currently a wrapper function for DCUHRE and hence per se not suited for singular integrands (see Section 2). A concise but complete example that illustrates how to use SINGINT follows:

```
#include <math.h>
#include <stdio.h>
#include "singint.h"

int called_by_singint_cut = 0;

void integrand(const int* ndim, const double x[],
             const int* numfun, double funvls[])
{
  funvls[0] = singular_function(ndim, x);
  if (called_by_singint_cut)
    singint_cut_callback(ndim, x, numfun, funvls);
}

int main(int argc, char* argv[])
{
  const double a[2] = { 0., 0. };
  const double b[2] = { 1., 1. };
  double result, abserr;

  singint_cut_parameter = 100.;
  called_by_singint_cut = 1;
  singint_cut(2, a, b, &integrand, 0.01, 0., 10000000, &result, &abserr);
  called_by_singint_cut = 0;
  printf("singint_cut result:  %.10g +- %.2g\n", result, abserr);

  singint_rec(2, a, b, &integrand, 0.01, 0., 10000000, &result, &abserr);
  printf("singint_rec result:  %.10g +- %.2g\n", result, abserr);

  return 0;
}
```

As shown, when using `singint_cut` it is first necessary to set the external global variable `singint_cut_parameter` to a suitable positive value. Moreover, additional post-processing is necessary after integrand evaluations, which is facilitated by calling `singint_cut_callback` in the integrand routine. A transparent code organization to that effect (involving the Boolean variable `called_by_singint_cut`) is also shown in the preceding example.

The argument `maxpts` has slightly different meanings for `singint_cut` and `singint_rec`. For the former integrator it specifies the maximal number of integrand evaluations used when integrating the bounded component, while for the latter integrator it specifies the maximal number of integrand evaluations in the Monte Carlo integration of any *terminal* subcell, i.e. a subcell where further subdivision would not accelerate the integration.[6]

As pointed out above, `singint_rec` due to its recursive nature has comparably small memory requirements. `singint_cut`, however, implements a globally optimising strategy, and increased precision hence requires the allocation of a larger workspace. The size of the workspace for the integration of the bounded and singular integrand components is

$$\underbrace{\texttt{DCUHRE\_WKSPC\_SIZE} \cdot 4 \text{ bytes}}_{\int b_c(x)\,dx} + \underbrace{\texttt{MAXNRSEEDCELLS} \cdot 4 \text{ bytes} + \texttt{NRBINS}^2 \cdot 1 \text{ byte}}_{\int s_c(x)\,dx} .$$

For a given integrand, the optimal value of the ratio $\texttt{MAXNRSEEDCELLS}/\texttt{NRBINS}^2$ is approximately equal to the fraction of the integration volume where $s_c(x)$ is nonvanishing. The shape of this area also suggests an adequate resolution and thus a suitable value for `NRBINS`. More specifically, the default setting of `NRBINS` = 100 and `MAXNRSEEDCELLS` = 10000 requires about 50kB. For high precision calculations in connection with the application described in Section 5.2 we used `NRBINS` = 5000 and `MAXNRSEEDCELLS` = 500000, which required about 25MB. Without prior knowledge of the singularity structure of the integrand, one would have set `MAXNRSEEDCELLS` to 5000^2, increasing the required memory by about 100MB.

With respect to $\int b_c(x)\,dx$, up to 10^7 integrand evaluations require `DCUHRE_WKSPC_SIZE` = 615402, i.e. about 2.3MB, while `DCUHRE_WKSPC_SIZE` = 92307714 allocates 352MB and allows for up to $1.5 \cdot 10^9$ integrand evaluations with a degree 13 rule.[7] The function `singint_ir_checks` can be used to obtain the necessary workspace size for other `maxpts` settings.

To conclude this section, we mention a difficulty for non-global recursive algorithms in general and the `singint_rec` implementation in particular. Due to the depth-first evaluation of integral contributions, no reliable estimate for the total integral exists until the algorithm terminates. It is hence impossible to translate a specified relative error to a desired absolute error while integrating. Since relative errors are not additive, it is also not possible to adapt the

[6] Note that imposing such an upper limit in `singint_rec` effects a runtime guarantee, but also invalidates the general statement in footnote 5.

[7] Note that in this case `maxpts` is chosen to be very close to the maximum value of a signed 4-byte integer, i.e. $2^{31} - 1$. Changing data types in DCUHRE would allow to go beyond this implicit upper limit.

strategy employed to limit the absolute error. `singint_rec` therefore accepts only absolute error requests and requires that `desired_relerr` be set to zero.

5 Numerical examples and practical application

5.1 Numerical examples

Three 2-dimensional integrands $f_i(x,y)$ in the unit hypercube $U_2 := [0,1] \times [0,1]$ with corner and line singularities are used to construct example integrands $g_i(x,y)$ in $C_2 := [-1,1] \times [-1,1]$ with singularities at quasi-unknown locations inside the integration region. [8]

To be specific, the integrands $f_i(x,y)$ are given by

$$f_1(x,y) = \frac{1}{\sqrt{x^2+y^2}\,\sqrt[5]{x}\,\sqrt[3]{y}\,((x-0.5)^2+(y-0.5)^2+0.01)} \approx 32.640\,, \quad (3)$$

$$f_2(x,y) = \frac{\ln(x+y)\ln(x)\ln(y)e^{2x+y}}{\sqrt[9]{x+y}\,\sqrt[5]{x}\,\sqrt[7]{y}} \approx -4.5849\,, \quad (4)$$

$$f_3(x,y) = \frac{(\ln(x))^2 e^{x+y}\cos(20x)}{\sqrt[9]{x}(\sqrt[3]{y})^2} \approx 4.1960\,. \quad (5)$$

The integrands $g_i(x,y)$ are then defined by

$$g_i(x,y) = \begin{cases} f_i(x,y) & \text{if } (x,y) \in U_2 \\ f_i(-x,y) & \text{if } (x,y) \in [-1,0] \times [0,1] \\ f_i(x,-y) & \text{if } (x,y) \in [0,1] \times [-1,0] \\ f_i(-x,-y) & \text{if } (x,y) \in [-1,0] \times [-1,0] \end{cases} \quad (6)$$

with $i = 1,2,3$.

Evidently,

$$\int_{C_2} g_i(x,y)\, dxdy = 4 \int_{U_2} f_i(x,y)\, dxdy\,, \qquad i = 1,2,3\,. \quad (7)$$

[8] These integrands $g_i(x,y)$ can thus not be integrated with the non-automatic, deterministic algorithms of Ref. [3].

The SINGINT test program output displayed in App. A shows that the global algorithm implemented in `singint_cut`, as well as the recursive algorithm implemented in `singint_rec` can be used to efficiently integrate the example integrands $g_i(x, y)$. It also confirms that results obtained with `singint_cut` are indeed independent of the cut parameter. The test program further demonstrates that the integrands $f_i(x, y)$ as well as a number of trivial, bounded test integrands are also computed accurately. [9]

5.2 Practical application

SINGINT was developed to facilitate a novel numerical approach to the calculation of multileg 1-loop amplitudes in theoretical particle physics [7]. These amplitudes are important ingredients for the computation of higher-order corrections to multi-particle cross sections. The procedure involves the evaluation of 1- and 2-dimensional integrals that contain integrable singularities of logarithmic and square-root type. The number and location of these singularities is fairly complex and depends strongly on the underlying physical configuration as discussed in detail in Ref. [7]. A representative example is displayed in Fig. 1.

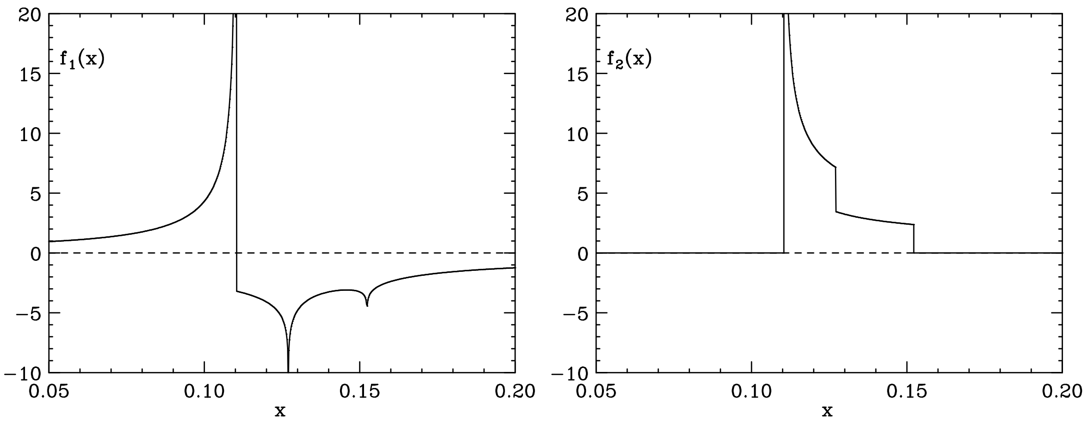

Fig. 1. Two functions $f_1(x)$ and $f_2(x)$ that illustrate the structure of the 1- and 2-dimensional integrands that are evaluated in the computation of multi-particle scattering amplitudes [7]. The peaks correspond to integrable square-root and logarithmic singularities. The structure of the 2-dimensional integrals is obtained by "layering" (with nontrivial offset) 1-dimensional cross sections of the type shown.

The ultimate objective is to be able to reliably compute these amplitudes for a multitude of physical configurations in time frames of 5 to 50 seconds with 1% relative error or better. Initial tests with pure Monte Carlo methods [8]

[9] These integrands can be integrated more efficiently by other means [3,6].

turned out to be 1 to 4 orders of magnitude too slow depending on the physical configuration. However, with the "mixed" algorithms described above we were able to achieve the desired runtimes. Generally, `singint_cut` performed better than `singint_rec` for the particular type of integrand considered here. We experimented with cut parameter values from 500 to 50000 and found the optimal window to range from about 5000 to 10000. When applying the recursive evaluation of subcells implemented in `singint_rec` we observed that neither the integration rule technique nor the Monte Carlo method dominated, which confirms the advantage of a mixed strategy in the case at hand. To further test the correctness of our implementation, we compared results obtained for special cases with corresponding results found in the literature. SINGINT also passed these application-related tests.

6 Conclusions

We introduced two efficient automatic procedures that facilitate the adaptive numerical integration of low-dimensional singular integrands. We showed how the combined use of integration rules and Monte Carlo sampling and the power of recursion enable new runtime and memory optimisations. An application in theoretical particle physics was discussed that demonstrates the practical value of our approach. To allow others to apply the algorithms, we described a C implementation with test examples and explained its use.

With the described physics application in mind, we formulated and implemented the global algorithm of Section 2 specifically for 2-dimensional integrals. However, its generalisation to more than two dimensions is straightforward. There are in fact applications in perturbative field theory, where higher-dimensional integrands similar to those of Section 5.2 need to be calculated numerically. We plan to investigate such cases in the future.

Acknowledgements

We would like to thank T. Binoth, A. Genz, B. Lautrup and J. Lyness for useful discussions.

References

[1] R. Piessens et al., QUADPACK, Springer Verlag, 1983.

[2] W.H. Press et al., Numerical Recipes, 2nd ed., Cambridge University Press, 1992.

[3] T.O. Espelid and A. Genz, Numerical Algorithms 8 (1994) 201 and references therein; K. Singstad and T.O. Espelid, J. Comp. Appl. Math. 112 (1999) 291; J.N. Lyness, Math. Comp. 30 (1976) 1.

[4] F. James, J. Hoogland and R. Kleiss, Comp. Phys. Comm. 99 (1997) 180.

[5] P. Bratley, B.L. Fox and H. Niederreiter, ACM Trans. Model. Comp. Sim. 2 (1992) 195.

[6] J. Berntsen, T.O. Espelid and A. Genz, ACM Trans. Math. Softw. 17 (1991) 437; J. Berntsen, T.O. Espelid and A. Genz, ACM Trans. Math. Softw. 17 (1991) 452.

[7] T. Binoth, G. Heinrich and N. Kauer, preprint Edinburgh 2002/16 (2002) [arXiv:hep-ph/0210023].

[8] G.P. Lepage, J. Comput. Phys. 27 (1978) 192; G.P. Lepage, preprint CLNS-80/447 (1980).

A Output of test program

```
singint_ir, simple1, unit h.c.:
1 +- 1.4e-14   (1.4e-14)
singint_mc, simple1, unit h.c.:
1 +- 0   (0)
singint_rec, simple1, unit h.c.:
1 +- 0   (0)
singint_cut, simple1, unit h.c., cut = 1.5:
1 +- 1.4e-14   (1.4e-14)
singint_cut, simple1, unit h.c., cut = 0.5:
1 +- 6.8e-15   (6.8e-15)
singint_ir, simple1, non-unit h.c.:
19.2 +- 2.6e-13   (1.4e-14)
singint_mc, simple1, non-unit h.c.:
19.2 +- 0   (0)
singint_rec, simple1, non-unit h.c.:
19.2 +- 0   (0)
singint_cut, simple1, non-unit h.c., cut = 1.5:
19.2 +- 2.6e-13   (1.4e-14)
singint_cut, simple1, non-unit h.c., cut = 0.5:
19.2 +- 1.3e-13   (6.8e-15)
singint_ir, simple2, non-unit h.c.:
26.88 +- 5.9e-14   (2.2e-15)
singint_mc, simple2, non-unit h.c.:
```

```
26.88006626 +- 0.2   (0.0074)
singint_rec, simple2, non-unit h.c.:
26.88 +- 5.9e-14   (2.2e-15)
singint_cut, simple2, non-unit h.c., cut = 7.0:
26.88 +- 5.9e-14   (2.2e-15)
singint_cut, simple2, non-unit h.c., cut = 1.0:
26.88387169 +- 0.19   (0.0069)
singint_cut, singcomp1, [-1, 1]^2, cut = 50000:
129.2232525 +- 1.4   (0.011)
singint_cut, singcomp1, [-1, 1]^2, cut = 10000:
129.7198049 +- 1.4   (0.011)
singint_cut, singcomp1, [-1, 1]^2, cut = 2000:
130.1922621 +- 1.3   (0.01)
singint_cut, singcomp2, [-1, 1]^2, cut = 50:
-18.32923723 +- 0.19   (0.01)
singint_cut, singcomp2, [-1, 1]^2, cut = 10:
-18.33036408 +- 0.18   (0.0099)
singint_cut, singcomp2, [-1, 1]^2, cut = 2:
-18.3208958 +- 0.16   (0.0088)
singint_cut, singcomp3, [-1, 1]^2, cut = 50000:
16.5502603 +- 0.17   (0.01)
singint_cut, singcomp3, [-1, 1]^2, cut = 10000:
16.72847529 +- 0.17   (0.01)
singint_cut, singcomp3, [-1, 1]^2, cut = 2000:
16.79531051 +- 0.17   (0.01)
singint_rec, singcomp1, [-1, 1]^2:
129.9953115 +- 1.3   (0.01)
singint_rec, singcomp2, [-1, 1]^2:
-18.33534209 +- 0.18   (0.0098)
singint_rec, singcomp3, [-1, 1]^2:
16.7794245 +- 0.17   (0.01)
```