

Cubature (Multi-dimensional integration)

From AbInitio

(Redirected from Cubature)

Contents

- 1 Cubature
- 2 Download
- 3 Usage
 - 3.1 "Vectorized" interface
 - 3.2 Example
 - 3.3 Infinite intervals
- 4 Test program

Cubature

Steven G. Johnson (<http://math.mit.edu/~stevenj>) has written a simple C subroutine for **adaptive multidimensional integration** (*cubature*) of **vector-valued integrands** over **hypercubes**, i.e. to compute integrals of the form:

$$\int_{a_1}^{b_1} \int_{a_2}^{b_2} \cdots \int_{a_n}^{b_n} \vec{f}(\vec{x}) d^n \vec{x}$$

(Of course, it can handle scalar integrands as the special case where \vec{f} is a one-dimensional vector: the dimensionalities of \vec{f} and \vec{x} are independent.) The integrand can be evaluated for an **array of points at once** to enable **easy parallelization**. The code, which is distributed as **free software** under the terms of the GNU General Public License (v2 or later), is based on the algorithms described in:

- A. C. Genz and A. A. Malik, "An adaptive algorithm for numeric integration over an N-dimensional rectangular region," *J. Comput. Appl. Math.* **6** (4), 295–302 (1980).
- J. Berntsen, T. O. Espelid, and A. Genz, "An adaptive algorithm for the approximate calculation of multiple integrals," *ACM Trans. Math. Soft.* **17** (4), 437–451 (1991).

This algorithm is **best suited for a moderate number of dimensions** (say, < 7), and is superseded for high-dimensional integrals by other methods (e.g. Monte Carlo variants or sparse grids).

(Note that we do *not* use any of the original DCUHRE code by Genz, which is not under a free/open-source license.) Our code is based in part on code borrowed from the HIntLib numeric-integration library (<http://mint.sbg.ac.at/HIntLib/>) by Rudolf Schürer and from code for Gauss-Kronrod quadrature (for 1d integrals) from the GNU Scientific Library (<http://www.gnu.org/software/gsl/>), both of which are free software under the GNU GPL. (Another free-software multi-dimensional integration library, unrelated to our code here but also implementing the Genz–Malik algorithm among other techniques, is Cuba (<http://www.feynarts.de/cuba/>).

I am also grateful to Dmitry Turbiner (dturbiner@alum.mit.edu), who implemented an initial prototype of the "vectorized" functionality (see below) for evaluating an array of points in a single call, which facilitates parallelization of the integrand evaluation.

Download

The current version of the code can be downloaded from:

- cubature-20101018.tgz (<http://ab-initio.mit.edu/cubature/cubature-20101018.tgz>)

a gzipped tar file. This unpacks to a directory containing a README file with instructions and a stand-alone cubature.c file that you can compile and link into your program, and a header file cubature.h that you #include. (The cubature.c code is small enough that it doesn't seem worthwhile to go to the trouble of installing it as a library.)

The cubature.c file also contains a little test program which is produced if you compile that file with -DTEST_INTEGRATOR (and is commented out otherwise), as described below.

B. Narasimhan wrote a GNU R interface for these routines, which can be downloaded here: <http://cran.r-project.org/web/packages/cubature/index.html>.

Usage

You should compile cubature.c and link it with your program, and #include the header file cubature.h.

The central subroutine you will be calling is:

```
int adapt_integrate(unsigned fdim, integrand f, void *fdata,
                   unsigned dim, const double *xmin, const double *xmax,
                   unsigned maxEval, double reqAbsError, double reqRelError,
                   double *val, double *err);
```

This integrates a function $F(x)$, returning a vector of FDIM integrands, where x is a DIM-dimensional vector ranging from XMIN to XMAX (i.e. in a hypercube $XMIN_i \leq x_i \leq XMAX_i$).

MAXEVAL specifies a maximum number of function evaluations (0 for no limit). [Note: the actual number of evaluations may somewhat exceed MAXEVAL: MAXEVAL is rounded up to an integer number of subregion evaluations.] Otherwise, the integration stops when the estimated |error| is less than REQABSError (the absolute error requested) **or** when the estimated |error| is less than REQRELError \times |integral value| (the relative error requested). (Either of the error tolerances can be set to **zero to ignore** it.)

VAL and ERR are arrays of length FDIM, which upon return are the computed integral values and estimated errors, respectively. (The estimated errors are based on an embedded cubature rule of lower order; for smooth functions, this estimate is usually conservative.)

The return value of adapt_integrate is 0 on success and nonzero if there was an error (currently, only out-of-memory situations).

The integrand function F should be a function of the form:

```
void f(unsigned ndim, const double *x, void *fdata,
       unsigned fdim, double *fval);
```

Here, the input is an array X of length NDIM (the point to be evaluated), the output is an array FVAL of length FDIM (the vector of function values at the point X).

The FDATA argument of F is equal to the FDATA argument passed to adapt_integrate—this can be used by the caller to pass any additional information through to F as needed (rather than using global variables, which are not re-entrant). If F does not need any additional data, you can just pass FDATA = NULL and ignore the FDATA argument to F.

"Vectorized" interface

These integration algorithms actually evaluate the integrand in "batches" of several points at a time. It is often useful to have access to this information so that your integrand function is not called for one point at a time, but rather for a whole "vector" of many points at once. For example, you may want to evaluate the integrand in parallel at different

points. This functionality is available by calling:

```
int adapt_integrate_v(unsigned fdim, integrand_v f, void *fdata,
                    unsigned dim, const double *xmin, const double *xmax,
                    unsigned maxEval, double reqAbsError, double reqRelError,
                    double *val, double *err);
```

All of the arguments and the return value are identical to `adapt_integrate`, above, except that now the integrand `F` is of type `integrand_v`, corresponding to a function of a different form. The integrand function `F` should now be a function of the form:

```
void f(unsigned ndim, unsigned npts, const double *x, void *fdata,
      unsigned fdim, double *fval);
```

Now, `X` is not a single point, but an array of `NPTS` points (length `NPTS×NDIM`), and upon return the values of all `FDIM` integrands at all `NPTS` points should be stored in `FVAL` (length `FDIM×NPTS`). In particular, `x[i*ndim + j]` is the `j`-th coordinate of the `i`-th point (`i < npts` and `j < ndim`), and the `k`-th function evaluation (`k < fdim`) for the `i`-th point is returned in `fval[k*npt + i]`.

The size of `NPTS` will vary with the dimensionality of the problem; higher-dimensional problems will have (exponentially) larger `NPTS`, allowing for the possibility of more parallelism. Currently, `NPTS` starts at 15 in 1d, 17 in 2d, and 33 in 3d, but as `adapt_integrate_v` calls your integrand more and more times the value of `NPTS` will grow. e.g. if you end up requiring several thousand points in total, `NPTS` may grow to several hundred. We utilize an algorithm from:

- I. Gladwell, "Vectorization of one dimensional quadrature codes," pp. 230–238 in *Numerical Integration. Recent Developments, Software and Applications*, G. Fairweather and P. M. Keast, eds., NATO ASI Series C203, Dordrecht (1987).

as described in the article "Parallel globally adaptive algorithms for multi-dimensional integration" by Bull and Freeman (<http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.42.6638>) (1994).

Example

As a simple example, consider the Gaussian integral of the scalar function $f(\vec{x}) = \exp(-\sigma|\vec{x}|^2)$ over the hypercube $[-2,2]^3$ in 3 dimensions. You could compute this integral via code that looks like:

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

void f(unsigned ndim, const double *x, void *fdata, unsigned fdim, double *fval) {
    double sigma = *((double *) fdata); // we can pass σ via fdata argument
    double sum = 0;
    unsigned i;
    for (i = 0; i < ndim; ++i) sum += x[i] * x[i];
    // compute the output value: note that fdim should == 1 from below
    fval[0] = exp(-sigma * sum);
}
```

then, later in the program where we call `adapt_integrate`:

```
{
    double xmin[3] = {-2,-2,-2}, xmax[3] = {2,2,2}, sigma = 0.5, val, err;
    adapt_integrate(1, f, &sigma, 3, xmin, xmax, 0, 0, 1e-4, &val, &err);
    printf("Computed integral = %0.10g +/- %g\n", val, err);
}
```

Here, we have specified a relative error tolerance of 10^{-4} (and no absolute error tolerance or maximum number of function evaluations). Note also that, to demonstrate the `fdata` parameter, we have used it to pass the σ value through

to our function (rather than hard-coding the value of σ in `f` or using a global variable).

The output should be:

```
Computed integral = 13.69609043 +/- 0.00136919
```

Note that the estimated *relative* error is $0.00136919/13.69609043 = 9.9969 \times 10^{-5}$, within our requested tolerance of 10^{-4} . The *actual* error in the integral value, as can be determined e.g. by running the integration with a much lower tolerance, is much smaller: the integral is too small by about 0.00002, for an actual relative error of about 1.4×10^{-6} . As mentioned above, for smooth integrands the estimated error is almost always conservative (which means, unfortunately, that the integrator usually does more function evaluations than it needs to).

With the vectorized interface `adapt_integrate_v`, one would instead do:

```
void f(unsigned ndim, unsigned npts, const double *x, void *fdata, unsigned fdim, double *fval) {
    double sigma = *((double *) fdata);
    unsigned i, j;
    for (j = 0; j < npts; ++j) { // evaluate the integrand for npts points
        double sum = 0;
        for (i = 0; i < ndim; ++i) sum += x[j*ndim+i] * x[j*ndim+i];
        fval[j] = exp(-sigma * sum);
    }
}
```

Infinite intervals

Integrals over infinite or semi-infinite intervals is possible by a change of variables. This is best illustrated in one dimension.

To compute an integral over a semi-infinite interval, you can perform the change of variables $x=a+t/(1-t)$:

$$\int_a^\infty f(x)dx = \int_0^1 f\left(a + \frac{t}{1-t}\right) \frac{1}{(1-t)^2} dt.$$

For an infinite interval, you can perform the change of variables $x=t/(1-t^2)$:

$$\int_{-\infty}^\infty f(x)dx = \int_{-1}^1 f\left(\frac{t}{1-t^2}\right) \frac{1+t^2}{(1-t^2)^2} dt.$$

Note the Jacobian factors multiplying $f(\cdot \cdot \cdot)$ in both integrals, and also that the limits of the t integrals are different in the two cases.

In multiple dimensions, one simply performs this change of variables on each dimension separately, as desired, multiplying the integrand by the corresponding Jacobian factor for each dimension being transformed.

The Jacobian factors diverge as the endpoints are approached. However, if $f(x)$ goes to zero at least as fast as $1/x^2$, then the limit of the integrand (including the Jacobian factor) is finite at the endpoints. If your $f(x)$ vanishes more slowly than $1/x^2$ but still faster than $1/x$, then the integrand blows up at the endpoints but the integral is still finite (it is an integrable singularity), so the code will work (although it may take many function evaluations to converge). If your $f(x)$ vanishes only as $1/x$, then it is not absolutely convergent and much more care is required even to define what you are trying to compute. (In any case, the quadrature/cubature rules currently employed in `cubature.c` do not evaluate the integrand at the endpoints, so you need not implement special handling for $|t|=1$.)

Test program

To compile a test program, just compile `cubature.c` while #defining `TEST_INTEGRATOR`, e.g. (on Unix or

GNU/Linux) via:

```
cc -DTEST_INTEGRATOR -o cubature_test cubature.c -lm
```

The usage is then:

```
./cubature_test <dim> <tol> <integrand> <maxeval>
```

where <dim> = # dimensions, <tol> = relative tolerance, <integrand> is 0–7 for one of eight possible test integrands (see below) and <maxeval> is the maximum number of function evaluations (0 for none, the default).

The different test integrands are:

- 0: a product of cosine functions
- 1: a Gaussian integral of $\exp(-x^2)$, remapped to $[0, \text{infinity})$ limits
- 2: volume of a hypersphere (integrating a discontinuous function!)
- 3: a simple polynomial (product of coordinates)
- 4: a Gaussian centered in the middle of the integration volume
- 5: a sum of two Gaussians
- 6: an example function by Tsuda, a product of terms with near poles
- 7: a test integrand by Morokoff and Caflisch, a simple product of dim-th roots of the coordinates (weakly singular at the boundary)

For example:

```
./cubature_test 3 1e-5 4
```

integrates the Gaussian function (4) to a desired relative error tolerance of 10^{-5} in 3 dimensions. The output is:

```
3-dim integral, tolerance = 1e-05
integrand 4: integral = 1, est err = 9.99952e-06, true err = 2.54397e-08
#evals = 82203
```

Notice that it finds the integral after 82203 function evaluations with an estimated error of about 10^{-5} , but the true error (compared to the exact result) is much smaller (2.5×10^{-8}): the error estimation is typically conservative when applied to smooth functions like this.

Retrieved from "http://ab-initio.mit.edu/wiki/index.php/Cubature_%28Multi-dimensional_integration%29"

- This page was last modified 00:52, 14 May 2011.
- This page has been accessed 16,002 times.
- Privacy policy
- About AbInitio
- Disclaimers