Cuba.jl Documentation

Release 0.4.0

Mose' Giordano

Jul 08, 2017

Contents

CHAPTER 1

Introduction

`Cuba.jl` is a Julia library for multidimensional numerical integration of real-valued functions of real arguments, using different algorithms.

This is just a Julia wrapper around the C Cuba library, version 4.2, by **Thomas Hahn**. All the credits goes to him for the underlying functions, blame me for any problem with the Julia interface.

All algorithms provided by Cuba library are supported in `Cuba.jl`:

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

For more details on the algorithms see the manual included in Cuba library and available in `deps/cuba-julia/cuba.pdf` after successful installation of `Cuba.jl`.

Integration is always performed on the n-dimensional unit hypercube $[0,1]^n$.

Tip: If you want to compute an integral over a different set, you have to scale the integrand function in order to have an equivalent integral on $[0,1]^n$ using substitution rules. For example, recall that in one dimension

$$\int_a^b f(x)\,\mathrm{d}x = \int_0^1 f(a+(b-a)y)(b-a)\,\mathrm{d}y$$

where the final $(b-a)$ is the one-dimensional version of the Jacobian.

Integration over a semi-infinite or an inifite domain is a bit trickier, but you can follow this advice from Steven G. Johnson: to compute an integral over a semi-infinite interval, you can perform the change of variables $x = a + y/(1-y)$:

$$\int_a^\infty f(x)\,\mathrm{d}x = \int_0^1 f\left(a + \frac{y}{1-y}\right)\frac{1}{(1-y)^2}\,\mathrm{d}y$$

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

$$\int_{-\infty}^\infty f(x)\,\mathrm{d}x = \int_0^1 f\left(\frac{2y-1}{(1-y)y}\right)\frac{2y^2-2y+1}{(1-y)^2y^2}\,\mathrm{d}y$$

In addition, recall that for an even function $\int_{-\infty}^\infty f(x)\,\mathrm{d}x = 2\int_0^\infty f(x)\,\mathrm{d}x$, while the integral of an odd function over the infinite interval $(-\infty,\infty)$ is zero.

All this generalizes straightforwardly to more than one dimension. In *Examples* section you can find the computation of a 3-dimensional *integral with non-constant boundaries* using `Cuba.jl` and two *integrals over infinite domains*.

`Cuba.jl` is available for GNU/Linux, Mac OS, and Windows (`i686` and `x86_64` architectures).

CHAPTER 2

Installation

`Cuba.jl` is available for Julia 0.5 and later versions, and can be installed with Julia built-in package manager. In a Julia session run the commands

```
julia> Pkg.update()
julia> Pkg.add("Cuba")
```

Installation script on GNU/Linux and Mac OS systems will download Cuba Library source code and build the Cuba shared object. In order to accomplish this task a C compiler is needed. Instead, on Windows a prebuilt version of the library is downloaded.

Older versions are also available for Julia 0.4.

CHAPTER 3

Usage

After installing the package, run

```
using Cuba
```

or put this command into your Julia script.

`Cuba.jl` provides the following functions to integrate:

`vegas` (*integrand, ndim=1, ncomp=1[; keywords...]*)

`suave` (*integrand, ndim=1, ncomp=1[; keywords...]*)

`divonne` (*integrand, ndim=1, ncomp=1[; keywords...]*)

`cuhre` (*integrand, ndim=1, ncomp=1[; keywords...]*)

Large parts of the following sections are borrowed from Cuba manual. Refer to it for more information on the details.

`Cuba.jl` wraps the 64-bit integers functions of Cuba library, in order to push the range of certain counters to its full extent. In detail, the following arguments:

- for Vegas: `nvec`, `minevals`, `maxevals`, `nstart`, `nincrease`, `nbatch`, `neval`,
- for Suave: `nvec`, `minevals`, `maxevals`, `nnew`, `nmin`, `neval`,
- for Divonne: `nvec`, `minevals`, `maxevals`, `ngiven`, `nextra`, `neval`,
- for Cuhre: `nvec`, `minevals`, `maxevals`, `neval`,

are passed to the Cuba library as 64-bit integers, so they are limited to be at most

```
julia> typemax(Int64)
9223372036854775807
```

There is no way to overcome this limit. See the following sections for the meaning of each argument.

Arguments

The only mandatory argument of integrator functions is:

- `integrand` (type: `Function`): the function to be integrated

Optional positional arguments are:

- `ndim` (type: `Integer`): the number of dimensions of the integratation domain. Defaults to 1 if omitted
- `ncomp` (type: `Integer`): the number of components of the integrand. Default to 1 if omitted

`ndim` and `ncomp` arguments must appear in this order, so you cannot omit `ndim` but not `ncomp`. `integrand` should be a function `integrand(x, f)` taking two arguments:

- the input vector `x` of length `ndim`
- the output vector `f` of length `ncomp`, used to set the value of each component of the integrand at point `x`

`x` and `f` are matrices with dimensions `(ndim, nvec)` and `(ncomp, nvec)`, respectively, when `nvec` > 1. See the *Vectorization* section below for more information.

Also anonymous functions are allowed as `integrand`. For those familiar with `Cubature.jl` package, this is the same syntax used for integrating vector-valued functions.

For example, the integral

$$\int_0^1 \cos(x)\,\mathrm{d}x = \sin(1) = 0.8414709848078965$$

can be computed with one of the following commands

```
julia> vegas((x, f) -> f[1] = cos(x[1]))
Component:
 1: 0.8414910005259612 ± 5.2708169787342156e-5 (prob.: 0.028607201258072673)
Integrand evaluations: 13500
Fail:                  0
Number of subregions:  0

julia> suave((x, f) -> f[1] = cos(x[1]))
Component:
 1: 0.84115236906584 ± 8.357995609919512e-5 (prob.: 1.0)
Integrand evaluations: 22000
Fail:                  0
Number of subregions:  22

julia> divonne((x, f) -> f[1] = cos(x[1]))
Component:
 1: 0.841468071955942 ± 5.3955070531551656e-5 (prob.: 0.0)
Integrand evaluations: 1686
Fail:                  0
Number of subregions:  14

julia> cuhre((x, f) -> f[1] = cos(x[1]))
Component:
 1: 0.8414709848078966 ± 2.2204460420128823e-16 (prob.: 3.443539937576958e-5)
Integrand evaluations: 195
Fail:                  0
Number of subregions:  2
```

In section *Examples* you can find more complete examples. Note that `x` and `f` are both arrays with type `Float64`, so `Cuba.jl` can be used to integrate real-valued functions of real arguments. See how to work with a *complex integrand*.

Note: if you used `Cuba.jl` until version 0.0.4, be aware that the user interface has been reworked in version 0.0.5 in a backward incompatible way.

Optional Keywords

All other arguments required by Cuba integrator routines can be passed as optional keywords. `Cuba.jl` uses some reasonable default values in order to enable users to invoke integrator functions with a minimal set of arguments. Anyway, if you want to make sure future changes to some default values of keywords will not affect your current script, explicitely specify the value of the keywords.

Common Keywords

These are optional keywords common to all functions:

- `nvec` (type: `Integer`, default: `1`): 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. Single Instruction Multiple Data (SIMD) vectorization, a larger value can be chosen. See *Vectorization* section.
- `reltol` (type: `Real`, default: `1e-4`), and `abstol` (type: `Real`, default: `1e-12`): the requested relative (ε_{rel}) and absolute (ε_{abs}) accuracies. The integrator tries to find an estimate $\hat{I}$ for the integral I which for every component c fulfills $|\hat{I}_c - I_c| \leq \max(\varepsilon_{\text{abs}}, \varepsilon_{\text{rel}}|I_c|)$.
- `flags` (type: `Integer`, default: `0`): 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.
  - (Vegas only)

    Bit 5 = `0`, take the integrator's state from the state file, if one is present.

    Bit 5 = `1`, reset the integrator's state even if a state file is present, i.e. keep only the grid. Together with Bit 4 this allows a grid adapted by one integration to be used for another integrand.
  - Bits 8–31 =: `level` determines the random-number generator.

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

- `seed` (type: `Integer`, default: `0`): the seed for the pseudo-random-number generator. This keyword is not available for *`cuhre()`*. 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`.

- `minevals` (type: `Real`, default: `0`): the minimum number of integrand evaluations required
- `maxevals` (type: `Real`, default: `1000000`): the (approximate) maximum number of integrand evaluations allowed
- `statefile` (type: `AbstractString`, default: `""`): a filename for storing the internal state. To not store the internal state, put `""` (empty string, this is the default) or `C_NULL` (C 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.

- `spin` (type: `Ptr{Void}`, default: `C_NULL`): this is the placeholder for the "spinning cores" pointer. `Cuba.jl` does not support parallelization, so this keyword should not have a value different from `C_NULL`.

Vegas-Specific Keywords

These optional keywords can be passed only to *`vegas()`*:

- `nstart` (type: `Integer`, default: `1000`): the number of integrand evaluations per iteration to start with
- `nincrease` (type: `Integer`, default: `500`): the increase in the number of integrand evaluations per iteration
- `nbatch` (type: `Integer`, default: `1000`): 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

- `gridno` (type: `Integer`, default: `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. 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.

Suave-Specific Keywords

These optional keywords can be passed only to *`suave()`*:

- `nnew` (type: `Integer`, default: `1000`): the number of new integrand evaluations in each subdivision
- `nmin` (type: `Integer`, default: `2`): the minimum number of samples a former pass must contribute to a subregion to be considered in that region's compound integral value. Increasing `nmin` may reduce jumps in the χ^2 value
- `flatness` (type: `Real`, default: `.25`): 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.

Divonne-Specific Keywords

These optional keywords can be passed only to *`divonne()`*:

- `key1` (type: `Integer`, default: `47`): 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).
  - `key2` (type: `Integer`, default: `1`): 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,

    * $n_2 \geq 40$, 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` (type: `Integer`, default: `1`): 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` (type: `Integer`, default: `5`): 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.

- `border` (type: `Real`, default: `0.`): 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 function cannot produce values directly on the integration boundary

- `maxchisq` (type: `Real`, default: `10.`): the 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` (type: `Real`, default: `0.25`): 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.

- `ngiven` (type: `Integer`, default: `0`): the number of points in the `xgiven` array

- `ldxgiven` (type: `Integer`, default: `0`): the leading dimension of `xgiven`, i.e. the offset between one point and the next in memory

- `xgiven` (type: `AbstractArray{Real}`, default: `zeros(Cdouble, ldxgiven, ngiven)`): 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.

- `nextra` (type: `Integer`, default: `0`): 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

- `peakfinder` (type: `Ptr{Void}`, default: `C_NULL`): the peak-finder subroutine

Cuhre-Specific Keyword

This optional keyword can be passed only to *`cuhre()`*:

- `key` (type: `Integer`, default: `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.

Output

The integrating functions *vegas()*, *suave()*, *divonne()*, and *cuhre()* (and the corresponding 64-bit integers functions) return an `Integral` object whose fields are

```
integral :: Vector{Float64}
error    :: Vector{Float64}
probl    :: Vector{Float64}
neval    :: Int64
fail     :: Int32
nregions :: Int32
```

The first three fields are arrays with length `ncomp`, the last three ones are scalars. The `Integral` object can also be iterated over like a tuple. In particular, if you assign the output of integrator functions to the variable named `result`, you can access the value of the `i`-th component of the integral with `result[1][i]` or `result.integral[i]` and the associated error with `result[2][i]` or `result.error[i]`.

- `integral` (type: `Vector{Float64}`, with `ncomp` components): the integral of `integrand` over the unit hypercube
- `error` (type: `Vector{Float64}`, with `ncomp` components): the presumed absolute error for each component of `integral`
- `probability` (type: `Vector{Float64}`, with `ncomp` components): 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` $> .95$
- `neval` (type: `Int64`): the actual number of integrand evaluations needed
- `fail` (type: `Int32`): 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 `maxevals` needs to be increased to reach the desired accuracy and returns this value.
- `nregions` (type: `Int32`): the actual number of subregions needed (always `0` in *vegas()*)

CHAPTER 4

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.

`Cuba.jl` cannot automatically vectorize the integrand function, of course, but it does pass (up to) `nvec` points per integrand call (*Common Keywords*). 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.

When `nvec` > 1, the input `x` is a matrix of dimensions `(ndim, nvec)`, while the output `f` is a matrix with dimensions `(ncomp, nvec)`. Vectorization can be used to evaluate more quickly the integrand function, for example by exploiting parallelism, thus speeding up computation of the integral. See the section *Vectorized Function* below for an example of a vectorized funcion.

Note: 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.

CHAPTER 5

Examples

One dimensional integral

The integrand of

$$\int_0^1 \frac{\log(x)}{\sqrt{x}}\,\mathrm{d}x$$

has an algebraic-logarithmic divergence for $x = 0$, but the integral is convergent and its value is -4. `Cuba.jl` integrator routines can handle this class of functions and you can easily compute the numerical approximation of this integral using one of the following commands:

```
julia> vegas( (x,f) -> f[1] = log(x[1])/sqrt(x[1]))
Component:
 1: -3.998162393712848 ± 0.0004406643716840933 (prob.: 0.28430529682022004)
Integrand evaluations: 1007500
Fail:                  1
Number of subregions:  0

julia> suave( (x,f) -> f[1] = log(x[1])/sqrt(x[1]))
Component:
 1: -3.9999762867171387 ± 0.00039504866661845624 (prob.: 1.0)
Integrand evaluations: 51000
Fail:                  0
Number of subregions:  51

julia> divonne( (x,f) -> f[1] = log(x[1])/sqrt(x[1]))
Component:
 1: -3.9997602130594374 ± 0.0003567874814901272 (prob.: 1.0)
Integrand evaluations: 11593
Fail:                  0
Number of subregions:  76

julia> cuhre( (x,f) -> f[1] = log(x[1])/sqrt(x[1]))
Component:
```

```
 1: -4.00000035506719 ± 0.0003395484028625721 (prob.: 0.0)
Integrand evaluations: 5915
Fail:                  0
Number of subregions:  46
```

Vector-valued integrand

Consider the integral

$$\int_\Omega \boldsymbol{f}(x, y, z)\,\mathrm{d}x\,\mathrm{d}y\,\mathrm{d}z$$

where $\Omega = [0, 1]^3$ and

$$\boldsymbol{f}(x, y, z) = \left(\sin(x)\cos(y)\exp(z),\ \exp(-(x^2 + y^2 + z^2)),\ \frac{1}{1 - xyz}\right)$$

In this case it is more convenient to write a simple Julia script to compute the above integral

```
using Cuba, SpecialFunctions

function integrand(x, f)
    f[1] = sin(x[1])*cos(x[2])*exp(x[3])
    f[2] = exp(-(x[1]^2 + x[2]^2 + x[3]^2))
    f[3] = 1/(1 - prod(x))
end

result, err = cuhre(integrand, 3, 3, abstol=1e-12, reltol=1e-10)
answer = [(e-1)*(1-cos(1))*sin(1), (sqrt(pi)*erf(1)/2)^3, zeta(3)]
for i = 1:3
    println("Component ", i)
    println(" Result of Cuba: ", result[i], " ± ", err[i])
    println(" Exact result:   ", answer[i])
    println(" Actual error:   ", abs(result[i] - answer[i]))
end
```

This is the output

```
Component 1
 Result of Cuba: 0.6646696797813739 ± 1.0050367631018485e-13
 Exact result:   0.6646696797813771
 Actual error:   3.219646771412954e-15
Component 2
 Result of Cuba: 0.4165383858806454 ± 2.932866749838454e-11
 Exact result:   0.41653838588663805
 Actual error:   5.9926508200192075e-12
Component 3
 Result of Cuba: 1.2020569031649702 ± 1.1958522385908214e-10
 Exact result:   1.2020569031595951
 Actual error:   5.375033751420233e-12
```

Integral with non-constant boundaries

The integral

$$\int_{-y}^{y}\int_{0}^{z}\int_{0}^{\pi}\cos(x)\sin(y)\exp(z)\,\mathrm{d}x\,\mathrm{d}y\,\mathrm{d}z$$

has non-constant boundaries. By applying the substitution rule repeatedly, you can scale the integrand function and get this equivalent integral over the fixed domain $\Omega = [0,1]^3$

$$\int_{\Omega} 2\pi^3 yz^2\cos(\pi yz(2x-1))\sin(\pi yz)\exp(\pi z)\,\mathrm{d}x\,\mathrm{d}y\,\mathrm{d}z$$

that can be computed with `Cuba.jl` using the following Julia script

```
using Cuba

function integrand(x, f)
    f[1] = 2pi^3*x[2]*x[3]^2*cos(pi*x[2]*x[3]*(2*x[1] - 1.0))*
           sin(pi*x[2]*x[3])*exp(pi*x[3])
end

result, err = cuhre(integrand, 3, 1, abstol=1e-12, reltol=1e-10)
answer = pi*e^pi - (4e^pi - 4)/5
println("Result of Cuba: ", result[1], " ± ", err[1])
println("Exact result:   ", answer)
println("Actual error:   ", abs(result[1] - answer))
```

This is the output

```
Result of Cuba: 54.98607586826157 ± 5.460606521639899e-9
Exact result:   54.98607586789537
Actual error:   3.6619951515604043e-10
```

Integrals over Infinite Domains

`Cuba.jl` assumes always as integration domain the hypercube $[0,1]^n$, but we have seen that using integration by substitution we can calculate integrals over different domains as well. In the *Introduction* we also proposed two useful substitutions that can be employed to change an infinite or semi-infinite domain into a finite one.

As a first example, consider the following integral with a semi-infinite domain:

$$\int_0^{\infty}\frac{\log(1+x^2)}{1+x^2}\,\mathrm{d}x$$

whose exact result is $\pi\log 2$. This can be computed with the following Julia script:

```
using Cuba

# The function we want to integrate over [0, ∞).
func(x) = log(1 + x^2)/(1 + x^2)

# Scale the function in order to integrate over [0, 1].
function integrand(x, f)
    f[1] = func(x[1]/(1 - x[1]))/(1 - x[1])^2
```

```
end

result, err = cuhre(integrand, abstol = 1e-12, reltol = 1e-10)
answer = pi*log(2)
println("Result of Cuba: ", result[1], " ± ", err[1])
println("Exact result:   ", answer)
println("Actual error:   ", abs(result[1] - answer))
```

This is the output:

```
Result of Cuba: 2.177586090305688 ± 2.1503995410096295e-10
Exact result:   2.177586090303602
Actual error:   2.085887018665744e-12
```

Now we want to calculate this integral, over an infinite domain

$$\int_{-\infty}^{\infty} \frac{1-\cos x}{x^2}\,\mathrm{d}x$$

which gives π. You can calculate the result with the code below. Note that integrand function has value $1/2$ for $x = 0$, but you have to inform Julia about this.

```
using Cuba

# The function we want to integrate over (-∞, ∞).
func(x) = x==0 ? 0.5*one(x) : (1 - cos(x))/x^2

# Scale the function in order to integrate over [0, 1].
function integrand(x, f)
    f[1] = func((2*x[1] - 1)/x[1]/(1 - x[1])) *
        (2*x[1]^2 - 2*x[1] + 1)/x[1]^2/(1 - x[1])^2
end

result, err = cuhre(integrand, abstol = 1e-7, reltol = 1e-7)
answer = float(pi)
println("Result of Cuba: ", result[1], " ± ", err[1])
println("Exact result:   ", answer)
println("Actual error:   ", abs(result[1] - answer))
```

The output of this script is

```
Result of Cuba: 3.1415928900555046 ± 2.050669142074607e-6
Exact result:   3.141592653589793
Actual error:   2.3646571145619077e-7
```

Complex integrand

As already explained, `Cuba.jl` operates on real quantities, so if you want to integrate a complex-valued function of complex arguments you have to treat complex quantities as 2-component arrays of real numbers. For example, if you do not remember Euler's formula, you can compute this simple integral

$$\int_0^{\pi/2} \exp(\mathrm{i}x)\,\mathrm{d}x$$

with the following Julia script

```
using Cuba

function integrand(x, f)
    # Complex integrand, scaled to integrate in [0, 1].
    tmp = cis(x[1]*pi/2)*pi/2
    # Assign to two components of "f" the real
    # and imaginary part of the integrand.
    f[1], f[2] = reim(tmp)
end

result = cuhre(integrand, 1, 2)
println("Result of Cuba: ", complex(result[1]...))
println("Exact result:   ", complex(1.0, 1.0))
```

This is the output

```
Result of Cuba: 1.0 + 1.0im
Exact result:   1.0 + 1.0im
```

Passing data to the integrand function

Cuba Library allows program written in C and Fortran to pass extra data to the integrand function with `userdata` argument. This is useful, for example, when the integrand function depends on changing parameters. In `Cuba.jl` the `userdata` argument is not available, but you do not normally need it.

For example, the cumulative distribution function $F(x;k)$ of chi-squared distribution is defined by

$$F(x;k) = \int_0^x \frac{t^{k/2-1}\exp(-t/2)}{2^{k/2}\Gamma(k/2)}\,\mathrm{d}t$$

The cumulative distribution function depends on parameter k, but the function passed as integrand to `Cuba.jl` integrator routines accepts as arguments only the input and output vectors. However you can easily define a function to calculate a numerical approximation of $F(x;k)$ based on the above integral expression because the integrand can access any variable visible in its scope. The following Julia script computes $F(x=\pi;k)$ for different k and compares the result with more precise values, based on the analytic expression of the cumulative distribution function, provided by GSL.jl package.

```
using Cuba, GSL

function chi2cdf(x::Real, k::Real)
    k2 = k/2
    # Chi-squared probability density function, without constant denominator.
    # The result of integration will be divided by that factor.
    function chi2pdf(t::Float64)
        # "k2" is taken from the outside.
        return t^(k2 - 1.0)*exp(-t/2)
    end
    # Neither "x" is passed directly to the integrand function,
    # but is visible to it.  "x" is used to scale the function
    # in order to actually integrate in [0, 1].
    x*cuhre((t,f) -> f[1] = chi2pdf(t[1]*x))[1][1]/(2^k2*gamma(k2))
end

x = pi
@printf("Result of Cuba: %.6f %.6f %.6f %.6f %.6f\n",
```

```
        map((k) -> chi2cdf(x, k), collect(1:5))...)
@printf("Exact result:   %.6f %.6f %.6f %.6f %.6f\n",
        map((k) -> cdf_chisq_P(x, k), collect(1:5))...)
```

This is the output

```
Result of Cuba: 0.923681 0.792120 0.629694 0.465584 0.321833
Exact result:   0.923681 0.792120 0.629695 0.465584 0.321833
```

Vectorized Function

Consider the integral

$$\int_\Omega \prod_{i=1}^{10} \cos(x_i)\,\mathrm{d}\boldsymbol{x} = \sin(1)^{10} = 0.1779883\ldots$$

where $\Omega = [0,1]^{10}$ and $\boldsymbol{x} = (x_1, \ldots, x_{10})$ is a 10-dimensional vector. A simple way to compute this integral is the following:

```
julia> using Cuba, BenchmarkTools

julia> cuhre((x, f) -> f[] = prod(cos.(x)), 10)
Component:
 1: 0.1779870665870775 ± 1.0707995959536173e-6 (prob.: 0.2438374075714901)
Integrand evaluations: 7815
Fail:                  0
Number of subregions:  2

julia> @benchmark cuhre((x, f) -> f[] = prod(cos.(x)), 10)
BenchmarkTools.Trial:
  memory estimate:  2.62 MiB
  allocs estimate:  39082
  --------------
  minimum time:     1.633 ms (0.00% GC)
  median time:      1.692 ms (0.00% GC)
  mean time:        1.867 ms (8.62% GC)
  maximum time:     3.660 ms (45.54% GC)
  --------------
  samples:          2674
  evals/sample:     1
```

We can use vectorization in order to speed up evaluation of the integrand function.

```
julia> function fun_vec(x,f)
           f[1,:] .= 1.0
           for j in 1:size(x,2)
               for i in 1:size(x, 1)
                   f[1, j] *= cos(x[i, j])
               end
           end
       end
fun_vec (generic function with 1 method)

julia> cuhre(fun_vec, 10, nvec = 1000)
```

```
Component:
 1: 0.1779870665870775 ± 1.0707995959536173e-6 (prob.: 0.2438374075714901)
Integrand evaluations: 7815
Fail:                  0
Number of subregions:  2

julia> @benchmark cuhre(fun_vec2, 10, nvec = 1000)
BenchmarkTools.Trial:
  memory estimate:  2.88 KiB
  allocs estimate:  54
  --------------
  minimum time:     949.976 μs (0.00% GC)
  median time:      954.039 μs (0.00% GC)
  mean time:        966.930 μs (0.00% GC)
  maximum time:     1.204 ms (0.00% GC)
  --------------
  samples:          5160
  evals/sample:     1
```

A further speed up can be gained by running the `for` loop in parallel with `Threads.@threads`. For example, running Julia with 4 threads:

```
julia> function fun_par(x,f)
           f[1,:] .= 1.0
           Threads.@threads for j in 1:size(x,2)
               for i in 1:size(x, 1)
                   f[1, j] *= cos(x[i, j])
               end
           end
       end
fun_par (generic function with 1 method)

julia> cuhre(fun_par, 10, nvec = 1000)
Component:
 1: 0.1779870665870775 ± 1.0707995959536173e-6 (prob.: 0.2438374075714901)
Integrand evaluations: 7815
Fail:                  0
Number of subregions:  2

julia> @benchmark cuhre(fun_par, 10, nvec = 1000)
BenchmarkTools.Trial:
  memory estimate:  3.30 KiB
  allocs estimate:  63
  --------------
  minimum time:     507.914 μs (0.00% GC)
  median time:      515.182 μs (0.00% GC)
  mean time:        520.667 μs (0.06% GC)
  maximum time:     3.801 ms (85.06% GC)
  --------------
  samples:          9565
  evals/sample:     1
```

CHAPTER 6

Performance

`Cuba.jl` cannot (yet?) take advantage of parallelization capabilities of Cuba Library. Nonetheless, it has performances comparable with equivalent native C or Fortran codes based on Cuba library when `CUBACORES` environment variable is set to `0` (i.e., multithreading is disabled). The following is the result of running the benchmark present in `test` directory on a 64-bit GNU/Linux system running Julia 0.7.0-DEV.363 (commit 6071f1a02e) equipped with an Intel(R) Core(TM) i7-4700MQ CPU. The C and FORTRAN 77 benchmark codes have been compiled with GCC 6.3.0.

```
$ CUBACORES=0 julia -e 'cd(Pkg.dir("Cuba")); include("test/benchmark.jl")'
INFO: Performance of Cuba.jl:
  0.271304 seconds (Vegas)
  0.579783 seconds (Suave)
  0.329504 seconds (Divonne)
  0.238852 seconds (Cuhre)
INFO: Performance of Cuba Library in C:
  0.319799 seconds (Vegas)
  0.619774 seconds (Suave)
  0.340317 seconds (Divonne)
  0.266906 seconds (Cuhre)
INFO: Performance of Cuba Library in Fortran:
  0.272000 seconds (Vegas)
  0.584000 seconds (Suave)
  0.308000 seconds (Divonne)
  0.232000 seconds (Cuhre)
```

Of course, native C and Fortran codes making use of Cuba Library outperform `Cuba.jl` when higher values of `CUBACORES` are used, for example:

```
$ CUBACORES=1 julia -e 'cd(Pkg.dir("Cuba")); include("test/benchmark.jl")'
INFO: Performance of Cuba.jl:
  0.279524 seconds (Vegas)
  0.581078 seconds (Suave)
  0.327319 seconds (Divonne)
  0.241211 seconds (Cuhre)
INFO: Performance of Cuba Library in C:
```

```
  0.115113 seconds (Vegas)
  0.596503 seconds (Suave)
  0.152511 seconds (Divonne)
  0.085805 seconds (Cuhre)
INFO: Performance of Cuba Library in Fortran:
  0.108000 seconds (Vegas)
  0.604000 seconds (Suave)
  0.160000 seconds (Divonne)
  0.092000 seconds (Cuhre)
```

`Cuba.jl` internally fixes `CUBACORES` to 0 in order to prevent from forking `julia` processes that would only slow down calculations eating up the memory, without actually taking advantage of concurrency. Furthemore, without this measure, adding more Julia processes with `addprocs()` would only make the program segfault.

CHAPTER 7

Related projects

Another Julia package for multidimenensional numerical integration is available: Cubature.jl, by Steven G. Johnson.

CHAPTER 8

Development

`Cuba.jl` is developed on GitHub: https://github.com/giordano/Cuba.jl. Feel free to report bugs and make suggestions at https://github.com/giordano/Cuba.jl/issues.

History

The ChangeLog of the package is available in NEWS.md file in top directory. There have been some breaking changes from time to time, beware of them when upgrading the package.

CHAPTER 9

License

The Cuba.jl package is licensed under the GNU Lesser General Public License, the same as Cuba library. The original author is Mosè Giordano.

CHAPTER 10

Credits

If you use this library for your work, please credit Thomas Hahn. Citable papers about Cuba Library:

- Hahn, T. 2005, Computer Physics Communications, 168, 78. DOI:10.1016/j.cpc.2005.01.010. arXiv:hep-ph/0404043. Bibcode:2005CoPhC.168...78H.
- Hahn, T. 2015, Journal of Physics Conference Series, 608, 012066. DOI:10.1088/1742-6596/608/1/012066. arXiv:1408.6373. Bibcode:2015JPhCS.608a2066H.

Index