GAIL—Guaranteed Automatic Integration Library in MATLAB: Documentation for Version 2.3.1*

Sou-Cheng T. Choi[1], Yuhan Ding[2], Fred J. Hickernell[3], Lan Jiang[4], Lluís Antoni Jiménez Rugama[5], Da Li[6], Jagadeeswaran Rathinavel[7], Xin Tong[8], Kan Zhang[9], Yizhi Zhang[10], and Xuan Zhou[11]

[1–11]Illinois Institute of Technology
[1] Kamakura Corporation
[4]Compass Inc.
[5]UBS
[7]Wi-Tronix, LLC
[8]University of Illinois, Chicago
[10]Jamran International Inc.
[11]J.P. Morgan

Jun 21, 2020

*Email: gail-users@googlegroups.com. Our work was supported in part by the National Science Foundation under grants NSF-DMS-1522687 and NSF-DMS-1115392; and by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under contract DE-AC02-06CH11357.

Contents

Open Source License

1 Introduction

Automatic and adaptive approximation, optimization, or integration of functions in a cone with guarantee of accuracy is a relatively new paradigm [12]. Our purpose is to create an open-source MATLAB package, Guaranteed Automatic Integration Library (GAIL) [9], following the philosophy of reproducible research championed by Claerbout [11] and Donoho [2], and sustainable practices of robust scientific software development [21, 22, 20]. For our conviction that true scholarship in computational sciences are characterized by reliable reproducibility [5, 7, 4], we employ the best practices in mathematical research and software engineering known to us and available in MATLAB.

The rest of this document describes the key features of functions in GAIL, which includes one-dimensional function approximation [6, 13, 12] and minimization [6, 28] using linear splines, one-dimensional numerical integration using trapezoidal rule [12], and last but not least, mean estimation and multidimensional integration by Monte Carlo methods [19, 17] or Quasi Monte Carlo methods [26, 25, 24, 23, 18].

If you find GAIL helpful in your work, please support us by citing the software [8], related papers and materials following the practices recommended by citing research software [27].

Note that some of our GAIL algorithms are being ported to an open-source Python package for Quasi-Monte Carlo methods (QMCPy) [10].

1.1 Downloads

GAIL can be downloaded from `http://gailgithub.github.io/GAIL_Dev/`.

Alternatively, you can get a local copy of the GAIL repository with this command:

```
git clone https://github.com/GailGithub/GAIL_Dev.git
```

1.2 Requirements

You will need to install MATLAB 7 or a later version.

GAIL is developed in MATLAB versions R2016a to R2020a. In particular, three of our core algorithms, cubSobol_g, cubBayesNet_g, and cubBayesLattice_g require the following MATLAB add-on toolboxes: Signal Processing Toolbox, Optimization Toolbox, Statistics and Machine Learning Toolbox. In MATLAB, we could use the following command to find out toolbox dependencies of an algorithm:

```
names = dependencies.toolboxDependencyAnalysis('cubBayesNet_g')
```

For development and testing purposes, we use the third-party toolboxes, Chebfun version 5.7.0 and Doctest for MATLAB, version 2010.

1.3 Documentation

Detailed documentation is available at `GAIL_Matlab/Documentation/html/GAIL.html`.

You can also go to MATLAB's Help. Under the section of Supplemental Software, you will find GAIL Toolbox's searchable HTML documentation.

A PDF version of GAIL's documentation with selected examples is available at `https://github.com/GailGithub/GAIL_Dev/blob/master/Documentation/gail_ug_2_3_1.pdf`

1.4 General Usage Notes

GAIL version 2.3.1 [8] includes the following algorithms:

1. **funappx_g** [6, 13, 12]: One-dimensional function approximation on bounded interval

2. **funmin_g** [6, 28]: global minimum value of univariate function on a closed interval
3. **integral_g** [29, 12]: One-dimensional integration on bounded interval
4. **meanMC_g** [19, 17]: Monte Carlo method for estimating mean of a random variable
5. **cubMC_g** [19, 17]: Monte Carlo method for numerical multiple integration
6. **cubLattice_g** [26]: Quasi-Monte Carlo method using rank-1 Lattices cubature for d-dimensional integration
7. **cubSobol_g** [26, 18, 23]: Quasi-Monte Carlo method using Sobol' cubature for d-dimensional integration
8. **cubBayesLattice_g** [25]: Bayesian cubature method for d-dimensional integration using lattice points
9. **cubBayesNet_g** [25, 24]: Bayesian cubature method for d-dimensional integration using Sobol points
10. **meanMC_CLT**: Monte Carlo method with Central Limit Theorem (CLT) confidence intervals for estimating mean of a random variable

Each one of our key GAIL algorithms, with the exception of **cubBayesLattice_g** and **cubBayesNet_g**, can parse inputs with the following three patterns of APIs, where `f` is a real-valued MATLAB function or function handle; `in_param` and `out_param` are MATLAB structure arrays; and `x` is an estimated output:

1. Ordered input values:

```
[x, out_param] = algo(f, inputVal1, inputVal2, inputVal3,...)
```

2. Input structure array:

```
[x, out_param] = algo(f, in_param)
```

3. Ordered input values, followed by optional name-value pairs:

```
[x, out_param] = algo(f, 'input2', inputVal2, 'input3', inputVal3,...)
```

For object classes **cubBayesLattice_g** and **cubBayesNet_g**, the output pattern is `[out, x]`, where `out` is an instance of the corresponding object class.

1.5 Installation Instruction

1. Unzip the contents of the zip file to a directory and maintain the existing directory and subdirectory structure. (Please note: If you install into the `toolbox` subdirectory of the MATLAB program hierarchy, you will need to click the button "Update toolbox path cache" from the File/Preferences... dialog in MATLAB.)
2. In MATLAB, add the GAIL directory to your path. This can be done by running `GAIL_Install.m`. Alternatively, this can be done by selecting "File/Set Path..." from the main or Command window menus, or with the command `pathtool`. We recommend that you select the "Save" button on this dialog so that GAIL is on the path automatically in future MATLAB sessions.
3. To check if GAIL is installed successfully, type `help funappx_g` to see if its documentation shows up.

Alternatively, you could do this:

1. Download `DownloadInstallGail_2_3_1.m` and put it where you want GAIL to be installed.
2. Execute it in MATLAB.

To uninstall GAIL, execute `GAIL_Uninstall`.

To reinstall GAIL, execute `GAIL_Install`.

1.6 Tests

We provide quick doctests for each of the functions above. To run doctests in **funappx_g**, for example, issue the command `doctest funappx_g`.

We also provide unit tests for MATLAB version 8 or later. To run unit tests for **funmin_g**, for instance, execute `run(ut_funmin_g)`.

We execute automated nightly fast tests and weekly long tests on our server. Moreover, these tests are now conducted for all MATLAB versions from R2016a to R2020a. The test reports are available on Mega cloud storage at https://mega.nz/. More specifically, fast and long test reports are archived in text files, `gail_daily_tests*.out` and `gail_weekly_tests*.out` at https://mega.nz/folder/FlMEjI5a#jVixXyAoI05ppbCstz8yEg respectively. Output files such as images of test scripts are archived at https://mega.nz/folder/I0cAEKJD#AyQ_8tmxkknfIsuEW0_jnA

1.7 Contact Information

Please send any queries, questions, or comments to

`gail-users@googlegroups.com`

1.8 Website

For more information about GAIL, visit GAIL project website.

1.9 Known Issues

During our documentation development with MATLAB releases 2019a and 2020a, the software's internal HTML viewer is found to display LaTeX expression in larger font size than it is intended to be. This is an aesthetic issue with no impact on the content accuracy. Users may use a web browser to view our HTML documentation instead. The main page to GAIL's HTML documentation is `GAIL.html`, located in the subfolder, `Documentation/html/`.

2 Release Notes

2.1 Major changes in algorithms

In this release, we have a new algorithm called **cubBayesNet_g**. Similar to **cubBayesLattice_g**, it is an automatic Bayesian cubature that considers the integrand a realization of a Gaussian process. **CubBayesNet_g** uses Sobol points whereas **cubBayesLattice_g** uses lattice points.

2.2 Major changes in publications

In the folder `Papers`, we have added a few recently published research articles and theses related to our core algorithms.

First, we have Rathinavel's 2019 PhD thesis [24] and his joint publication with Hickernell [25] that develop the theory behind **cubBayesLattice_g** and **cubBayesNet_g**.

In addition, we have included Ding, Hickernell, and Jiménez Rugama's recent paper, *An Adaptive Algorithm Employing Continuous Linear Functionals* [14].

2.3 Bug Fixes

In the previous versions of GAIL, three of our multiple integration algorithms, **cubMC_g**, **cubLattice_g**, and **cubSobol_g** when applied to an integral on a non-unit hypercube with respect to uniform measure, the numerical approximations omitted the proper normalization constant, i.e., dividing by the volume of the hypercube. This problem has been resolved in this release. We illustrate the bug fix with the simple example below. Consider $f(x, y) = \exp(-x^2 - y^2)$ with $(x, y) \in \mathcal{D} := [-1, 2]^2$. Integrating f with respect to the Lebesgue measure, we obtain

$$\mathcal{I} = \int_{\mathcal{D}} f(x)\, dy\, dx = \frac{\pi}{4}\big(\operatorname{erf}(1) + \operatorname{erf}(2)\big)^2 \approx 2.65333.$$

Integrating f with respect to the uniform measure m, we have instead,

$$\int_{\mathcal{D}} f(x)\, m(dx) = \frac{1}{m(\mathcal{D})}\mathcal{I} = \frac{1}{9}\mathcal{I} \approx 0.29481,$$

since the integration domain has volume $m(\mathcal{D}) = 9$; see, for example, [3]. Our GAIL functions by default integrate with respect to the uniform measure, but previous versions returned answers with respect to the Lebesgue measure.

3 funappx_g

1-D guaranteed locally adaptive function approximation (or function recovery) on $[a, b]$

3.1 Syntax

fappx = **funappx_g**(f)

fappx = **funappx_g**(f,a,b,abstol)

fappx = **funappx_g**(f,'a',a,'b',b,'abstol',abstol)

fappx = **funappx_g**(f,in_param)

[fappx, out_param] = **funappx_g**(f,...)

3.2 Description

fappx = **funappx_g**(f) approximates function f on the default interval [0,1] by an approximated function handle fappx within the guaranteed absolute error tolerance of 1e-6. When Matlab version is higher or equal to 8.3, fappx is an interpolant generated by griddedInterpolant. When Matlab version is lower than 8.3, fappx is a function handle generated by ppval and interp1. Input f is a function handle. The statement y = f(x) should accept a vector argument x and return a vector y of function values that is of the same size as x.

fappx = **funappx_g**(f,a,b,abstol) for a given function f and the ordered input parameters that define the finite interval [a,b], and a guaranteed absolute error tolerance abstol.

fappx = **funappx_g**(f,'a',a,'b',b,'abstol',abstol) approximates function f on the finite interval [a,b], given a guaranteed absolute error tolerance abstol. All four field-value pairs are optional and can be supplied in different order.

fappx = **funappx_g**(f,in_param) approximates function f on the finite interval [in_param.a,in_param.b], given a guaranteed absolute error tolerance in_param.abstol. If a field is not specified, the default value is used.

[fappx, out_param] = **funappx_g**(f,...) returns an approximated function fappx and an output structure out_param.

Properties

- fappx can be used for linear extrapolation outside [a,b].

Input Arguments

- f — input function
- in_param.a — left end point of interval, default value is 0.
- in_param.b — right end point of interval, default value is 1.
- in_param.abstol — guaranteed absolute error tolerance, default value is 1e-6.

Optional Input Arguments

- in_param.ninit — initial number of subintervals. Default to 20.
- in_param.nmax — when number of points hits the value, iteration will stop, default value is 1e7.
- in_param.maxiter — max number of iterations, default value is 1000.

Output Arguments

- fappx — approximated function handle (Note: When Matlab version is higher or equal to 8.3, fappx is an interpolant generated by griddedInterpolant. When Matlab version is lower than 8.3, fappx is a function handle generated by ppval and interp1.)
- out_param.f — input function.
- out_param.a — left end point of interval.
- out_param.b — right end point of interval.
- out_param.abstol — guaranteed absolute error tolerance.
- out_param.maxiter — max number of iterations.
- out_param.ninit — initial number of subintervals.
- out_param.exitflag — this is a vector with two elements, for tracking important warnings in the algorithm. The algorithm is considered successful (with out_param.exitflag == [0 0]) if no other flags arise warning that the results are not guaranteed. The initial value is [0 0] and the final value of this parameter is encoded as follows:

  [1 0]: If reaching overbudget. It states whether the max budget is attained without reaching the guaranteed error tolerance.

  [0 1]: If reaching overiteration. It states whether the max iterations is attained without reaching the guaranteed error tolerance.
- out_param.iter — number of iterations.
- out_param.npoints — number of points we need to reach the guaranteed absolute error tolerance.
- out_param.errest — an estimation of the absolute error for the approximation.

3.3 Guarantee

Please check the details of the guarantee in [6].

3.4 Examples

Example 1 Approximate function x^2 on $[-2, 2]$ with error tolerance 10^{-7}, default cost budget and initial number of subintervals 18.

```
f = @(x) x.^2; [~, out_param] = funappx_g(f,-2,2,1e-7,18)
```

```
out_param =
           a: -2
      abstol: 1.0000e-07
           b: 2
           f: @(x)x.^2
     maxiter: 1000
       ninit: 18
        nmax: 10000000
    exitflag: [0 0]
        iter: 12
     npoints: 36865
      errest: 2.9448e-08
```

Example 2 Approximate function x^2 on $[-2, 2]$ with default error tolerance, default cost budget and initial number of subintervals 17.

```
f = @(x) x.^2;
[~, out_param] = funappx_g(f,'a',-2,'b',2,'ninit',17)


out_param =
           a: -2
      abstol: 1.0000e-06
           b: 2
           f: @(x)x.^2
     maxiter: 1000
       ninit: 17
        nmax: 10000000
    exitflag: [0 0]
        iter: 10
     npoints: 8705
      errest: 5.2896e-07
```

Example 3 Approximate function x^2 on $[-5, 5]$ with error tolerance 10^{-6}, default cost budget and initial number of subintervals 18.

```
clear in_param; in_param.a = -5; in_param.b = 5; f = @(x) x.^2;
in_param.abstol = 10^(-6); in_param.ninit=18;
[~, out_param] = funappx_g(f,in_param)


out_param =
           a: -5
      abstol: 1.0000e-06
           b: 5
           f: @(x)x.^2
     maxiter: 1000
       ninit: 18
        nmax: 10000000
    exitflag: [0 0]
        iter: 11
     npoints: 18433
      errest: 7.3654e-07
```

3.5 See Also

interp1, griddedInterpolant, integral_g, funmin_g, meanMC_g, cubMC_g

4 funmin_g

1-D guaranteed locally adaptive function optimization on $[a, b]$

4.1 Syntax

fmin = **funmin_g**(f)

fmin = **funmin_g**(f,a,b,abstol)

fmin = **funmin_g**(f,'a',a,'b',b,'abstol',abstol)

fmin = **funmin_g**(f,in_param)

[fmin, out_param] = **funmin_g**(f,...)

4.2 Description

fmin = **funmin_g**(f) finds minimum value of function f on the default interval [0,1] within the guaranteed absolute error tolerance of 1e-6. Input f is a function handle.

fmin = **funmin_g**(f,a,b,abstol) finds minimum value of function f with ordered input parameters that define the finite interval [a,b], and a guaranteed absolute error tolerance abstol.

fmin = **funmin_g**(f,'a',a,'b',b,'abstol',abstol) finds minimum value of function f on the interval [a,b] with a guaranteed absolute error tolerance. All three field-value pairs are optional and can be supplied in different order.

fmin = **funmin_g**(f,in_param) finds minimum value of function f on the interval [in_param.a,in_param.b] with a guaranteed absolute error tolerance in_param.abstol. If a field is not specified, the default value is used.

[fmin, out_param] = **funmin_g**(f,...) returns minimum value fmin of function f and an output structure out_param.

Input Arguments

- f — input function.
- in_param.a — left end point of interval, default value is 0.
- in_param.b — right end point of interval, default value is 1.
- in_param.abstol — guaranteed absolute error tolerance, default value is 1e-6.

Optional Input Arguments

- in_param.ninit — initial number of subintervals. Default to 20.
- in_param.nmax — cost budget, default value is 1e7.
- in_param.maxiter — max number of iterations, default value is 1000.

Output Arguments

- out_param.f — input function
- out_param.a — left end point of interval

- out_param.b — right end point of interval
- out_param.abstol — guaranteed absolute error tolerance
- out_param.nmax — cost budget
- out_param.ninit — initial number of points we use
- out_param.npoints — number of points needed to reach the guaranteed absolute error tolerance
- out_param.exit — this is a vector with two elements, for tracking important warnings in the algorithm. The algorithm is considered successful (with out_param.exit == [0 0]) if no flags arise warning that the results are not guaranteed. The initial value is [0 0] and the final value of this parameter is encoded as follows:

  [1 0]: If reaching overbudget. It states whether the max budget is attained without reaching the guaranteed error tolerance.

  [0 1]: If reaching overiteration. It states whether the max iterations is attained without reaching the guaranteed error tolerance.
- out_param.errest — estimation of the absolute error bound
- out_param.iter — number of iterations
- out_param.intervals — the intervals containing point(s) where the minimum occurs. Each column indicates one interval where the first raw is the left point and the second row is the right point

4.3 Guarantee

Please check the details of the guarantee in [6].

4.4 Examples

Example 1 Minimize function $\exp(0.01(x-0.5)^2)$ with default input parameters.

```
f = @(x) exp(0.01*(x-0.5).^2); [fmin,out_param] = funmin_g(f)
```

```
fmin =
     1
out_param =
            f: @(x)exp(0.01*(x-0.5).^2)
            a: 0
            b: 1
       abstol: 1.0000e-06
        ninit: 20
         nmax: 10000000
      maxiter: 1000
     exitflag: [0 0]
         iter: 5
      npoints: 69
       errest: 2.5955e-07
    intervals: [2x1 double]
```

Example 2 Minimize function $\exp(0.01(x-0.5)^2)$ on $[-2,2]$ with error tolerance 10^{-7}, cost budget 1000000, initial number of points 10.

```
f = @(x) exp(0.01*(x-0.5).^2);
[fmin,out_param] = funmin_g(f,-2,2,1e-7,10,1000000)
```

```
fmin =
     1
out_param =
            f: @(x)exp(0.01*(x-0.5).^2)
            a: -2
            b: 2
       abstol: 1.0000e-07
        ninit: 10
         nmax: 1000000
      maxiter: 1000
     exitflag: [0 0]
         iter: 9
      npoints: 79
       errest: 6.1251e-08
    intervals: [2x1 double]
```

Example 3 Minimize function $\exp(0.01(x-0.5)^2)$ on $[-13, 8]$ with error tolerance 10^{-7}, cost budget 1000000, initial number of points 100.

```
clear in_param; in_param.a = -13; in_param.b = 8;
in_param.abstol = 1e-7;
in_param.ninit = 100;
in_param.nmax = 10^6;
[fmin,out_param] = funmin_g(f,in_param)
```

```
fmin =
   1.0000e+00
out_param =
            f: @(x)exp(0.01*(x-0.5).^2)
            a: -13
            b: 8
       abstol: 1.0000e-07
        ninit: 100
         nmax: 1000000
      maxiter: 1000
     exitflag: [0 0]
         iter: 8
      npoints: 203
       errest: 6.7816e-08
    intervals: [2x1 double]
```

Example 4 Minimize function $\exp(0.01(x-0.5)^2)$ on $[-2, 2]$ with error tolerance 10^{-5}, cost budget 1000000, initial number of points 64.

```
f=@(x) exp(0.01*(x-0.5).^2);
[fmin,out_param] = funmin_g(f,'a',-2,'b',2,'ninit',64,'nmax',1e6,'abstol',1e-5)
```

```
fmin =
     1
out_param =
            f: @(x)exp(0.01*(x-0.5).^2)
            a: -2
            b: 2
       abstol: 1.0000e-05
```

```
       ninit: 64
        nmax: 1000000
     maxiter: 1000
    exitflag: [0 0]
        iter: 3
     npoints: 107
      errest: 8.0997e-06
   intervals: [2x1 double]
```

4.5 See Also

fminbnd, funappx_g, integral_g

5 integral_g

1-D guaranteed function integration using Simpson's rule

5.1 Syntax

q = **integral_g**(f)

q = **integral_g**(f,a,b,abstol)

q = **integral_g**(f,'a',a,'b',b,'abstol',abstol)

q = **integral_g**(f,in_param)

[q, out_param] = **integral_g**(f,...)

5.2 Description

q = **integral_g**(f) computes q, the definite integral of function f on the interval [a,b] by Simpson's rule with in a guaranteed absolute error of 1e-6. Default starting number of sample points taken is 100 and default cost budget is 1e7. Input f is a function handle. The function y = f(x) should accept a vector argument x and return a vector result y, the integrand evaluated at each element of x.

q = **integral_g**(f,a,b,abstol)computes q, the definite integral of function f on the finite interval [a,b] by Simpson's rule with the ordered input parameters, and guaranteed absolute error tolerance abstol.

q = **integral_g**(f,'a',a,'b',b,'abstol',abstol) computes q, the definite integral of function f on the finite interval [a,b] by Simpson's rule within a guaranteed absolute error tolerance abstol. All four field-value pairs are optional and can be supplied.

q = **integral_g**(f,in_param) computes q, the definite integral of function f by Simpson's rule within a guaranteed absolute error in_param.abstol. If a field is not specified, the default value is used.

[q, out_param] = **integral_g**(f,...) returns the approximated integration q and output structure out_param.

Input Arguments

- f — input function
- in_param.a — left end of the integral, default value is 0
- in_param.b — right end of the integral, default value is 1
- in_param.abstol — guaranteed absolute error tolerance, default value is 1e-6

Optional Input Arguments

- in_param.nlo — lowest initial number of function values used, default value is 10
- in_param.nhi — highest initial number of function values used, default value is 1000
- in_param.nmax — cost budget (maximum number of function values), default value is 1e7

Output Arguments

- q — approximated integral
- out_param.f — input function

- out_param.a — low end of the integral
- out_param.b — high end of the integral
- out_param.abstol — guaranteed absolute error tolerance
- out_param.nlo — lowest initial number of function values
- out_param.nhi — highest initial number of function values
- out_param.nmax — cost budget (maximum number of function values)
- out_param.ninit — initial number of points we use, computed by nlo and nhi
- out_param.hcut — cut off value of the largest width between points used to estimate the third derivative of the function. See [29] for details.
- out_param.exceedbudget — it is true if the algorithm tries to use more points than cost budget, false otherwise.
- out_param.exceedbudget — it is true if the algorithm tries to use more points than cost budget, false otherwise.
- out_param.conechange — it is true if the cone constant has been changed, false otherwise. See [29] for details. If true, you may wish to change the input in_param.ninit to a larger number.
- out_param.npoints — number of points we need to reach the guaranteed absolute error tolerance abstol.
- out_param.errest — approximation error defined as the differences between the true value and the approximated value of the integral.

5.3 Guarantee

Please check the details of the guarantee in [29].

5.4 Examples

Example 1 Integrate function x^2 with default input parameter to make the error less than 10^{-6}.

```
[q, out_param] = integral_g(@(x) x.^2)
```

```
q =
    0.3333
out_param =
               f: @(x)x.^2
               a: 0
               b: 1
          abstol: 1.0000e-06
             nlo: 10
             nhi: 1000
            nmax: 10000000
           ninit: 62
            hcut: 10.1667
    exceedbudget: 0
      conechange: 0
         npoints: 67
          errest: 1.0907e-18
         VarfpCI: [9.8449e-10 1.4901e-09]
```

Example 2 Integrate function $\exp(-x^2)$ on $[1, 2]$ with lowest initial number of function values 100 and highest initial number of function values 10000, absolute error tolerance 10^{-5} and cost budget 10000000.

```
f = @(x) exp(-x.^2);
[q, out_param] = integral_g(f,'a',1,'b',2,'nlo',100,'nhi',10000,'abstol',1e-5,'nmax',1e7)
```

```
q =
   0.1353
out_param =
               f: @(x)exp(-x.^2)
               a: 1
               b: 2
          abstol: 1.0000e-05
             nlo: 100
             nhi: 10000
            nmax: 10000000
           ninit: 602
            hcut: 100.1667
    exceedbudget: 0
      conechange: 0
         npoints: 607
          errest: 4.3845e-13
         VarfpCI: [2.8380 4.2574]
```

5.5 See Also

integral, quad, funappx_g, funmin_g, meanMC_g, cubMC_g, cubLattice_g, cubSobol_g, cubBayesLattice_g

6 meanMC_g

Monte Carlo method to estimate the mean of a random variable.

6.1 Syntax

tmu = **meanMC_g**(Yrand)

tmu = **meanMC_g**(Yrand,abstol,reltol,alpha,fudge,nSig,n1,tbudget,nbudget)

tmu = **meanMC_g**(Yrand,'abstol',abstol,'reltol',reltol,'alpha',alpha,
'fudge',fudge,'nSig',nSig,'n1',n1,'tbudget',tbudget,'nbudget',nbudget)

[tmu, out_param] = **meanMC_g**(Yrand,in_param)

6.2 Description

tmu = **meanMC_g**(Yrand) estimates the mean, mu, of a random variable Y to within a specified generalized error tolerance, tolfun := max(abstol,reltol*|mu|), i.e., `mu - tmu` <= tolfun with probability at least (1 - alpha), where abstol is the absolute error tolerance, and reltol is the relative error tolerance. Usually the reltol determines the accuracy of the estimation, however, if `mu` is rather small, then abstol determines the accuracy of the estimation. Input Yrand is a function handle that accepts a positive integer input n and returns an n x 1 vector of IID instances of the random variable Y.

tmu = **meanMC_g**(Yrand,abstol,reltol,alpha) estimates the mean of a random variable Y to within a specified generalized error tolerance tolfun with guaranteed confidence level 1-alpha using all ordered parsing inputs abstol, reltol, alpha, fudge, nSig, n1, tbudget, nbudget.

tmu = **meanMC_g**(Yrand,'abstol',abstol,'reltol',reltol,'alpha',alpha) estimates the mean of a random variable Y to within a specified generalized error tolerance tolfun with guaranteed confidence level 1-alpha. All the field-value pairs are optional and can be supplied in different order, if a field is not supplied, the default value is used.

[tmu, out_param] = **meanMC_g**(Yrand,in_param) estimates the mean of a random variable Y to within a specified generalized error tolerance tolfun with the given parameters in_param and produce the estimated mean tmu and output parameters out_param. If a field is not specified, the default value is used.

Input Arguments

- Yrand — he function for generating n IID instances of a random variable Y whose mean we want to estimate. Y is often defined as a function of some random variable X with a simple distribution. The input of Yrand should be the number of random variables n, the output of Yrand should be n function values. For example, if Y = X.ˆ2 where X is a standard uniform random variable, then one may define Yrand = @(n) rand(n,1).ˆ2.

- in_param.abstol — the absolute error tolerance, which should be positive, default value is 1e-2.

- in_param.reltol — the relative error tolerance, which should be between 0 and 1, default value is 1e-1.

- in_param.alpha — the uncertainty, which should be a small positive percentage, default value is 1%.

- in_param.fudge — standard deviation inflation factor, which should be larger than 1, default value is 1.2.

- in_param.nSig — initial sample size for estimating the sample variance, which should be a moderately large integer bigger than or equal to 30, the default value is 1e4.

- in_param.n1 — initial sample size for estimating the sample mean, which should be a moderately large positive integer at least 30, the default value is 1e4.
- in_param.tbudget — the time budget in seconds to do the two-stage estimation, which should be positive, the default value is 100 seconds.
- in_param.nbudget — the sample budget to do the two-stage estimation, which should be a large positive integer, the default value is 1e9.

Output Arguments

- tmu — the estimated mean of Y.
- out_param.tau — the iteration step.
- out_param.n — the sample size used in each iteration.
- out_param.nremain — the remaining sample budget to estimate mu. It was calculated by the sample left and time left.
- out_param.ntot — total sample used.
- out_param.hmu — estimated mean in each iteration.
- out_param.tol — the reliable upper bound on error for each iteration.
- out_param.var — the sample variance.
- out_param.exitflag — the state of program when exiting:

  0 Successs

  1 Not enough samples to estimate the mean
- out_param.kurtmax — the upper bound on modified kurtosis.
- out_param.time — the time elapsed in seconds.

6.3 Guarantee

This algorithm attempts to calculate the mean, `mu`, of a random variable to a prescribed error tolerance, `tolfun` = max(`abstol`, `reltol` |`mu`|), with guaranteed confidence level (`1 - alpha`). If the algorithm terminates without showing any warning messages and provides an answer `tmu`, then the follow inequality would be satisfied:

`Pr(| mu - tmu | <= tolfun) >= 1-alpha`.

The cost of the algorithm, `N_tot`, is also bounded above by `N_up`, which is defined in terms of `abstol`, `reltol`, `nSig`, `n1`, `fudge`, `kurtmax`, `beta`. And the following inequality holds:

`Pr(N_tot <= N_up) >= 1-beta`.

Please refer to our paper for detailed arguments and proofs.

6.4 Examples

Example 1 Calculate the mean of x^2 when x is uniformly distributed in $[0, 1]$, with the absolute error tolerance $= 10^{-3}$ and uncertainty 5%.

```
in_param.reltol = 0; in_param.abstol = 1e-3;
in_param.alpha = 0.05; Yrand = @(n) rand(n,1).^2;
tmu = meanMC_g(Yrand,in_param); exactsol  = 1/3;
check = double(abs(exactsol-tmu) < 1e-3)
```

```
check =
     1
```

Example 2 Calculate the mean of $\exp(x)$ when x is uniformly distributed in $[0, 1]$, with the absolute error tolerance 10^{-3}.

```
tmu = meanMC_g(@(n)exp(rand(n,1)),1e-3,0); exactsol = exp(1)-1;
check = double(abs(exactsol-tmu) < 1e-3)
```

```
check =
     1
```

Example 3 Calculate the mean of $\cos(x)$ when x is uniformly distributed in $[0, 1]$, with the relative error tolerance 10^{-2} and uncertainty 0.05.

```
tmu = meanMC_g(@(n)cos(rand(n,1)),'reltol',1e-3,'abstol',1e-4,'alpha',0.01);
exactsol = sin(1);
check = double(abs(exactsol-tmu) < max(1e-3,1e-2*abs(exactsol)))
```

```
check =
     1
```

6.5 See Also

funappx_g, integral_g, cubMC_g, cubSobol_g, cubLattice_g, cubBayesLattice_g

7 meanMC_CLT

Monte Carlo method to estimate the mean of a random variable

7.1 Syntax

sol = **MEANMC_CLT**(Y,absTol,relTol,alpha,nSig,inflate)

7.2 Description

sol = **MEANMC_CLT**(Y,absTol,relTol,alpha,nSig,inflate) estimates the mean, mu, of a random variable to within a specified error tolerance, i.e., | mu - tmu | <= max(absTol,relTol|mu|) with probability at least 1-alpha, where abstol is the absolute error tolerance. The default values are abstol=1e-2 and alpha=1%. Input Y is a function handle that accepts a positive integer input n and returns an n x 1 vector of IID instances of the random variable.

This is a heuristic algorithm based on a Central Limit Theorem approximation.

Input Arguments

- Y — the function or structure for generating n IID instances of a random variable Y whose mean we want to estimate. Y is often defined as a function of some random variable X with a simple distribution. The input of Yrand should be the number of random variables n, the output of Yrand should be n function values. For example, if Y = X.ˆ2 where X is a standard uniform random variable, then one may define Yrand = @(n) rand(n,1).ˆ2.
- absTol — the absolute error tolerance, which should be non-negative — default = 1e-2
- relTol — the relative error tolerance, which should be non-negative and no greater than 1 — default = 0
- alpha — the uncertainty, which should be a small positive percentage — default = 1%
- nSig — the number of samples used to compute the sample variance — default = 1000
- inflate — the standard deviation inflation factor — default = 1.2

Output Arguments

- Y — the random generator
- absTol — the absolute error tolerance
- relTol — the relative error tolerance
- alpha — the uncertainty
- mu — the estimated mean of Y.
- stddev — sample standard deviation of the random variable
- nSample — total sample used.
- time — the time elapsed in seconds.
- errBd — the error bound.

7.3 Examples

Example 1 Estimate the integral with integrand $f(x) = x_1 x_2$ in the interval $[0,1]^2$ with absolute tolerance 10^{-3} and relative tolerance 0:

```
[mu,out] = meanMC_CLT(@(n) prod(rand(n,2),2), 0.001);
exact = 1/4;
check = double(abs(exact - mu) < 2e-3)
```

```
check =
     1
```

Example 2 Estimate the integral $f(x) = \exp(-x^2)$ in the interval $[0,1]$ using x as a control variate and relative error 10^{-3}:

```
f = @(x)[exp(-x.^2), x];
YXn = @(n)f(rand(n,1));
s = struct('Y',YXn,'nY',1,'trueMuCV',1/2);
exact = erf(1)*sqrt(pi)/2;
success = 0; runs = 1000; tol = 1e-3;
for i=1:runs, success = success + double(abs(exact-meanMC_CLT(s,0,tol)) < tol*exact); end
check = success >= 0.99 * runs
```

```
check =
     1
```

Example 3 Estimate the Keister's integration in dimension 1 with $a = 1$, $\frac{1}{\sqrt{2}}$ and using $\cos(x)$ as a control variate:

```
normsqd = @(x) sum(x.*x,2);
f = @(normt,a,d) ((2*pi*a^2).^(d/2)) * cos(a*sqrt(normt)).* exp((1/2-a^2)*normt);
f1 = @(x,a,d) f(normsqd(x),a,d);
f2 = @(x)[f1(x,1,1),f1(x,1/sqrt(2),1),cos(x)];
YXn = @(n)f2(randn(n,1));
s = struct('Y',YXn,'nY',2,'trueMuCV',1/sqrt(exp(1)));
[hmu,out] = meanMC_CLT(s,0,1e-3);
exact = 1.380388447043143;
check = double(abs(exact-hmu) < max(0,1e-3*abs(exact)))
```

```
check =
     1
```

Example 4 Estimate the integral with integrand $f(x) = x_1^3 x_2^3 x_3^3$ in the interval $[0,1]^3$ with pure absolute error 10^{-3} using $x_1 x_2 x_3$ as a control variate:

```
f = @(x) [x(:,1).^3.*x(:,2).^3.*x(:,3).^3, x(:,1).*x(:,2).*x(:,3)];
s = struct('Y',@(n)f(rand(n,3)),'nY',1,'trueMuCV',1/8);
[hmu,out] = meanMC_CLT(s,1e-3,0);
exact = 1/64;
check = double(abs(exact-hmu) < max(1e-3,1e-3*abs(exact)))
```

```
check =
     1
```

Example 5 Estimate the integrals with integrands $f_1(x) = x_1^3 x_2^3 x_3^3$ and $f_2(x) = x_1^2 x_2^2 x_3^2 - \frac{1}{27} + \frac{1}{64}$ in the interval $[0,1]^3$ using $x_1 x_2 x_3$ and $x_1 + x_2 + x_3$ as control variates:

```
  f = @(x) [x(:,1).^3.*x(:,2).^3.*x(:,3).^3, ...
            x(:,1).^2.*x(:,2).^2.*x(:,3).^2-1/27+1/64, ...
            x(:,1).*x(:,2).*x(:,3), ...
            x(:,1)+x(:,2)+x(:,3)];
  s = struct('Y',@(n)f(rand(n,3)),'nY',2,'trueMuCV',[1/8 1.5]);
  [hmu,out] = meanMC_CLT(s,1e-4,1e-3);
  exact = 1/64;
  check = double(abs(exact-hmu) < max(1e-4,1e-3*abs(exact)))
```

```
check =
     1
```

7.4 See Also

funappx_g, integral_g, cubMC_g, meanMC_g, cubLattice_g, cubSobol_g, cubBayesLattice_g

8 cubMC_g

Monte Carlo method to evaluate a multidimensional integral

8.1 Syntax

[Q,out_param] = **cubMC_g**(f,hyperbox)

Q = **cubMC_g**(f,hyperbox,measure,abstol,reltol,alpha)

Q = **cubMC_g**(f,hyperbox,'measure',measure,'abstol',abstol,'reltol',reltol,'alpha',alpha)

[Q out_param] = **cubMC_g**(f,hyperbox,in_param)

8.2 Description

[Q,out_param] = **cubMC_g**(f,hyperbox) estimates the integral of f over hyperbox to within a specified generalized error tolerance, tolfun = max(abstol, reltol*| I |), i.e., | I - Q | <= tolfun with probability at least (1 - alpha), where abstol is the absolute error tolerance, and reltol is the relative error tolerance. Usually the reltol determines the accuracy of the estimation, however, if | I | is rather small, then abstol determines the accuracy of the estimation. Input f is a function handle that accepts an n x d matrix input, where d is the dimension of the hyperbox, and n is the number of points being evaluated simultaneously.

When measure is 'uniform', 'uniform box', 'normal' or 'Gaussian', the input hyperbox is a 2 x d matrix, where the first row corresponds to the lower limits and the second row corresponds to the upper limits. When measure is 'uniform ball' or 'uniform sphere', the input hyperbox is a vector with d+1 elements, where the first d values correspond to the center of the ball and the last value corresponds to the radius of the ball. For these last two measures, a user can optionally specify what transformation should be used in order to get a uniform distribution on a ball of sphere. When measure is 'uniform ball_box', the box-to-ball transformation, which gets a set of points uniformly distributed on a ball from a set of points uniformly distributed on a box, will be used. When measure is 'uniform ball_normal', the normal-to-ball transformation, which gets a set of points uniformly distributed on a ball from a set of points normally distributed on the space, will be used. Similarly, the measures 'uniform sphere_box' and 'uniform sphere_normal' can be defined. The default transformations are the box-to-ball and the box-to-sphere transformations, depending on the region of integration.

Q = **cubMC_g**(f,hyperbox,measure,abstol,reltol,alpha) estimates the integral of function f over hyperbox to within a specified generalized error tolerance tolfun with guaranteed confidence level 1-alpha using all ordered parsing inputs f, hyperbox, measure, abstol, reltol, alpha, fudge, nSig, n1, tbudget, nbudget, flag. The input f and hyperbox are required and others are optional.

Q = **cubMC_g**(f,hyperbox,'measure',measure,'abstol',abstol,'reltol',reltol,'alpha',alpha) estimates the integral of f over hyperbox to within a specified generalized error tolerance tolfun with guaranteed confidence level 1-alpha. All the field-value pairs are optional and can be supplied in different order. If an input is not specified, the default value is used.

[Q out_param] = **cubMC_g**(f,hyperbox,in_param) estimates the integral of f over hyperbox to within a specified generalized error tolerance tolfun with the given parameters in_param and produce output parameters out_param and the integral Q.

Input Arguments

- f — the integrand.
- hyperbox — the integration hyperbox. The default value is [zeros(1,d); ones(1,d)], the default d is 1.

- in_param.measure — the measure for generating the random variable, the default is 'uniform'. The other measures could be handled are 'uniform box', 'normal'/'Gaussian', 'uniform ball'/'uniform ball_box'/'uniform ball_normal' and 'uniform sphere'/'uniform sphere_box'/'uniform sphere_normal'. The input should be a string type, hence with quotes.

- in_param.abstol — the absolute error tolerance, the default value is 1e-2.

- in_param.reltol — the relative error tolerance, the default value is 1e-1.

- in_param.alpha — the uncertainty, the default value is 1%.

Optional Input Arguments

- in_param.fudge — the standard deviation inflation factor, the default value is 1.2.

- in_param.nSig — initial sample size for estimating the sample variance, which should be a moderately large integer at least 30, the default value is 1e4.

- in_param.n1 — initial sample size for estimating the sample mean, which should be a moderately large positive integer at least 30, the default value is 1e4.

- in_param.tbudget — the time budget to do the estimation, the default value is 100 seconds.

- in_param.nbudget — the sample budget to do the estimation, the default value is 1e9.

- in_param.flag — the value corresponds to parameter checking status:

  0 not checked

  1 checked by meanMC_g

  2 checked by cubMC_g

Output Arguments

- Q — the estimated value of the integral.

- out_param.n — the sample size used in each iteration.

- out_param.ntot — total sample used, including the sample used to convert time budget to sample budget and the sample in each iteration step.

- out_param.nremain — the remaining sample budget to estimate I. It was calculated by the sample left and time left.

- out_param.tau — the iteration step.

- out_param.hmu — estimated integral in each iteration.

- out_param.tol — the reliable upper bound on error for each iteration.

- out_param.kurtmax — the upper bound on modified kurtosis.

- out_param.time — the time elapsed in seconds.

- out_param.var — the sample variance.

8.3 Guarantee

This algorithm attempts to calculate the integral of function f over a hyperbox to a prescribed error tolerance `tolfun` = max(`abstol`, `reltol` |`I`|) with guaranteed confidence level `1-alpha`. If the algorithm terminates without showing any warning messages and provides an answer `Q`, then the following inequality would be satisfied:

`Pr(| Q - I | <= tolfun) >= 1-alpha`.

The cost of the algorithm, `N_tot`, is also bounded above by `N_up`, which is a function in terms of `abstol`, `reltol`, `nSig`, `n1`, `fudge`, `kurtmax`, `beta`. And the following inequality holds:

`Pr (N_tot <= N_up) >= 1-beta`.

Please refer to our paper for detailed arguments and proofs.

8.4 Examples

Example 1 Estimate the integral with integrand $f(x) = \sin(x)$ over the interval $[1, 2]$ with default parameters.

```
f = @(x) sin(x); interval = [1;2];
Q = cubMC_g(f,interval,'uniform',1e-3,1e-2);
exactsol = 0.9564;
check = double(abs(exactsol-Q) < max(1e-3,1e-2*abs(exactsol)))
```

```
check =
     1
```

Example 2 Estimate the integral with integrand $f(x) = \exp(-x_1^2 - x_2^2)$ over the hyperbox $[0, 0; 1, 1]$, where $x = [x_1, x_2]$ is a vector.

```
f = @(x) exp(-x(:,1).^2-x(:,2).^2); hyperbox = [0 0;1 1];
Q = cubMC_g(f,hyperbox,'uniform',1e-3,0);
exactsol = 0.5577;
check = double(abs(exactsol-Q) < 1e-3)
```

```
check =
     1
```

Example 3 Estimate the integral with integrand $f(x) = 2^d \prod(x_1 x_2 \cdots x_d) + 0.555$ over the hyperbox $[0, 1]^d$, where $x = [x_1, x_2, \ldots, x_d]$ is a vector.

```
d = 3; f = @(x) 2^d*prod(x,2)+0.555; hyperbox = [zeros(1,d); ones(1,d)];
in_param.abstol = 1e-3;in_param.reltol=1e-3;
Q = cubMC_g(f,hyperbox,in_param);
exactsol = 1.555;
check = double(abs(exactsol-Q) < max(1e-3,1e-3*abs(exactsol)))
```

```
check =
     1
```

Example 4 Estimate the integral with integrand $f(x) = \exp(-x_1^2 - x_2^2)$ in R^2, where $x = [x_1, x_2]$ is a vector.

```
f = @(x) exp(-x(:,1).^2-x(:,2).^2); hyperbox = [-inf -inf;inf inf];
Q = cubMC_g(f,hyperbox,'normal',0,1e-2);
exactsol = 1/3;
check = double(abs(exactsol-Q) < max(0,1e-2*abs(exactsol)))
```

```
check =
     1
```

Example 5 Estimate the integral with integrand $f(x) = x_1^2 + x_2^2$ in the disk with center $(0,0)$ and radius 1, where $x = [x_1, x_2]$ is a vector.

```
f = @(x) x(:,1).^2+x(:,2).^2; hyperbox = [0,0,1];
Q = cubMC_g(f,hyperbox,'uniform ball','abstol',1e-3,'reltol',1e-3);
exactsol = pi/2;
check = double(abs(exactsol-Q) < max(1e-3,1e-3*abs(exactsol)))
```

```
check =
     1
```

8.5 See Also

funappx_g, integral_g, meanMC_g, cubLattice_g, cubSobol_g, cubBayesLattice_g

9 cubLattice_g

Quasi-Monte Carlo method using rank-1 Lattices cubature over a d-dimensional region to integrate within a specified generalized error tolerance with guarantees under Fourier coefficients cone decay assumptions.

9.1 Syntax

[q,out_param] = **cubLattice_g**(f,hyperbox)

q = **cubLattice_g**(f,hyperbox,measure,abstol,reltol)

q = **cubLattice_g**(f,hyperbox,'measure',measure,'abstol',abstol,'reltol',reltol)

q = **cubLattice_g**(f,hyperbox,in_param)

9.2 Description

[q,out_param] = **cubLattice_g**(f,hyperbox) estimates the integral of f over the d-dimensional region described by hyperbox, and with an error guaranteed not to be greater than a specific generalized error tolerance, tolfun:=max(abstol,reltol*| integral(f) |). Input f is a function handle. f should accept an n x d matrix input, where d is the dimension and n is the number of points being evaluated simultaneously.

When measure is 'uniform', the input hyperbox is a 2 x d matrix, where the first row corresponds to the lower limits and the second row corresponds to the upper limits of the integral. When measure is 'uniform ball' or 'uniform sphere', the input hyperbox is a vector with d+1 elements, where the first d values correspond to the center of the ball and the last value corresponds to the radius of the ball. For these last two measures, a user can optionally specify what transformation should be used in order to get a uniform distribution on a ball. When measure is 'uniform ball_box', the box-to-ball transformation, which gets a set of points uniformly distributed on a ball from a set of points uniformly distributed on a box, will be used. When measure is 'uniform ball_normal', the normal-to-ball transformation, which gets a set of points uniformly distributed on a ball from a set of points normally distributed on the space, will be used. Similarly, the measures 'uniform sphere_box' and 'uniform sphere_normal' can be used to specify the desired transformations. The default transformations are the box-to-ball and the box-to-sphere transformations, depending on the region of integration. Given the construction of our Lattices, d must be a positive integer with 1 <= d <= 600.

q = **cubLattice_g**(f,hyperbox,measure,abstol,reltol) estimates the integral of f over the hyperbox. The answer is given within the generalized error tolerance tolfun. All parameters should be input in the order specified above. If an input is not specified, the default value is used. Note that if an input is not specified, the remaining tail cannot be specified either. Inputs f and hyperbox are required. The other optional inputs are in the correct order: measure,abstol,reltol,shift,mmin,mmax,fudge, and transform.

q = **cubLattice_g**(f,hyperbox,'measure',measure,'abstol',abstol,'reltol',reltol) estimates the integral of f over the hyperbox. The answer is given within the generalized error tolerance tolfun. All the field-value pairs are optional and can be supplied in any order. If an input is not specified, the default value is used.

q = **cubLattice_g**(f,hyperbox,in_param) estimates the integral of f over the hyperbox. The answer is given within the generalized error tolerance tolfun.

Input Arguments

- f — the integrand whose input should be a matrix n x d where n is the number of data points and d the dimension, which cannot be greater than 600. By default f is f=@ x.^2.

- hyperbox — the integration region defined by its bounds. When measure is 'uniform' or 'normal', hyperbox must be a 2 x d matrix, where the first row corresponds to the lower limits and the second row corresponds to the upper limits of the integral. When measure is 'uniform ball' or 'uniform sphere', the input hyperbox is a vector with d+1 elements, where the first d values correspond to the center of the ball and the last value corresponds to the radius of the ball. The default value is [0;1].

- in_param.measure — for f(x)*mu(dx), we can define mu(dx) to be the measure of a uniformly distributed random variable in the hyperbox or normally distributed with covariance matrix I_d. The possible values are 'uniform', 'normal', 'uniform ball', 'uniform ball_box', 'uniform ball_normal', 'uniform sphere', 'uniform sphere_box' and 'uniform sphere_normal'. For 'uniform', the hyperbox must be a finite volume, for 'normal', the hyperbox can only be defined as (-Inf,Inf)^d and, for 'uniform ball' or 'uniform sphere', hyperbox must have finite values for the coordinates of the center and a finite positive value for the radius. By default it is 'uniform'.

- in_param.abstol — the absolute error tolerance, abstol>=0. By default it is 1e-4. For pure absolute tolerance, set in_param.reltol = 0.

- in_param.reltol — the relative error tolerance, which should be in [0,1]. Default value is 1e-2. For pure absolute tolerance, set in_param.abstol = 0.

Optional Input Arguments

- in_param.shift — the Rank-1 lattices can be shifted to avoid the origin or other particular points. The shift is a vector in [0,1]^d. By default we consider a shift uniformly sampled from [0,1]^d.

- in_param.mmin — the minimum number of points to start is 2^mmin. The cone condition on the Fourier coefficients decay requires a minimum number of points to start. The advice is to consider at least mmin=10. mmin needs to be a positive integer with mmin<=mmax. By default it is 10.

- in_param.mmax — tthe maximum budget is 2^mmax. By construction of our Lattices generator, mmax is a positive integer such that mmin<=mmax. mmax should not be bigger than the gail.lattice_gen allows. The default value is 20.

- in_param.fudge — the positive function multiplying the finite sum of Fast Fourier coefficients specified in the cone of functions. This input is a function handle. The fudge should accept an array of nonnegative integers being evaluated simultaneously. For more technical information about this parameter, refer to the references. By default it is @(m) 5*2.^-m.

- in_param.transform — the algorithm is defined for continuous periodic functions. If the input function f is not, there are 5 types of transform to periodize it without modifying the result. By default it is the Baker's transform. The options are:

  id : no transformation.

  Baker : Baker's transform or tent map in each coordinate. Preserving only continuity but simple to compute. Chosen by default.

  C0 : polynomial transformation only preserving continuity.

  C1 : polynomial transformation preserving the first derivative.

  C1sin : Sidi's transform with sine, preserving the first derivative. This is in general a better option than 'C1'.

Output Arguments

- q — the estimated value of the integral.

- out_param.d — dimension over which the algorithm integrated.

- out_param.n — number of Rank-1 lattice points used for computing the integral of f.

- out_param.bound_err — predicted bound on the error based on the cone condition. If the function lies in the cone, the real error will be smaller than generalized tolerance.

- out_param.time — time elapsed in seconds when calling cubLattice_g.
- out_param.exitflag — this is a binary vector stating whether warning flags arise. These flags tell about which conditions make the final result certainly not guaranteed. One flag is considered arisen when its value is 1. The following list explains the flags in the respective vector order:

  1 : If reached overbudget, meaning the max budget is attained without reaching the guaranteed error tolerance.

  2 : If the function lies outside the cone, results are not guaranteed to be accurate. Note that this parameter is computed on the transformed function, not the input function. For more information on the transforms, check the input parameter in_param.transform; for information about the cone definition, check the article mentioned below.

9.3 Guarantee

This algorithm computes the integral of real valued functions in $[0, 1]^d$ with a prescribed generalized error tolerance. The Fourier coefficients of the integrand are assumed to be absolutely convergent. If the algorithm terminates without warning messages, the output is given with guarantees under the assumption that the integrand lies inside a cone of functions. The guarantee is based on the decay rate of the Fourier coefficients. For integration over domains other than $[0, 1]^d$, this cone condition applies to $f \circ \psi$ (the composition of the functions) where ψ is the transformation function for $[0, 1]^d$ to the desired region. For more details on how the cone is defined, please refer to the references [26].

9.4 Examples

Example 1 Estimate the integral with integrand $f(x) = x_1 x_2$ in the interval $[0, 1]^2$:

```
f = @(x) prod(x,2); hyperbox = [zeros(1,2);ones(1,2)];
q = cubLattice_g(f,hyperbox,'uniform',1e-5,0,'transform','C1sin');
exactsol = 1/4;
check = double(abs(exactsol-q) < 1e-5)
```

```
check =
     1
```

Example 2 Estimate the integral with integrand $f(x) = x_1^2 x_2^2 x_3^2$ in the interval R^3 where x_1, x_2 and x_3 are normally distributed:

```
f = @(x) x(:,1).^2.*x(:,2).^2.*x(:,3).^2; hyperbox = [-inf(1,3);inf(1,3)];
q = cubLattice_g(f,hyperbox,'normal',1e-3,1e-3,...
    'transform','C1sin','shift',2^(-25)*ones(1,3));
exactsol = 1;
check = double(abs(exactsol-q) < max(1e-3,1e-3*abs(exactsol)))
```

```
check =
     1
```

Example 3 Estimate the integral with integrand $f(x) = \exp(-x_1^2 - x_2^2)$ in the interval $[-1, 2]^2$:

```
f = @(x) exp(-x(:,1).^2-x(:,2).^2); hyperbox = [-ones(1,2);2*ones(1,2)];
q = cubLattice_g(f,hyperbox,'uniform',1e-3,1e-2,'transform','C1');
exactsol = 1/9*(sqrt(pi)/2*(erf(2)+erf(1)))^2;
check = double(abs(exactsol-q) < max(1e-3,1e-2*abs(exactsol)))
```

```
check =
     1
```

Example 4 Estimate the price of an European call with $S_0 = 100$, $K = 100$, $r = \sigma^2/2$, $\sigma = 0.05$, and $T = 1$.

```
f = @(x) exp(-0.05^2/2)*max(100*exp(0.05*x)-100,0);
hyperbox = [-inf(1,1);inf(1,1)];
q = cubLattice_g(f,hyperbox,'normal',1e-4,1e-2,'transform','C1sin');
price = normcdf(0.05)*100 - 0.5*100*exp(-0.05^2/2);
check = double(abs(price-q) < max(1e-4,1e-2*abs(price)))
```

```
check =
     1
```

Example 5 Estimate the integral with integrand $f(x) = 8x_1x_2x_3x_4x_5$ in the interval $[0,1]^5$ with pure absolute error 10^{-5}.

```
f = @(x) 8*prod(x,2); hyperbox = [zeros(1,5);ones(1,5)];
q = cubLattice_g(f,hyperbox,'uniform',1e-5,0); exactsol = 1/4;
check = double(abs(exactsol-q) < 1e-5)
```

```
check =
     1
```

Example 6 Estimate the integral with integrand $f(x) = 3/(5-4(\cos(2\pi x)))$ in the interval $[0,1]$ with pure absolute error 10^{-5}.

```
f = @(x) 3./(5-4*(cos(2*pi*x))); hyperbox = [0;1];
q = cubLattice_g(f,hyperbox,'uniform',1e-5,0,'transform','id');
exactsol = 1;
check = double(abs(exactsol-q) < 1e-5)
```

```
check =
     1
```

Example 7 Estimate the integral with integrand $f(x) = x_1^2 + x_2^2$ over the disk with center $(0,0)$ and radius 1 with pure absolute error 10^{-4}, where $x = [x_1x_2]$ is a vector.

```
f = @(x) x(:,1).^2+x(:,2).^2; hyperbox = [0,0,1];
q = cubLattice_g(f,hyperbox,'uniform ball','abstol',1e-4,'reltol',0);
exactsol = pi/2;
check = double(abs(exactsol-q) < 1e-4)
```

```
check =
     1
```

9.5 See Also

cubSobol_g, cubMC_g, cubBayesLattice_g, meanMC_g, meanMC_CLT, integral_g

10 cubSobol_g

Quasi-Monte Carlo method using Sobol' cubature over the d-dimensional region to integrate within a specified generalized error tolerance with guarantees under Walsh-Fourier coefficients cone decay assumptions

10.1 Syntax

[q,out_param] = **cubSobol_g**(f,hyperbox)

q = **cubSobol_g**(f,hyperbox,measure,abstol,reltol)

q = **cubSobol_g**(f,hyperbox,'measure',measure,'abstol',abstol,'reltol',reltol)

q = **cubSobol_g**(f,hyperbox,in_param)

10.2 Description

[q,out_param] = **cubSobol_g**(f,hyperbox) estimates the integral of f over the d-dimensional region described by hyperbox, and with an error guaranteed not to be greater than a specific generalized error tolerance, tolfun:=max(abstol,reltol*| integral(f) |). Input f is a function handle. f should accept an n x d matrix input, where d is the dimension and n is the number of points being evaluated simultaneously.

When measure is 'uniform', the input hyperbox is a 2 x d matrix, where the first row corresponds to the lower limits and the second row corresponds to the upper limits of the integral. When measure is 'uniform ball' or 'uniform sphere', the input hyperbox is a vector with d+1 elements, where the first d values correspond to the center of the ball and the last value corresponds to the radius of the ball. For these last two measures, a user can optionally specify what transformation should be used in order to get a uniform distribution on a ball. When measure is 'uniform ball_box', the box-to-ball transformation, which gets a set of points uniformly distributed on a ball from a set of points uniformly distributed on a box, will be used. When measure is 'uniform ball_normal', the normal-to-ball transformation, which gets a set of points uniformly distributed on a ball from a set of points normally distributed on the space, will be used. Similarly, the measures 'uniform sphere_box' and 'uniform sphere_normal' can be used to specify the desired transformations. The default transformations are the box-to-ball and the box-to-sphere transformations, depending on the region of integration. Given the construction of Sobol' sequences, d must be a positive integer with 1 <= d<= 1111.

q = **cubSobol_g**(f,hyperbox,measure,abstol,reltol) estimates the integral of f over the hyperbox. The answer is given within the generalized error tolerance tolfun. All parameters should be input in the order specified above. If an input is not specified, the default value is used. Note that if an input is not specified, the remaining tail cannot be specified either. Inputs f and hyperbox are required. The other optional inputs are in the correct order: measure,abstol,reltol,mmin,mmax,and fudge.

q = **cubSobol_g**(f,hyperbox,'measure',measure,'abstol',abstol,'reltol',reltol) estimates the integral of f over the hyperbox. The answer is given within the generalized error tolerance tolfun. All the field-value pairs are optional and can be supplied in any order. If an input is not specified, the default value is used.

q = **cubSobol_g**(f,hyperbox,in_param) estimates the integral of f over the hyperbox. The answer is given within the generalized error tolerance tolfun.

Input Arguments

- f — the integrand whose input should be a matrix n x d where n is the number of data points and d the dimension, which cannot be greater than 1111. By default $f(x) = x^2$.

  — if using control variates, f needs to be a structure with two fields: First field: 'func', need to be a function handle with n x (J+1) dimension outputs, where J is the number of control variates.

- First column is the output of target function, next J columns are the outputs of control variates.
- Second field: 'cv', need to be a 1 x J vector that stores the exact means of control variates in the same order from the function handle. For examples of how to use control variates, please check Example 7 below.

- hyperbox — the integration region defined by its bounds. When measure is 'uniform' or 'normal', hyperbox must be a 2 x d matrix, where the first row corresponds to the lower limits and the second row corresponds to the upper limits of the integral. When measure is 'uniform ball' or 'uniform sphere', the input hyperbox is a vector with d+1 elements, where the first d values correspond to the center of the ball and the last value corresponds to the radius of the ball. The default value is [0;1].

- in_param.measure — for f(x)*mu(dx), we can define mu(dx) to be the measure of a uniformly distributed random variable in the hyperbox or normally distributed with covariance matrix I_d. The possible values are 'uniform', 'normal', 'uniform ball', 'uniform ball_box', 'uniform ball_normal', 'uniform sphere', 'uniform sphere_box' and 'uniform sphere_normal'. For 'uniform', the hyperbox must be a finite volume, for 'normal', the hyperbox can only be defined as (-Inf,Inf)^d and, for 'uniform ball' or 'uniform sphere', hyperbox must have finite values for the coordinates of the center and a finite positive value for the radius. By default it is 'uniform'.

- in_param.abstol — the absolute error tolerance, abstol>=0. By default it is 1e-4. For pure absolute tolerance, set in_param.reltol = 0.

- in_param.reltol — the relative error tolerance, which should be in [0,1]. Default value is 1e-2. For pure absolute tolerance, set in_param.abstol = 0.

Optional Input Arguments

- in_param.mmin — the minimum number of points to start is 2^mmin. The cone condition on the Fourier coefficients decay requires a minimum number of points to start. The advice is to consider at least mmin=10. mmin needs to be a positive integer with mmin<=mmax. By default it is 10.

- in_param.mmax — the maximum budget is 2^mmax. By construction of the Sobol' generator, mmax is a positive integer such that mmin<=mmax<=53. The default value is 24.

- in_param.fudge — the positive function multiplying the finite sum of Fast Walsh Fourier coefficients specified in the cone of functions. This input is a function handle. The fudge should accept an array of nonnegative integers being evaluated simultaneously. For more technical information about this parameter, refer to the references. By default it is `@(m) 5*2.^-m`.

Output Arguments

- q — the estimated value of the integral.

- out_param.d — dimension over which the algorithm integrated.

- out_param.n — number of Sobol' points used for computing the integral of f.

- out_param.bound_err — predicted bound on the error based on the cone condition. If the function lies in the cone, the real error will be smaller than generalized tolerance.

- out_param.time — time elapsed in seconds when calling cubSobol_g.

- out_param.beta — the value of beta when using control variates as in f-(h-Ih)beta, if using 'betaUpdate' option, beta is a vector storing value of each iteration.

- y — fast transform coefficients of the input function.

- kappanumap — wavenumber mapping used in the error bound.

- out_param.exitflag — this is a binary vector stating whether warning flags arise. These flags tell about which conditions make the final result certainly not guaranteed. One flag is considered arisen when its value is 1. The following list explains the flags in the respective vector order:

  1 : If reaching overbudget. It states whether the max budget is attained without reaching the guaranteed error tolerance.

  2 : If the function lies outside the cone. In this case, results are not guaranteed. For more information about the cone definition, check the article mentioned below.

10.3 Guarantee

This algorithm computes the integral of real valued functions in $[0,1]^d$ with a prescribed generalized error tolerance. The Walsh-Fourier coefficients of the integrand are assumed to be absolutely convergent. If the algorithm terminates without warning messages, the output is given with guarantees under the assumption that the integrand lies inside a cone of functions. The guarantee is based on the decay rate of the Walsh-Fourier coefficients. For integration over domains other than $[0,1]^d$, this cone condition applies to $f \circ \psi$ (the composition of the functions) where ψ is the transformation function for $[0,1]^d$ to the desired region. For more details on how the cone is defined, please refer to the references below.

10.4 Examples

Example 1 Estimate the integral with integrand $f(x) = x_1 x_2$ in the hyperbox $[0,1]^2$:

```
f = @(x) prod(x,2); hyperbox = [zeros(1,2); ones(1,2)];
q = cubSobol_g(f,hyperbox,'uniform',1e-5,0); exactsol = 1/4;
check = double(abs(exactsol-q) < 1e-5)
```

```
check =
     1
```

Example 2 Estimate the integral with integrand $f(x) = x_1^2 x_2^2 x_3^2$ in the hyperbox R^3 where x_1, x_2 and x_3 are normally distributed:

```
f = @(x) x(:,1).^2.*x(:,2).^2.*x(:,3).^2; hyperbox = [-inf(1,3);inf(1,3)];
q = cubSobol_g(f,hyperbox,'normal',1e-3,1e-3); exactsol = 1;
check = double(abs(exactsol-q) < max(1e-3,1e-3*abs(exactsol)))
```

```
check =
     1
```

Example 3 Estimate the integral with integrand $f(x) = exp(-x_1^2 - x_2^2)$ in the hyperbox $[-1,2]^2$:

```
f = @(x) exp(-x(:,1).^2-x(:,2).^2); hyperbox = [-ones(1,2); 2*ones(1,2)];
q = cubSobol_g(f,hyperbox,'uniform',1e-3,1e-2);
exactsol = 1/9*(sqrt(pi)/2*(erf(2)+erf(1)))^2;
check = double(abs(exactsol-q) < max(1e-3,1e-2*abs(exactsol)))
```

```
check =
     1
```

Example 4 Estimate the price of an European call with $S_0 = 100$, $K = 100$, $r = \sigma^2/2$, $\sigma = 0.05$, and $T = 1$.

```
f = @(x) exp(-0.05^2/2)*max(100*exp(0.05*x)-100,0);
hyperbox = [-inf(1,1);inf(1,1)];
q = cubSobol_g(f,hyperbox,'normal',1e-4,1e-2);
price = normcdf(0.05)*100 - 0.5*100*exp(-0.05^2/2);
check = double(abs(price-q) < max(1e-4,1e-2*abs(price)))
```

```
check =
     1
```

Example 5 Estimate the integral with integrand $f(x) = 8x_1x_2x_3x_4x_5$ in the interval $[0,1)^5$ with pure absolute error 10^{-5}.

```
f = @(x) 8*prod(x,2); hyperbox = [zeros(1,5);ones(1,5)];
q = cubSobol_g(f,hyperbox,'uniform',1e-5,0); exactsol = 1/4;
check = double(abs(exactsol-q) < 1e-5)
```

```
check =
     1
```

Example 6 Estimate the integral with integrand $f(x) = x_1^2 + x_2^2$ over the disk with center $(0,0)$ and radius 1 with pure absolute error 10^{-5}, where $x = [x_1, x_2]$ is a vector.

```
f = @(x) x(:,1).^2+x(:,2).^2; hyperbox = [0,0,1];
q = cubSobol_g(f,hyperbox,'uniform ball','abstol',1e-4,'reltol',0);
exactsol = pi/2;
check = double(abs(exactsol-q) < 1e-4)
```

```
check =
     1
```

Example 7 Estimate the integral with integrand $f(x) = 10x_1 - 5x_2^2 + x_3^3$ in the interval $[0,2)^3$ with pure absolute error 10^{-5} using two control variates $h_1(x) = x_1$ and $h_2(x) = x_2^2$.

```
g.func = @(x) [10*x(:,1)-5*x(:,2).^2+1*x(:,3).^3, x(:,1), x(:,2).^2];
g.cv = [1,4/3]; hyperbox= [zeros(1,3);2*ones(1,3)];
q = cubSobol_g(g,hyperbox,'uniform',1e-6,0); exactsol = 16/3;
check = double(abs(exactsol-q) < 1e-6)
```

```
check =
     1
```

10.5 See Also

cubLattice_g, cubBayesLattice_g, cubMC_g, meanMC_g, meanMC_CLT, integral_g

11 cubBayesLattice_g

Bayesian cubature method to estimate the integral of a random variable using rank-1 Lattices over a d-dimensional region within a specified generalized error tolerance with guarantees under Bayesian assumptions.

11.1 Syntax

[OBJ,Q] = **cubBayesLattice_g**(f,dim,'absTol',absTol,'relTol',relTol,
'order',order,'ptransform',ptransform,'arbMean',arbMean)

OBJ = **cubBayesLattice_g**(f,dim,'absTol',absTol,'relTol',relTol,
'order',order,'ptransform',ptransform,'arbMean',arbMean)

[Q,OutP] = **compInteg**(OBJ)

[OBJ,Q] = **cubBayesLattice_g**(f,dim)

[OBJ,Q] = **cubBayesLattice_g**(f,dim,absTol,relTol)

[OBJ,Q] = **cubBayesLattice_g**(f,dim,inParams)

11.2 Description

[OBJ,Q] = **cubBayesLattice_g**(f,dim,'absTol',absTol,'relTol',relTol,'order',order,'ptransform',ptransform, 'arbMean',arbMean) initializes the object with the given parameters and also returns an estimate of integral Q.

[Q,OutP] = **compInteg**(OBJ) estimates the integral of f over hyperbox $[0,1]^{\mathrm{dim}}$ using rank-1 Lattice sampling to within a specified generalized error tolerance, tolfun = max(abstol, reltol*| I |), i.e., | I - Q | <= tolfun with confidence of at least 99%, where I is the true integral value, Q is the estimated integral value, abstol is the absolute error tolerance, and reltol is the relative error tolerance. Usually the reltol determines the accuracy of the estimation; however, if | I | is rather small, then abstol determines the accuracy of the estimation. Given the construction of our Lattices, d must be a positive integer with 1 <= dim <= 600. For higher dimensions, it is recommended to use simpler periodization transformation like 'Baker'.

It is recommended to use **compInteg** for estimating the integral repeatedly after the object initialization.

OutP is the structure holding additional output params, more details provided below. Input f is a function handle that accepts an n x d matrix input, where d is the dimension of the hyperbox, and n is the number of points being evaluated simultaneously.

The following additional input parameter passing styles also supported:

[OBJ,Q] = **cubBayesLattice_g**(f,dim) estimates the integral of f over hyperbox $[0,1]^{\mathrm{dim}}$ using rank-1 Lattice sampling. All other input parameters are initialized with default values as given below. Returns the initialized object OBJ and the estimate of integral Q.

[OBJ,Q] = **cubBayesLattice_g**(f,dim,absTol,relTol); estimates the integral of f over hyperbox $[0,1]^{\mathrm{dim}}$ using rank-1 Lattice sampling. All parameters should be input in the order specified above. The answer is given within the generalized error tolerance tolfun. All other input parameters are initialized with default values as given below.

[OBJ,Q] = **cubBayesLattice_g**(f,dim,inParms); estimates the integral of f over hyperbox $[0,1]^{\mathrm{dim}}$ using rank-1 Lattice sampling. The structure inParams shall hold the optional input parameters.

Input Arguments

- f — the integrand.
- dim — number of dimensions of the integrand.

Optional Input Arguments

- absTol — absolute error tolerance | I - Q | <= absTol. Default is 0.01
- relTol — relative error tolerance | I - Q | <= I*relTol. Default is 0
- order — order of the Bernoulli polynomial of the kernel r=1,2. If r==0, algorithm automatically chooses the kernel order which can be a non-integer value. Default is 2
- ptransform — periodization variable transform to use: 'Baker', 'C0', 'C1', 'C1sin', or 'C2sin'. Default is 'C1sin'
- arbMean — If false, the algorithm assumes the integrand was sampled from a Gaussian process of zero mean. Default is 'true'
- alpha — confidence level for a credible interval of Q. Default is 0.01
- mmin — min number of samples to start with: 2^{mmin}. Default is 10
- mmax — max number of samples allowed: 2^{mmax}. Default is 22
- useGradient — If true uses gradient descent in parameter search. Default is false
- oneTheta — If true uses common shape parameter for all dimensions, else allow shape parameter vary across dimensions. Default is true

Output Arguments

- n — number of samples used to compute the integral of f.
- time — time to compute the integral in seconds.
- exitFlag — indicates the exit condition of the algorithm:

  1: integral computed within the error tolerance and without exceeding max sample limit 2^{mmax}

  2: used max number of samples and yet not met the error tolerance
- ErrBd — estimated integral error | I - Q |
- optParams — optional parameters useful to debug and get better understanding of the algorithm
- optParams.aMLEAll — returns the shape parameters computed

11.3 Guarantee

This algorithm attempts to calculate the integral of function f over the hyperbox $[0,1]^{\mathrm{dim}}$ to a prescribed error tolerance tolfun:= max(abstol,reltol*| I |) with guaranteed confidence level, e.g., 99% when alpha=0.5%. If the algorithm terminates without showing any warning messages and provides an answer Q, then the following inequality would be satisfied:

Pr(| Q - I | <= tolfun) = 99%

Please refer to our paper [25] for detailed arguments and proofs.

11.4 Examples

Example 1: Integrating a simple Quadratic function
Estimate the integral with integrand $f(x) = x^2$ over the interval $[0, 1]$ with default parameters: order=2, ptransform=C1sin, abstol=0.01, relTol=0

```
warning('off','GAIL:cubBayesLattice_g:fdnotgiven')
[~,muhat] = cubBayesLattice_g;
exactInteg = 1.0/3;
warning('on','GAIL:cubBayesLattice_g:fdnotgiven')
check = double(abs(exactInteg-muhat) < 0.01)
```

```
check =
     1
```

Example 2: ExpCos Estimate the integral of Exponential of Cosine function $f(x) = \exp\left(\sum_{i=1}^{2} cos(2\pi x_i)\right)$ over the interval $[0, 1]^2$ with parameters: order=2, C1sin variable transform, abstol=0.001, relTol=0.01

```
fun = @(x) exp(sum(cos(2*pi*x), 2));
dim=2; absTol=1e-3; relTol=1e-2;
exactInteg = besseli(0,1)^dim;
inputArgs = {'relTol',relTol, 'order',2, 'ptransform','C1sin'};
inputArgs = [inputArgs {'absTol',absTol,'oneTheta',false}];
obj=cubBayesLattice_g(fun,dim,inputArgs{:});
[muhat,outParams]=compInteg(obj);
check = double(abs(exactInteg-muhat) < max(absTol,relTol*abs(exactInteg)))
etaDim = size(outParams.optParams.aMLEAll, 2)
```

```
check =
     1
etaDim =
     2
```

Example 3: Keister function Estimate the integral with keister function as integrand over the interval $[0, 1]^2$ with parameters: order=2, C1 variable transform, abstol=0.001, relTol=0.01

```
dim=3; absTol=1e-3; relTol=1e-2;
normsqd = @(t) sum(t.*t,2); %squared l_2 norm of t
replaceZeros = @(t) (t+(t==0)*eps); % to avoid getting infinity, NaN
yinv = @(t)(erfcinv( replaceZeros(abs(t)) ));
ft = @(t,dim) cos( sqrt( normsqd(yinv(t)) )) *(sqrt(pi))^dim;
fKeister = @(x) ft(x,dim); exactInteg = Keistertrue(dim);
inputArgs ={'absTol',absTol, 'relTol',relTol};
inputArgs =[inputArgs {'order',2, 'ptransform','C1','arbMean',true}];
obj=cubBayesLattice_g(fKeister,dim,inputArgs{:});
[muhat,outParams]=compInteg(obj);
check = double(abs(exactInteg-muhat) < max(absTol,relTol*abs(exactInteg)))
etaDim = size(outParams.optParams.aMLEAll, 2)
```

```
check =
     1
etaDim =
     1
```

Example 4: Multivariate normal probability

Estimate the multivariate normal probability for the given hyper interval $\begin{pmatrix} -6 \\ -2 \\ -2 \end{pmatrix}$ and $\begin{pmatrix} 5 \\ 2 \\ 1 \end{pmatrix}$ in $\mathbf{R^3}$ having zero mean and covariance $\begin{pmatrix} 4 & 1 & 1 \\ 0 & 1 & 0.5 \\ 0 & 0 & 0.25 \end{pmatrix}$ with parameters: order=1, C1sin variable transform, abstol=0.001, relTol=0.01

```
dim=2; absTol=1e-3; relTol=1e-2; fName = 'MVN';
C = [4 1 1; 0 1 0.5; 0 0 0.25]; MVNParams.Cov = C'*C; MVNParams.C = C;
MVNParams.a = [-6 -2 -2]; MVNParams.b = [5 2 1]; MVNParams.mu = 0;
MVNParams.CovProp.C = chol(MVNParams.Cov)';
muBest = 0.676337324357787;
integrand =@(t) GenzFunc(t,MVNParams);
inputArgs={'absTol',absTol,'relTol',relTol};
inputArgs=[inputArgs {'order',1,'ptransform','C1sin','arbMean',true}];
inputArgs=[inputArgs {'useGradient',true}];
[~,muhat]=cubBayesLattice_g(integrand,dim, inputArgs{:});
check = double(abs(muBest-muhat) < max(absTol,relTol*abs(muBest)))
```

```
check =
     1
```

Example 5: Keister function

Estimating the Keister integral with Kernel order r chosen automatically

```
dim=2; absTol=1e-3; relTol=1e-2;
normsqd = @(t) sum(t.*t,2); %squared l_2 norm of t
replaceZeros = @(t) (t+(t==0)*eps); % to avoid getting infinity, NaN
yinv = @(t)(erfcinv( replaceZeros(abs(t)) ));
ft = @(t,dim) cos( sqrt( normsqd(yinv(t)) )) *(sqrt(pi))^dim;
fKeister = @(x) ft(x,dim); exactInteg = Keistertrue(dim);
inputArgs ={'absTol',absTol, 'relTol',relTol};
inputArgs =[inputArgs {'order',0, 'ptransform','C1','arbMean',true}];
obj=cubBayesLattice_g(fKeister,dim,inputArgs{:});
[muhat,outParams] = compInteg(obj);
check = double(abs(exactInteg-muhat) < max(absTol,relTol*abs(exactInteg)))
check = double(outParams.optParams.r > 0)
```

```
check =
     1
check =
     1
```

Example 6

A simple example which uses dimension specific shape parameter

```
const = [1E-4 1 1E4];
fun = @(x)sum(bsxfun(@times, const, sin(2*pi*x.^2)), 2);
dim=3; absTol=1e-3; relTol=1e-2;
```

```
exactInteg = fresnels(2)*sum(const)/2;
inputArgs = {'relTol',relTol, 'order',2, 'ptransform','C1sin'};
inputArgs = [inputArgs {'absTol',absTol,'oneTheta',false,'useGradient',false}];
obj=cubBayesLattice_g(fun, dim, inputArgs{:});
[muhat,outParams]=compInteg(obj);
check = double(abs(exactInteg-muhat) < max(absTol,relTol*abs(exactInteg)))
etaDim = size(outParams.optParams.aMLEAll, 2)
```

```
check =
     1
etaDim =
     3
```

11.5 See Also

cubBayesNet_g, cubSobol_g, cubLattice_g, cubMC_g, meanMC_g, meanMC_CLT, integral_g

12 cubBayesNet_g

Bayesian cubature method to estimate the integral of a random variable using digital nets over a d-dimensional region within a specified generalized error tolerance with guarantees under Bayesian assumptions. Currently, only Sobol points are supported.

12.1 Syntax

[OBJ,Q] = **cubBayesNet_g**(f,dim,'absTol',absTol,'relTol',relTol,'order',order,'arbMean',arbMean)

[OBJ] = **cubBayesNet_g**(f,dim,'absTol',absTol,'relTol',relTol,'order',order,'arbMean',arbMean)

[Q,OutP] = **compInteg**(OBJ)

[OBJ,Q] = **cubBayesNet_g**(f,dim)

[OBJ,Q] = **cubBayesNet_g**(f,dim,absTol,relTol)

[OBJ,Q] = **cubBayesNet_g**(f,dim,inParams)

12.2 Description

[OBJ,Q] = **cubBayesNet_g**(f,dim,'absTol',absTol,'relTol',relTol,'order',order, 'arbMean',arbMean); initializes the object with the given parameters and also returns an estimate of integral Q.

[Q,OutP] = **compInteg**(OBJ) estimates the integral of f over hyperbox $[0,1]^{\mathrm{dim}}$ using digital nets (Sobol points) to within a specified generalized error tolerance, tolfun = max(abstol, reltol*| I |), i.e., | I - Q | <= tolfun with confidence of at least 99%, where I is the true integral value, Q is the estimated integral value, abstol is the absolute error tolerance, and reltol is the relative error tolerance. Usually the reltol determines the accuracy of the estimation; however, if | I | is rather small, then abstol determines the accuracy of the estimation.

It is recommended to use **compInteg** for estimating the integral repeatedly after the object initialization.

OutP is the structure holding additional output params, more details provided below. Input f is a function handle that accepts an n x d matrix input, where d is the dimension of the hyperbox, and n is the number of points being evaluated simultaneously.

The following additional input parameter passing styles also supported:

[OBJ,Q] = **cubBayesNet_g**(f,dim); estimates the integral of f over hyperbox $[0,1]^{\mathrm{dim}}$ using digital nets (Sobol points). All other input parameters are initialized with default values as given below. Returns the initialized object OBJ and the estimate of integral Q.

[OBJ,Q] = **cubBayesNet_g**(f,dim,absTol,relTol); estimates the integral of f over hyperbox $[0,1]^{\mathrm{dim}}$ using digital nets (Sobol points). All parameters should be input in the order specified above. The answer is given within the generalized error tolerance tolfun. All other input parameters are initialized with default values as given below.

[OBJ,Q] = **cubBayesNet_g**(f,dim,inParms); estimates the integral of f over hyperbox $[0,1]^{\mathrm{dim}}$ using digital nets (Sobol points). The structure inParams shall hold the optional input parameters.

Input Arguments

- f — the integrand
- dim — number of dimensions of the integrand

Optional Input Arguments

- absTol — absolute error tolerance | I - Q | <= absTol. Default is 0.01
- relTol — relative error tolerance | I - Q | <= I*relTol. Default is 0
- arbMean — If false, the algorithm assumes the integrand was sampled from a Gaussian process of zero mean. Default is 'true'
- alpha — confidence level for a credible interval of Q. Default is 0.01
- mmin — min number of samples to start with: 2^{mmin}. Default is 8
- mmax — max number of samples allowed: 2^{mmax}. Default is 20

Output Arguments

- n — number of samples used to compute the integral of f.
- time — time to compute the integral in seconds.
- exitFlag — indicates the exit condition of the algorithm:
  1: integral computed within the error tolerance and without exceeding max sample limit 2^{mmax}
  2: used max number of samples and yet not met the error tolerance
- ErrBd — estimated integral error | I - Q |
- optParams — optional parameters useful to debug and get better understanding of the algorithm
- optParams.aMLEAll — returns the shape parameters computed

12.3 Guarantee

This algorithm attempts to calculate the integral of function f over the hyperbox $[0, 1]^{\text{dim}}$ to a prescribed error tolerance tolfun:= max(abstol,reltol*| I |) with guaranteed confidence level, e.g.,99% when alpha=0.5%. If the algorithm terminates without showing any warning messages and provides an answer Q, then the following inequality would be satisfied:

Pr(| Q - I | <= tolfun) = 99%

Please refer to our paper [24] for detailed arguments and proofs.

12.4 Examples

Example 1: Quadratic
Estimate the integral with integrand $f(x) = x^2$ over the interval $[0, 1]$ with default parameters: order=1, abstol=0.01, relTol=0

```
warning('off','GAIL:cubBayesNet_g:fdnotgiven')
[~,muhat] = cubBayesNet_g;
exactInteg = 1.0/3;
warning('on','GAIL:cubBayesNet_g:fdnotgiven')
check = double(abs(exactInteg-muhat) < 0.01)
```

```
check =
     1
```

Example 2: ExpCos Estimate the integral with integrand $f(x) = \exp\left(\sum_{i=1}^{2} cos(2\pi x_i)\right)$ over the interval $[0,1]^2$ with parameters: order=2, abstol=0.001, relTol=0.01

```
fun = @(x) exp(sum(cos(2*pi*x), 2));
dim=2; absTol=1e-3; relTol=1e-2;
exactInteg = besseli(0,1)^dim;
inputArgs = {'absTol',absTol,'relTol',relTol};
[~,muhat]=cubBayesNet_g(fun, dim, inputArgs{:});
check = double(abs(exactInteg-muhat) < max(absTol,relTol*abs(exactInteg)))
```

```
check =
     1
```

Example 3: Keister function Estimate the Keister's integrand, a multidimensional integral inspired by a physics application over the interval $[0,1]^2$ with parameters: order=2, abstol=0.001, relTol=0.01

```
dim=2; absTol=1e-3; relTol=1e-2;
normsqd = @(t) sum(t.*t,2); %squared l_2 norm of t
replaceZeros = @(t) (t+(t==0)*eps); % to avoid getting infinity, NaN
yinv = @(t)(erfcinv( replaceZeros(abs(t)) ));
ft = @(t,dim) cos( sqrt( normsqd(yinv(t)) )) *(sqrt(pi))^dim;
fKeister = @(x) ft(x,dim); exactInteg = Keistertrue(dim);
inputArgs ={'absTol',absTol, 'relTol',relTol};
inputArgs =[inputArgs {'arbMean',true}];
[~,muhat]=cubBayesNet_g(fKeister,dim,inputArgs{:});
check = double(abs(exactInteg-muhat) < max(absTol,relTol*abs(exactInteg)))
```

```
check =
     1
```

Example 4: Multivariate normal probability For $\mathbf{X} \sim \mathbf{N}(\mu, \mathbf{\Sigma})$, estimate the following probability:

$$P(\mathbf{a} \le \mathbf{X} \le \mathbf{b}) = \int_{\mathbf{a}}^{\mathbf{b}} \frac{\mathrm{e}^{(\mathbf{x}-\mu)^{\mathbf{T}}\mathbf{\Sigma}^{-1}(\mathbf{x}-\mu)}}{(2\pi)^{d/2}\left|\Sigma\right|^{1/2}}\,\mathrm{d}\mathbf{x}.$$

Given $\begin{pmatrix} -6 \\ -2 \\ -2 \end{pmatrix}$ and $\begin{pmatrix} 5 \\ 2 \\ 1 \end{pmatrix}$ with zero mean $\mu = 0$ and covariance $\begin{pmatrix} 4 & 1 & 1 \\ 0 & 1 & 0.5 \\ 0 & 0 & 0.25 \end{pmatrix}$.

```
C = [4 1 1; 0 1 0.5; 0 0 0.25]; MVNParams.Cov = C'*C; MVNParams.C = C;
MVNParams.a = [-6 -2 -2]; MVNParams.b = [5 2 1]; MVNParams.mu = 0;
MVNParams.CovProp.C = chol(MVNParams.Cov)';
muBest = 0.676337324357787;
integrand =@(t) GenzFunc(t,MVNParams);
inputArgs={'absTol',absTol,'relTol',relTol};
inputArgs=[inputArgs {'arbMean',true}];
obj=cubBayesNet_g(integrand,dim, inputArgs{:});
[muhat,outParams] = compInteg(obj);
check = double(abs(muBest-muhat) < max(absTol,relTol*abs(muBest)))
etaDim = size(outParams.optParams.aMLEAll, 2)
```

```
check =
     1
etaDim =
     1
```

12.5 See Also

cubBayesLattice_g, cubSobol_g, cubLattice_g, cubMC_g, meanMC_g, meanMC_CLT, integral_g

13 Demos

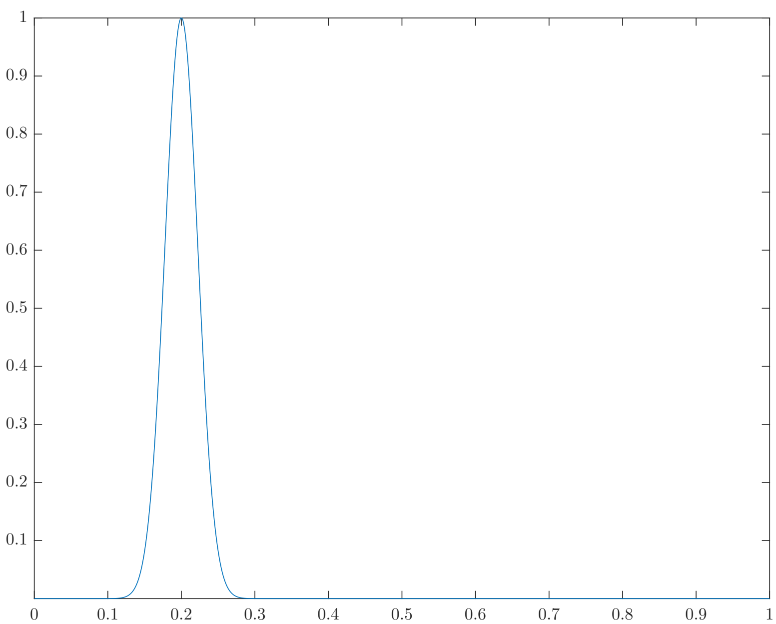
13.1 A GUI (graphical user interface) for funappx_g

To approximate a peaky function with **funappx_g** and to show how **funappx_g** generates grid points for locally adaptive linear spline approximation

Function definition

Define a peaky function as follows:

```
close all; clear all; format compact; format short;
f = @(x) exp(-1000*(x-0.2).^2);
x = 0:0.0001:1;
figure;
plot(x,f(x))
axis tight
```



Function Approximation

We use **funappx_g** to approximate f over the interval $[0,1]$ with error tolerance 10^{-2} and 15 initial subintervals:

```
[~,out_param] = funappx_g(@(x) exp(-1000*(x-0.2).^2),0,1,1e-2,15)
```

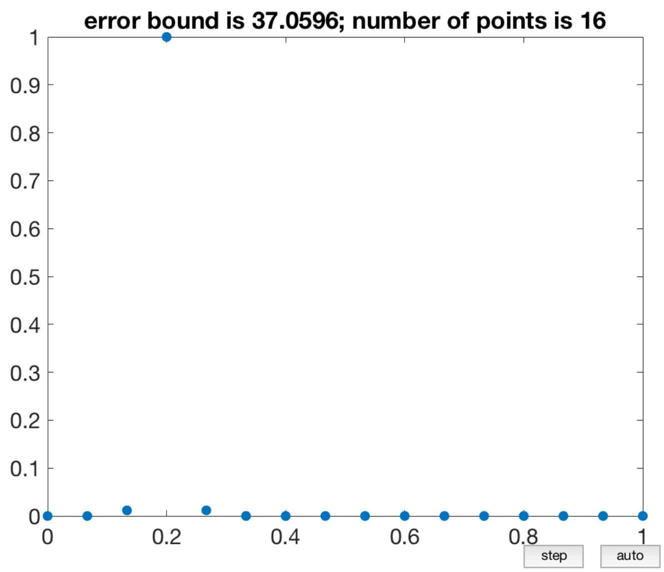
We find that to reach the error tolerance, we need 105 points to approximate the function.

```
out_param =
           a: 0
      abstol: 0.0100
           b: 1
           f: @(x)exp(-1000*(x-0.2).^2)
     maxiter: 1000
       ninit: 15
        nmax: 10000000
    exitflag: [0 0]
        iter: 7
```

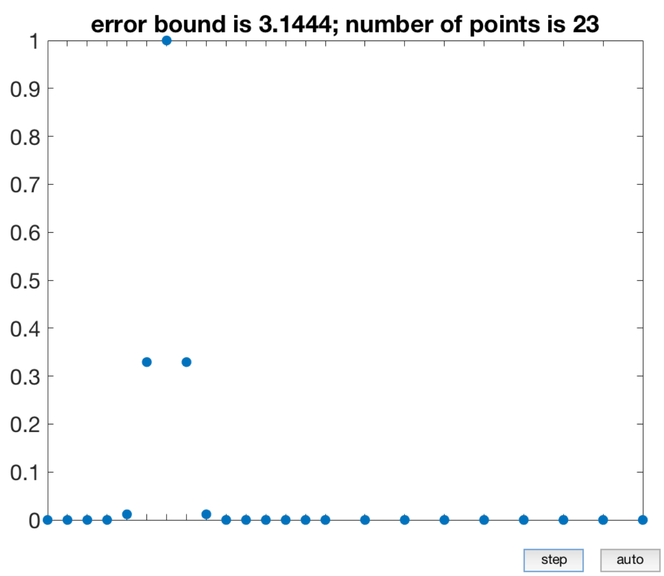
```
npoints: 105
 errest: 0.0028
```

Process to Generate Grid Points

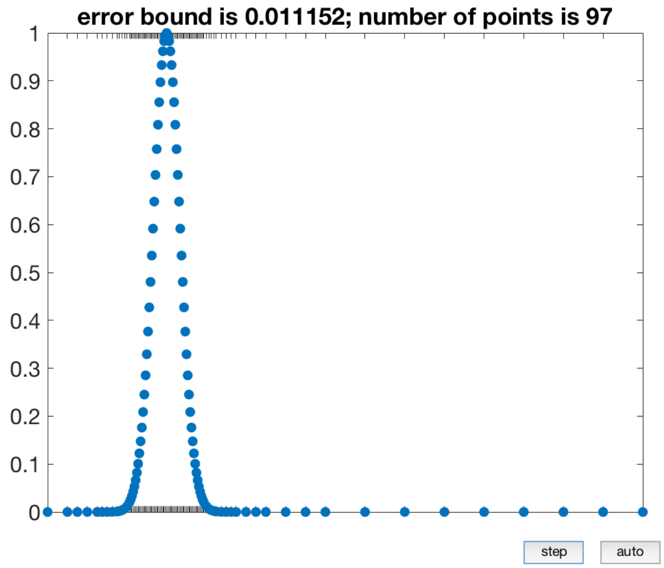
Step 1: start with 16 evenly spaced points:



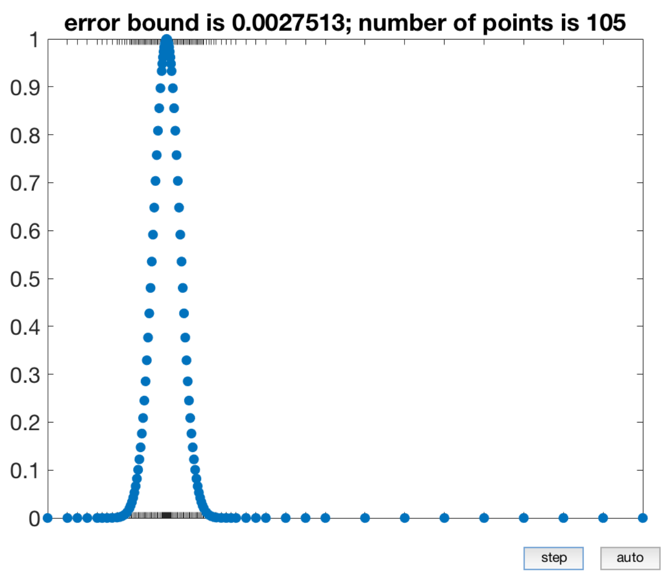
Step 2: add points to the peaky part:



Step 6: after several iterations, the approximation error almost meets the given tolerance:



Step 7: the error tolerance is reached:



This process can also be reproduced by the following command:
`funappx_g_gui(@(x) exp(-1000*(x-0.2).^2),0,1,1e-2,15,15);`

13.2 Compare funmin_g with fminbnd and chebfun

Function definition and minimization

Define a function with two minima as follows:

$$f(x) = -5\exp(-100(x-0.15)^2) - \exp(-80(x-0.65)^2).$$

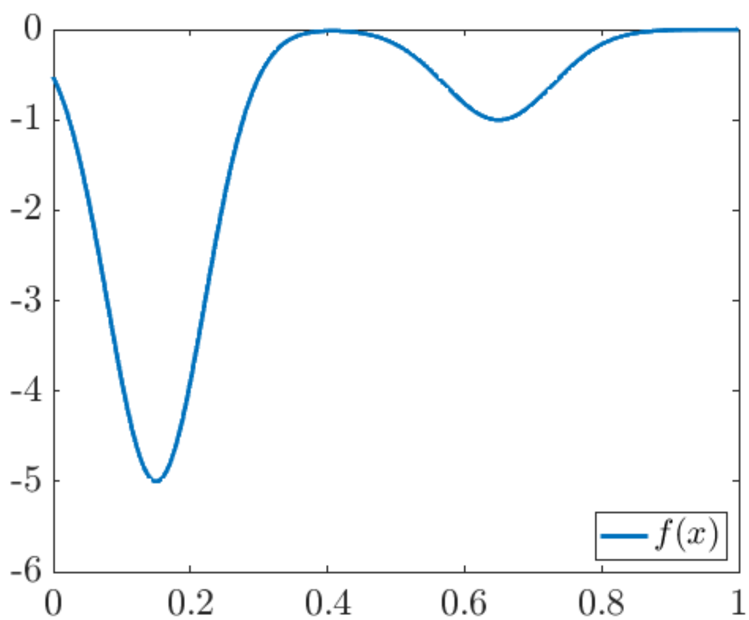
We use **funmin_g** [6, 28], MATLAB's **fminbnd** [1, 15], and Chebfun's **min** [16] to find the minimum of f over the interval $[0, 1]$.

Set up

```
close all; clearvars; format compact; format short;
gail.InitializeDisplay
set(0,'defaultLineMarkerSize',15)
```

Plot function

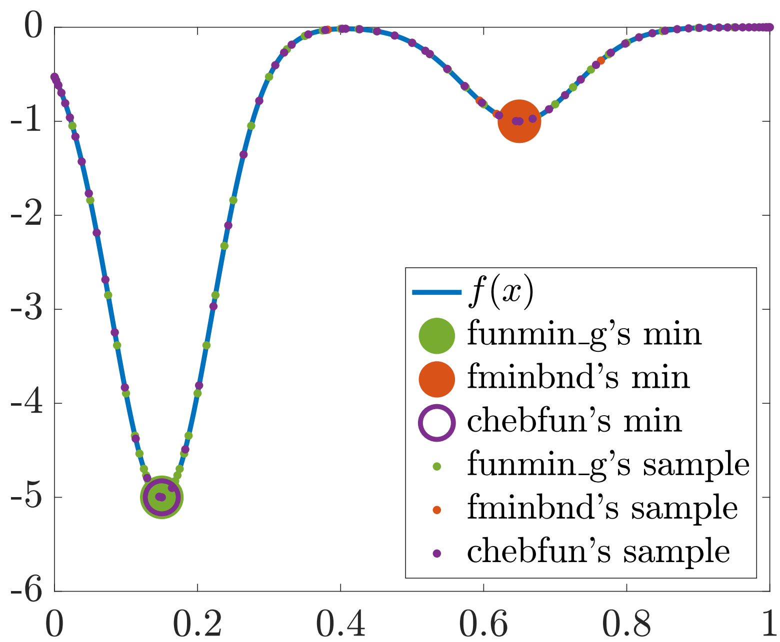
```
xplot = 0:0.001:1;
fplot = fmin_ex1(xplot);
h(1) = plot(xplot,fplot,'-');
set(h(1),'color',MATLABBlue)
h_legend = legend([h(1)],{'$f(x)$'},'Location','Southeast');
set(h_legend,'interpreter','latex');
hold on
```



Plot minimum values and sample points

```
xAll = [];
fAll = [];
save fmin_ex1X xAll fAll
[ffmg,outfmg] = funmin_g(@fmin_ex1,0,1);
h(2) = plot(mean(outfmg.intervals),ffmg,'.');
set(h(2),'color',MATLABGreen,'MarkerSize',80)
load fmin_ex1X xAll fAll
h(3) = plot(xAll,fAll,'.');
set(h(3),'color',MATLABGreen)

% fminbnd
xAll = [];
fAll = [];
```

```
save fmin_ex1X xAll fAll
options = optimset('TolX',outfmg.abstol,'TolFun',outfmg.abstol);
[xfmb,ffmb] = fminbnd(@fmin_ex1,0,1,options);
h(4) = plot(xfmb,ffmb,'.');
set(h(4),'color',MATLABOrange,'MarkerSize',80)
load fmin_ex1X xAll fAll
h(5) = plot(xAll,fAll,'.');
set(h(5),'color',MATLABOrange)

% chebfun
xAll = [];
fAll = [];
save fmin_ex1X xAll fAll
chebf = chebfun(@fmin_ex1,[0,1],'chebfuneps', outfmg.abstol, 'splitting','on');
chebfval = min(chebf);
chebxvals = roots(diff(chebf));
[v,i] = min(abs(fmin_ex1(chebxvals)-chebfval));
chebxval = chebxvals(i);
chebn = length(chebf);
h(6) = plot(chebxval,chebfval,'o');
set(h(6),'color',MATLABPurple,'MarkerSize',20)
load fmin_ex1X xAll fAll
h(7) = plot(xAll,fAll,'.');
set(h(7),'color',MATLABPurple)
h_legend = legend([h(1) h(2) h(4) h(6) h(3) h(5) h(7)],{'$f(x)$','funmin\_g''s min',...
    'fminbnd''s min','chebfun''s min','funmin\_g''s sample',...
    'fminbnd''s sample','chebfun''s sample'},...
    'Location','Southeast');
set(h_legend,'interpreter','latex');

function y = fmin_ex1(x)
if exist('fmin_ex1X.mat','file')
   load fmin_ex1X xAll fAll
else
   xAll = [];
   fAll = [];
end
xAll = [xAll; x(:)];
y = -5*exp(-100*(x-0.15).^2) - exp(-80*(x-0.65).^2);
fAll = [fAll; y(:)];
save fmin_ex1X xAll fAll
end
```



13.3 Integrate a spiky function using integral_g

Function definition

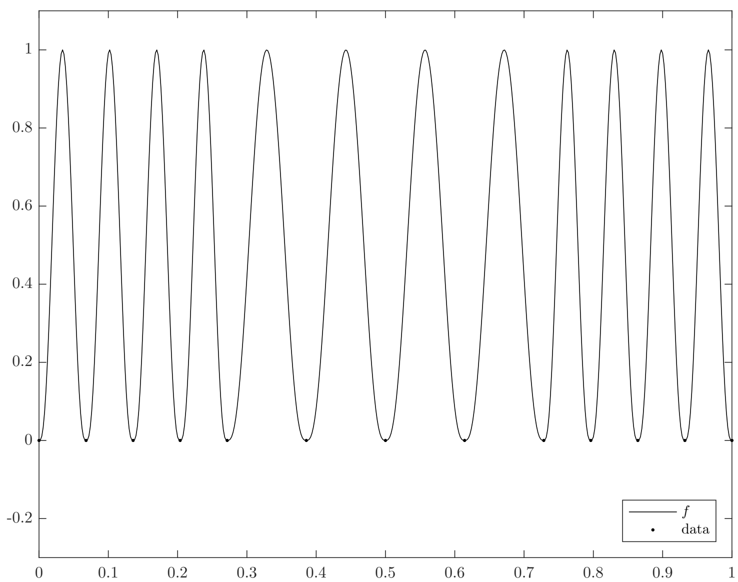
This example is taken from [1], where a function is defined on $[0, 1]$ with twelve spikes.

```
close all; clear all; format compact; format short e;
[~,~,MATLABVERSION] = GAILstart(false);

xquad = 0.13579; %number used by quad to split interval into three parts
xleft = [0 xquad/2 xquad 3*xquad/2 2*xquad];
xctr = [2*xquad 1/4+xquad 1/2 3/4-xquad 1-2*xquad];
xrght = [1-2*xquad 1-3*xquad/2 1-xquad 1-xquad/2 1];
xall = [xleft xctr(2:5) xrght(2:5)]';
nnode = length(xall);

fbump = @(x) 4^3*((x.*(1-x)).^3).*((x>=0)&(x<=1)); %one bump
xplot = (0:0.002:1)'; %points to plot
spikyfun = @(x) foolfunmaker(x, @(x,c) fbump((x-c(1))/c(2)),...
    ones(nnode-1,1), [xall(1:nnode-1) diff(xall)]);
```

Plot of the spiky function

In the following, we plot $f(x)$ and show the data sampling points picked by MATLAB's built-in integration function **quad**, which explains why **quad** essentially gives the answer zero for our spiky function:

```
figure;
h = plot(xplot,spikyfun(xplot), 'k-', xall, zeros(nnode,1), 'k.');
axis([0 1 -0.3 1.1])
set(gca,'Ytick',-0.2:0.2:1)
legend(h,{'$f$','data'},'location','southeast')
```



Integral approximation

We use MATLAB built-in functions and **integral_g** from GAIL to integrate f over the unit interval:

```
a = 0;
b = 1;
abstol = 1e-11;
if MATLABVERSION >= 8,
    MATintegralspiky = integral(spikyfun,a,b,'AbsTol',abstol)
end
MATquadspiky = quad(spikyfun,a,b,abstol)
MATgailspiky = integral_g(spikyfun,a,b,abstol)
```

```
MATintegralspiky =
   4.5714e-01
MATquadspiky =
   2.7021e-44
MATgailspiky =
   4.5714e-01
```

Compute approximation errors

The true integral value of the spiky function is 16/35. The following code computes absolute errors from the above approximation methods. Only **integral_g** achieves the required accuracy with respect to the absolute tolerance of 10^{-11} in this example.

```
integralspiky = 16/35;
if MATLABVERSION >= 8,
  abs_errors = abs(integralspiky - [MATintegralspiky, MATquadspiky, MATgailspiky])
else
  abs_errors = abs(integralspiky - [MATquadspiky, MATgailspiky])
end
if_meet_abstol = (abs_errors < abstol)
```

```
abs_errors =
   6.1854e-10   4.5714e-01   1.4322e-14
if_meet_abstol =
     0     0     1
```

13.4 Counting the success rate of meanMC_g

Define an integration problem as follows:

$$I = \int_0^1 x^2 dx.$$

The analytical solution is $\frac{\{}{1}\}\{3\}$. If we use **meanMC_g** to estimate the integral with 1000 replications, we expect the success rate to be bigger than or equal to `(1 - alpha)`.

```
success = 0;
n = 1000;
in_param.reltol = 0; in_param.abstol = 1e-3;
in_param.alpha = 0.05; Yrand = @(n) rand(n,1).^2;
exactsol = 1/3;
for i = 1:n,
    tmu = meanMC_g(Yrand,in_param);
    check = abs(exactsol-tmu) < 1e-3;
    if check == 1,
        success = success + 1;
    end
end
disp(['Over ' num2str(n) ' replications, there are ' num2str(success) ' successes.'])
disp(['The success rate is ' num2str(success/n) ', which is larger than '...
    num2str(1-in_param.alpha) '.'])
```

```
Over 1000 replications, there are 991 successes.
The success rate is 0.991, which is larger than 0.95.
```

13.5 Estimation of normal probabilities by by multiple integration algorithms in GAIL

For $\mathbf{X} \sim \mathbf{N}(\mu, \mathbf{\Sigma})$, we will estimate the following probability:

$$P(\mathbf{a} \le \mathbf{X} \le \mathbf{b}) = \int_{\mathbf{a}}^{\mathbf{b}} \frac{\mathrm{e}^{(\mathbf{x}-\mu)^{\mathbf{T}}\mathbf{\Sigma}^{-1}(\mathbf{x}-\mu)}}{(2\pi)^{d/2}\left|\Sigma\right|^{1/2}}\,\mathrm{d}\mathbf{x}.$$

We present three tests, each of which approximates the aforementioned probability using **cubSobol_g**, **cubMC_g** and **cubBayesLattice_g**, **cubLattice_g**, and **cubBayesNet_g** which are quasi-Monte Carlo, IID Monte Carlo and Bayesian cubature algorithms respectively in GAIL. In order to facilitate the computations when d is high (>4), we are going to apply a special transformation of the integrand proposed by Alan Genz.

Basic integration parameters set up

For all the examples, the dimension of the problem is $d = 20$. The user input tolerances are also set up below: `abstol` is the absolute error tolerance, and `reltol` the relative error tolerance. When `reltol` is set to 0, the algorithms use pure absolute error bound, and vice versa. Finally, for simplicity we define the mean of the distribution to be $\mu = \mathbf{0}$:

```
function demo_normal_probabilities_small(nRep)

d = 20;  % Dimension of the problem
abstol = 1e-3; % User input, absolute error bound
reltol = 0;  % User input, relative error bound
mu = zeros(d,1); % Mean of the distribution
if nargin < 1
   nRep = 10;
end
nTest = 2;
Ivec(nTest) = 0;
approx_prob_MC(nRep,nTest) = 0;
% out_param_MC(nRep,nTest) = 0;
timeMC(nRep,nTest) = 0;
nSampleMC(nRep,nTest) = 0;

approx_prob_sobol(nRep,nTest) = 0;
% out_param_sobol(nRep,nTest) = 0;
timeSob(nRep,nTest) = 0;
nSampleSob(nRep,nTest) = 0;

approx_prob_lat(nRep,nTest) = 0;
% out_param_lat(nRep,nTest) = 0;
timeLat(nRep,nTest) = 0;
nSampleLat(nRep,nTest) = 0;


approx_prob_BayLat(nRep,nTest) = 0;
% out_param_BayLat(nRep,nTest) = 0;
timeBayLat(nRep,nTest) = 0;
nSampleBayLat(nRep,nTest) = 0;

approx_prob_BaySob(nRep,nTest) = 0;
timeBaySob(nRep,nTest) = 0;
nSampleBaySob(nRep,nTest) = 0;
```

First test: $\Sigma = I_d$

For this first example, we consider $\Sigma = I_d$, and $\mathbf{b} = -\mathbf{a} = (\mathbf{3.5}, \dots, \mathbf{3.5})$. In this case, the solution of the integral is known so we can verify that the error conditions are met:

```
Sigma = eye(d); % We set the covariance matrix to the identity
factor = 3.5;
hyperbox = [-factor*ones(1,d) ; factor*ones(1,d)]; % We define the integration limits
exactsol = (gail.stdnormcdf(factor)-gail.stdnormcdf(-factor))^d; % Exact integral solution
Ivec(1) = exactsol;
```

```
% Solution approx_prob and integration output parameters in out_param
% Test 1.1: cubMC_g
for k=1:nRep
  [approx_prob_MC(k,1),out_param_MC(k,1)] = multi_normcdf_cubMC(hyperbox,mu,Sigma,abstol,reltol);
end
timeMC(:,1) = [out_param_MC(:,1).time];
nSampleMC(:,1) = [out_param_MC(:,1).ntot];
report_integration_result('Test 1.1', 'cubMC_g',abstol,reltol,exactsol,...
  mean(approx_prob_MC(:,1)),mean(timeMC(:,1)),mean(nSampleMC(:,1)))

% Test 1.2: cubLattice_g
for k=1:nRep
  [approx_prob_lat(k,1),out_param_lat(k,1)] =
   multi_normcdf_cubLat(hyperbox,mu,Sigma,abstol,reltol);
end
timeLat(:,1) = [out_param_lat(:,1).time];
nSampleLat(:,1) = [out_param_lat(:,1).n];
report_integration_result('Test 1.2', 'cubLattice_g',abstol,reltol,exactsol,...
  mean(approx_prob_lat(:,1)),mean(timeLat(:,1)),mean(nSampleLat(:,1)))

% Test 1.3: cubSobol_g
for k=1:nRep
  [approx_prob_sobol(k,1),out_param_sobol(k,1)] =
   multi_normcdf_cubSobol(hyperbox,mu,Sigma,abstol,reltol);
end
timeSob(:,1) = [out_param_sobol(:,1).time];
nSampleSob(:,1) = [out_param_sobol(:,1).n];
report_integration_result('Test 1.3', 'cubSobol_g',abstol,reltol,exactsol,...
  mean(approx_prob_sobol(:,1)),mean(timeSob(:,1)),mean(nSampleSob(:,1)))

% Test 1.4: cubBayesLattice_g
for k=1:nRep
  [approx_prob_BayLat(k,1),out_param_BayLat(k,1)] =
   multi_normcdf_cubBayesLat(hyperbox,mu,Sigma,abstol,reltol);
end
timeBayLat(:,1) = [out_param_BayLat(:,1).time];
nSampleBayLat(:,1) = [out_param_BayLat(:,1).n];
report_integration_result('Test 1.4', 'cubBayesLattice_g', abstol,reltol,...
   NaN,mean(approx_prob_BayLat(:,1)), (mean(timeBayLat(:,1))), (mean(nSampleBayLat(:,1))))

% Test 1.5: cubBayesNet_g
for k=1:nRep
  [approx_prob_BaySob(k,1),out_param_BaySob(k,1)] =
   multi_normcdf_cubBayesNet(hyperbox,mu,Sigma,abstol,reltol);
end
timeBaySob(:,1) = [out_param_BaySob(:,1).time];
nSampleBaySob(:,1) = [out_param_BaySob(:,1).n];
report_integration_result('Test 1.5','cubBayesNet_g',abstol,reltol,NaN,...
   mean(approx_prob_BaySob(:,1)),mean(timeBayLat(:,1)),mean(nSampleBayLat(:,1)))
```

```
Test 1.1: cubMC_g
  Estimated probability: 0.990736
       True probability: 0.990736
  The algorithm took 0.050 seconds and 10013 points
```

```
  Real error is 4.441e-16, which is less than the tolerance 1.000e-03
Test 1.2: cubLattice_g
  Estimated probability: 0.990736
       True probability: 0.990736
  The algorithm took 0.023 seconds and 1024 points
  Real error is 2.709e-14, which is less than the tolerance 1.000e-03
Test 1.3: cubSobol_g
  Estimated probability: 0.990736
       True probability: 0.990736
  The algorithm took 0.018 seconds and 1024 points
  Real error is 2.709e-14, which is less than the tolerance 1.000e-03
Test 1.4: cubBayesLattice_g
  Estimated probability: 0.990736
  The algorithm took 0.004 seconds and 256 points
Test 1.5: cubBayesNet_g
  Estimated probability: 0.990736
  The algorithm took 0.004 seconds and 256 points
```

Second test: $\Sigma = 0.4I_d + 0.6\mathbf{1}\mathbf{1}^\mathbf{T}$

For this second example, we consider $\Sigma = 0.4I_d + 0.6\mathbf{1}\mathbf{1}^\mathbf{T}$ (1 on the diagonal, 0.6 off the diagonal), $\mathbf{a} = (-\infty, \ldots, -\infty)$, and $\mathbf{b} = \sqrt{\mathbf{d}}(\mathbf{U_1}, \ldots, \mathbf{U_d})$ ($\mathbf{b}$ is chosen randomly). The solution for this integral is known too so we can verify the real error:

```
sig = 0.6;
Sigma = sig*ones(d,d); Sigma(1:d+1:d*d) = 1; % set the covariance matrix
hyperbox = [-Inf*ones(1,d) ; sqrt(d)*rand(1,d)]; % define the integration limits
exactsol = integral(@(t)MVNPexact(t,hyperbox(2,:),sig),...
  -inf, inf,'Abstol',1e-8,'RelTol',1e-8)/sqrt(2*pi);
Ivec(2) = exactsol;

% Solution approx_prob and integration output parameters in out_param
% Test 2.1: cubMC_g
for k=1:nRep
  [approx_prob_MC(k,2),out_param_MC(k,2)] =
   multi_normcdf_cubMC(hyperbox,mu,Sigma,abstol,reltol);
end
timeMC(:,2) = [out_param_MC(:,2).time];
nSampleMC(:,2) = [out_param_MC(:,2).ntot];
report_integration_result('Test 2.1','cubMC_g',abstol,reltol,...
  exactsol,mean(approx_prob_MC(:,2)),mean(timeMC(:,2)),mean(nSampleMC(:,2)))

% Test 2.2: cubLattice_g
for k=1:nRep
  [approx_prob_lat(k,2),out_param_lat(k,2)] =
   multi_normcdf_cubLat(hyperbox,mu,Sigma,abstol,reltol);
end
timeLat(:,2) = [out_param_lat(:,2).time];
nSampleLat(:,2) = [out_param_lat(:,2).n];
report_integration_result('Test 2.2','cubLattice_g',abstol,reltol,...
  exactsol,mean(approx_prob_lat(:,2)),mean(timeLat(:,2)),mean(nSampleLat(:,2)))

% Test 2.3: cubSobol_g
for k=1:nRep
```

```
  [approx_prob_sobol(k,2),out_param_sobol(k,2)] =
   multi_normcdf_cubSobol(hyperbox,mu,Sigma,abstol,reltol);
end
timeSob(:,2) = [out_param_sobol(:,2).time];
nSampleSob(:,2) = [out_param_sobol(:,2).n];
report_integration_result('Test 2.3','cubSobol_g',abstol,reltol,...
  exactsol,mean(approx_prob_sobol(:,2)),mean(timeSob(:,2)),mean(nSampleSob(:,2)))

% Test 2.4: cubBayesLattice_g
for k=1:nRep
  [approx_prob_BayLat(k,2),out_param_BayLat(k,2)] = multi_normcdf_cubBayesLat(...
    hyperbox,mu,Sigma,abstol,reltol);
end
timeBayLat(:,2) = [out_param_BayLat(:,2).time];
nSampleBayLat(:,2) = [out_param_BayLat(:,2).n];
report_integration_result('Test 2.4','cubBayesLattice_g',abstol,reltol,...
  NaN,mean(approx_prob_BayLat(:,2)),mean(timeBayLat(:,2)),mean(nSampleBayLat(:,2)))

% Test 2.5: cubBayesNet_g
for k=1:nRep
  [approx_prob_BaySob(k,2),out_param_BaySob(k,2)] = multi_normcdf_cubBayesNet(...
    hyperbox,mu,Sigma,abstol,reltol);
end
timeBaySob(:,2) = [out_param_BaySob(:,2).time];
nSampleBaySob(:,2) = [out_param_BaySob(:,2).n];
report_integration_result('Test 2.5','cubBayesNet_g',abstol,reltol,...
  NaN,mean(approx_prob_BaySob(:,2)),mean(timeBaySob(:,2)),mean(nSampleBaySob(:,2)))


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Ivec = repmat(Ivec,nRep,1);

absErrMC = abs(Ivec-approx_prob_MC);
succMC = mean(absErrMC <= abstol)
avgAbsErrMC = mean(absErrMC)

absErrSob = abs(Ivec-approx_prob_sobol);
succSob = mean(absErrSob <= abstol)
avgAbsErrSob = mean(absErrSob)

absErrLat = abs(Ivec-approx_prob_lat);
succLat = mean(absErrLat <= abstol)
avgAbsErrLat = mean(absErrLat)

absErrBayLat = abs(Ivec-approx_prob_BayLat);
succBayLat = mean(absErrBayLat <= abstol)
avgAbsErrBayLat = mean(absErrBayLat)

absErrBaySob = abs(Ivec-approx_prob_BaySob);
succBaySob = mean(absErrBaySob <= abstol)
avgAbsErrBaySob = mean(absErrBaySob)

timeMC        = mean(timeMC);
timeLat       = mean(timeLat);
```

```
timeSob       = mean(timeSob);
timeBayLat    = mean(timeBayLat);
timeBaySob    = mean(timeBaySob);
nSampleMC     = mean(nSampleMC);
nSampleLat    = mean(nSampleLat);
nSampleSob    = mean(nSampleSob);
nSampleBayLat = mean(nSampleBayLat);
nSampleBaySob = mean(nSampleBaySob);

outFileName = gail.save_mat('Paper_cubBayesLattice_g',['MVNCubExBayesDataNRep' int2str(nRep)],...
    true, abstol, ...
    avgAbsErrMC, avgAbsErrLat, avgAbsErrSob, avgAbsErrBayLat, avgAbsErrBaySob, ...
    succMC, succLat, succSob, succBayLat, succBaySob, ...
    timeMC, timeLat, timeSob, timeBayLat, timeBaySob, ...
    nSampleMC, nSampleLat, nSampleSob, nSampleBayLat, nSampleBaySob);

MVNCubExBayesOut(outFileName)
fprintf('')
```

```
Test 2.1: cubMC_g
  Estimated probability: 0.244175
       True probability: 0.244200
  The algorithm took 1.827 seconds and 1.155954e+06 points
  Real error is 2.451e-05, which is less than the tolerance 1.000e-03
Test 2.2: cubLattice_g
  Estimated probability: 0.244255
       True probability: 0.244200
  The algorithm took 0.009 seconds and 2048 points
  Real error is 5.495e-05, which is less than the tolerance 1.000e-03
Test 2.3: cubSobol_g
  Estimated probability: 0.244012
       True probability: 0.244200
  The algorithm took 0.010 seconds and 2048 points
  Real error is 1.880e-04, which is less than the tolerance 1.000e-03
Test 2.4: cubBayesLattice_g
  Estimated probability: 0.244300
  The algorithm took 0.042 seconds and 8192 points
Test 2.5: cubBayesNet_g
  Estimated probability: 0.244225
  The algorithm took 0.244 seconds and 8192 points
succMC =
     1     1
avgAbsErrMC =
   1.0e-03 *
    0.0000    0.1074
succSob =
     1     1
avgAbsErrSob =
   1.0e-03 *
    0.0000    0.3296
succLat =
    1.0000    0.8000
avgAbsErrLat =
   1.0e-03 *
```

```
    0.0000    0.4478
succBayLat =
     1     1
avgAbsErrBayLat =
   1.0e-03 *
    0.0000    0.1620
succBaySob =
     1     1
avgAbsErrBaySob =
   1.0e-04 *
    0.0000    0.8314
```

Third test: $\Sigma = 0.4I_d + 0.6\mathbf{1}\mathbf{1}^\mathbf{T}$

For this last example, we consider the same covariance matrix in the second test but the upper and lower limits are different, $\mathbf{a} = -\mathbf{d/3}(\mathbf{U_1}, \ldots, \mathbf{U_d})$, and $\mathbf{b} = \mathbf{d/3}(\mathbf{U_{d+1}}, \ldots, \mathbf{U_{2d}})$ (both $\mathbf{a}$ and $\mathbf{b}$ are chosen randomly):

```
hyperbox = [-(d/3)*rand(1,d) ; (d/3)*rand(1,d)]; % We define the integration limits

% Solution approx_prob and integration output parameters in out_param
% Test 3.1: cubMC_g
[approx_prob,out_param] = multi_normcdf_cubMC(hyperbox,mu,Sigma,abstol,reltol);
report_integration_result('Test 3.1','cubMC_g',abstol,reltol,...
  NaN,approx_prob,out_param.time,out_param.ntot)

% Test 3.2: cubSobol_g
[approx_prob,out_param] = multi_normcdf_cubSobol(hyperbox,mu,Sigma,abstol,reltol);
report_integration_result('Test 3.2','cubSobol_g',abstol,reltol,...
  NaN,approx_prob,out_param.time,out_param.n)

% Test 3.3: cubBayesLattice_g
[approx_prob,out_param] = multi_normcdf_cubBayesLat(hyperbox,mu,...
  Sigma,abstol,reltol);
report_integration_result('Test 3.3','cubBayesLattice_g',abstol,reltol,...
  NaN,approx_prob,out_param.time,out_param.n)

% Test 3.4: cubBayesNet_g
[approx_prob,out_param] = multi_normcdf_cubBayesNet(hyperbox,mu,...
  Sigma,abstol,reltol);
report_integration_result('Test 3.4','cubBayesNet_g',abstol,reltol,...
  NaN,approx_prob,out_param.time,out_param.n)
fprintf('')
```

```
Test 3.1: cubMC_g
  Estimated probability: 0.035129
  The algorithm took 0.078 seconds and 38167 points
Test 3.2: cubSobol_g
  Estimated probability: 0.035451
  The algorithm took 0.014 seconds and 1024 points
Test 3.3: cubBayesLattice_g
  Estimated probability: 0.035162
  The algorithm took 0.015 seconds and 1024 points
```

Appendix: Auxiliary function definitions

The following functions are defined for the above test examples. `multi_normcdf_cubSobol` and `multi_normcdf_cubMC` redefine **cubSobol_g** and **cubMC_g** respectively for computing normal probabilities based on Alan Genz's transformation. `f` is the function resulting from applying Alan Genz's transform that is called in either **cubSobol_g** or **cubMC_g**.

```
function [p,out, y, kappanumap] = multi_normcdf_cubSobol(hyperbox,mu,...
    Sigma,abstol,reltol)
  % Using cubSobol_g, multi_normcdf_cubMC computes the cumulative
  % distribution function of the multivariate normal distribution with mean
  % mu, covariance matrix Sigma and within the region defined by hyperbox.
  hyperbox = bsxfun(@minus, hyperbox, mu');
  C = chol(Sigma)'; d = size(C,1);
  a = hyperbox(1,1)/C(1,1); b = hyperbox(2,1)/C(1,1);
  s = gail.stdnormcdf(a); e = gail.stdnormcdf(b);
  [p, out, y, kappanumap] = cubSobol_g(...
    @(x) f(s,e,hyperbox,x,C), [zeros(1,d-1);ones(1,d-1)],...
    'uniform',abstol,reltol);
end

function [p,out, y, kappanumap] = multi_normcdf_cubLat(hyperbox,mu,...
    Sigma,abstol,reltol)
  % Using cubLattice_g, multi_normcdf_cubLat computes the cumulative
  % distribution function of the multivariate normal distribution with mean
  % mu, covariance matrix Sigma and within the region defined by hyperbox.
  hyperbox = bsxfun(@minus, hyperbox, mu');
  C = chol(Sigma)'; d = size(C,1);
  a = hyperbox(1,1)/C(1,1); b = hyperbox(2,1)/C(1,1);
  s = gail.stdnormcdf(a); e = gail.stdnormcdf(b);
  [p, out, y, kappanumap] = cubLattice_g(...
    @(x) f(s,e,hyperbox,x,C), [zeros(1,d-1);ones(1,d-1)],...
    'uniform',abstol,reltol);
end

function [p,out] = multi_normcdf_cubBayesLat(hyperbox,mu,Sigma,abstol,reltol)
  % Using cubBayesLattice_g, multi_normcdf_cubBayesLat computes the cumulative
  % distribution function of the multivariate normal distribution with mean
  % mu, covariance matrix Sigma and within the region defined by hyperbox.

  hyperbox = bsxfun(@minus, hyperbox, mu');
  C = chol(Sigma)';
  a = hyperbox(1,1)/C(1,1); b = hyperbox(2,1)/C(1,1);
  s = gail.stdnormcdf(a); e = gail.stdnormcdf(b);

  [~,dim] = size(hyperbox);
  inputArgs = {'dim',dim, 'absTol',abstol, 'reltol',reltol, ...
    'order',1, 'ptransform','Baker', ....
    'stopAtTol',true, 'stopCriterion','full'...
    'arbMean',true, 'alpha',0.01 ...
    'optTechnique','None'};

  inputArgs{end+1} = 'f'; inputArgs{end+1} = @(x) f(s,e,hyperbox,x,C);
  inputArgs{end+1} = 'fName'; inputArgs{end+1} = 'MVN';
```

```
  objCubBayes=cubBayesLattice_g(inputArgs{:});
  [p,out]=compInteg(objCubBayes);

end

function [p,out] = multi_normcdf_cubBayesNet(hyperbox,mu,Sigma,abstol,reltol)
  % Using cubBayesLattice_g, multi_normcdf_cubBayes computes the cumulative
  % distribution function of the multivariate normal distribution with mean
  % mu, covariance matrix Sigma and within the region defined by hyperbox.

  hyperbox = bsxfun(@minus, hyperbox, mu');
  C = chol(Sigma)';
  a = hyperbox(1,1)/C(1,1); b = hyperbox(2,1)/C(1,1);
  s = gail.stdnormcdf(a); e = gail.stdnormcdf(b);

  [~,dim] = size(hyperbox);
  inputArgs = {'dim',dim, 'absTol',abstol, 'reltol',reltol, ...
    'order',1, ....
    'stopAtTol',true, 'stopCriterion','full'...
    'arbMean',true, 'alpha',0.01 ...
    'optTechnique','None'};

  inputArgs{end+1} = 'f'; inputArgs{end+1} = @(x) f(s,e,hyperbox,x,C);
  inputArgs{end+1} = 'fName'; inputArgs{end+1} = 'MVN';

  objCubBayes=cubBayesNet_g(inputArgs{:});
  [p,out]=compInteg(objCubBayes);

end

function [Q,param] = multi_normcdf_cubMC(hyperbox,mu,Sigma,abstol,reltol)
  % Using cubMC_g, multi_normcdf_cubMC computes the cumulative distribution
  % function of the multivariate normal distribution with mean mu, covariance
  % matrix Sigma and within the region defined by hyperbox.
  hyperbox = bsxfun(@minus, hyperbox, mu');
  C = chol(Sigma)'; d = size(C,1);
  a = hyperbox(1,1)/C(1,1); b = hyperbox(2,1)/C(1,1);
  s = gail.stdnormcdf(a); e = gail.stdnormcdf(b);
  [Q,param] = cubMC_g(...
    @(x) f(s,e,hyperbox,x,C), [zeros(1,d-1);ones(1,d-1)],...
    'uniform',abstol,reltol);
end

function f_eval = f(s,e,hyperbox,w,C)
  % This is the integrand resulting from applying Alan Genz's transformation,
  % which is recursively defined.
  f_eval = (e-s)*ones(size(w,1),1);
  aux = ones(size(w,1),1);
  y = [];
  for i = 2:size(hyperbox,2);
    y = [y gail.stdnorminv(s+w(:,i-1).*(e-s))];
    aux = sum(bsxfun(@times,C(i,1:i-1),y),2);
    a = (hyperbox(1,i)-aux)/C(i,i);
    b = (hyperbox(2,i)-aux)/C(i,i);
```

```
    s = gail.stdnormcdf(a);
    e = gail.stdnormcdf(b);
    f_eval = f_eval .* (e-s);
  end

  f_eval(isnan(f_eval)) = 0; % reset NaN vlaues to zero
end

function MVNPfunvalfinal = MVNPexact(t,b,sig)
  % MVNPexact calculates the true solution of multivariate normal probability
  % when the covariance matrix is in a special form: diagonal is 1 and off
  % diagonal elements are all the same.
  %
  % b   - the upper limits of the integral with size 1 x d
  % sig - the off diagonal element
  % dim - the dimension of the integral
  % t   - the variable
  MVNPfunval = (gail.stdnormcdf((b(1)+sqrt(sig)*t)/sqrt(1-sig)));
  dim =  length(b);
  for i =2:dim
    MVNPfunval= MVNPfunval.*(gail.stdnormcdf((b(i)+sqrt(sig)*t)/sqrt(1-sig)));
    %i=i+100;
  end
  MVNPfunvalfinal = MVNPfunval.*exp(-t.^2/2);
end

function report_integration_result(testId,algo,abstol,reltol,exactsol,approxsol,timeSec,nSample)
  fprintf('%s: %s\n', testId,algo)
  fprintf('  Estimated probability: %f \n', approxsol)
  if ~isnan(exactsol)
    fprintf('       True probability: %f \n', exactsol)
  end

  fprintf('  The algorithm took %1.3f seconds and %d points \n', timeSec,nSample)

  if ~isnan(exactsol)
    errTol = gail.tolfun(abstol,reltol,1,exactsol,'max');
    errReal = abs(exactsol-approxsol);
    if errReal > errTol
      ME = MException('cubBayesLattice_g_demo:errorExceeded', ...
        'Real error %1.2e exceeds given tolerance %1.2e',errReal,errTol);
      throw(ME)
    else
      fprintf('  Real error is %1.3e, which is less than the tolerance %1.3e\n',...
        errReal, errTol)
    end
  end
end
```

Acknowledgements

We thank the contributions of Aleksei Sorokin, Noah Grudowski, Francisco Hernandez, Cu Hauw Hung, Yueyi Li, Xincheng Sheng, Xiaoyang Zhao, Tianci Zhu, and the IIT classes of SCI 498 Adaptive Monte Carlo Algorithms with Applications to Financial Risk Management, Summer 2016; MATH 491 Reading & Research, Summer 2015; SCI 498/MATH 491 Computational Social Sciences, Summer 2016; MATH 491-195 Solving Problems in the Social Sciences Using Tools from Computational Mathematics and Statistics, Summer 2015; Math 573 Reliable Mathematical Software, Fall 2013.

References

[1] Richard P. Brent. *Algorithms for Minimization Without Derivatives.* Prentice-Hall, 1973.

[2] Jonathan B. Buckheit and David L. Donoho. *Wavelab and reproducible research.* Springer, 1995.

[3] Marek Capiński and Ekkehard Kopp. *Measure, Integral and Probability.* Springer-Verlag, 1999.

[4] Sou-Cheng Choi, David L. Donoho, Ana Georgina Flesia, Xiaoming Huo, Ofer Levi, and Danzhu Shi. About Beamlab—a toolbox for new multiscale methodologies. Technical report, Stanford University, 2002.

[5] Sou-Cheng T. Choi. MINRES-QLP Pack and reliable reproducible research via supportable scientific software. *Journal of Open Research Software*, 2(1), 2014.

[6] Sou-Cheng T Choi, Yuhan Ding, Fred J Hickernell, and Xin Tong. Local adaption for approximation and minimization of univariate functions. *Journal of Complexity*, 40:17–33, 2017.

[7] Sou-Cheng T. Choi and Fred J. Hickernell. IIT MATH-573 Reliable Mathematical Software, 2013.

[8] Sou-Cheng T. Choi, Fred J. Hickernell, Yuhan Ding, Lan Jiang, Lluís Antoni Jiménez Rugama, Da Li, Jagadeeswaran Rathinavel, Xin Tong, Kan Zhang, Yizhi Zhang, and Xuan Zhou. GAIL: Guaranteed Automatic Integration Library (Version 2.3.1), MATLAB Software, 2020.

[9] Sou-Cheng T. Choi, Fred J. Hickernell, Yuhan Ding, Lan Jiang, Lluís Antoni Jiménez Rugama, Xin Tong, Yizhi Zhang, and Xuan Zhou. GAIL: Guaranteed Automatic Integration Library (Version 2.1), MATLAB Software, 2015.

[10] Sou-Cheng T. Choi, Fred J. Hickernell, Jagadeeswaran Rathinavel, Michael McCourt, and Aleksei Sorokin. QMCPy: A quasi-Monte Carlo Python library. `https://qmcsoftware.github.io/QMCSoftware/`, 2020. Working.

[11] Jon Claerbout. Reproducible Computational Research: A history of hurdles, mostly overcome.

[12] Nicholas Clancy, Yuhan Ding, Caleb Hamilton, Fred J. Hickernell, and Yizhi Zhang. The cost of deterministic, adaptive, automatic algorithms: Cones, not balls. *Journal of Complexity*, 30:21–45, 2014.

[13] Yuhan Ding. *Guaranteed adaptive univariate function approximation.* PhD thesis, Illinois Institute of Technology, 2015.

[14] Yuhan Ding, Fred J. Hickernell, and Lluís Antoni Jiménez Rugama. An adaptive algorithm employing continuous linear functionals. In Bruno Tuffin and Pierre L'Ecuyer, editors, *Monte Carlo and Quasi-Monte Carlo Methods*, pages 161–181, Cham, 2020. Springer International Publishing.

[15] George E. Forsythe, Michael A. Malcolm, and Cleve B. Moler. *Computer methods for mathematical computations.* Prentice-Hall, 1976.

[16] Nicholas Hale, Lloyd N. Trefethen, and Tobin A. Driscoll. *Chebfun Version 5.7*, 2017.

[17] Fred J. Hickernell, Lan Jiang, Yuewei Liu, and Art B. Owen. Guaranteed conservative fixed width confidence intervals via Monte Carlo sampling. In J. Dick, F. Y. Kuo, G. W. Peters, and I. H. Sloan, editors, *Monte Carlo and Quasi-Monte Carlo methods 2012*, pages 105–128. Springer-Verlag, Berlin, 2013.

[18] Fred J. Hickernell, Lluís Antoni Jiménez Rugama, and D. Li. Adaptive quasi-Monte Carlo methods for cubature. In J. Dick, F. Y. Kuo, and H. Woźniakowski, editors, *Contemporary Computational Mathematics — a celebration of the 80th birthday of Ian Sloan*, pages 597–619. Springer-Verlag, 2018.

[19] Lan Jiang. *Guaranteed Adaptive Monte Carlo Methods for Estimating Means of Random Variables*. PhD thesis, Illinois Institute of Technology, 2016.

[20] Daniel Katz, Sou-Cheng Choi, Hilmar Lapp, Ketan Maheshwari, Frank Löffler, Matthew Turk, Marcus Hanwell, Nancy Wilkins-Diehr, James Hetherington, James Howison, Shel Swenson, Gabrielle Allen, Anne Elster, Bruce Berriman, and Colin Venters. Summary of the first workshop on sustainable software for science: Practice and experiences (WSSSPE1). *Journal of Open Research Software*, 2(1), 2014.

[21] Daniel S. Katz, Sou-Cheng T. Choi, Kyle E. Niemeyer, James Hetherington, Frank Löffler, Dan Gunter, Ray Idaszak, Steven R. Brandt, Mark A. Miller, Sandra Gesing, Nick D. Jones, Nic Weber, Suresh Marru, Gabrielle Allen, Birgit Penzenstadler, Colin C. Venters, Ethan Davis, Lorraine Hwang, Ilian Todorov, Abani Patra, and Miguel de Val-Borro. Report on the third workshop on sustainable software for science: Practice and experiences (WSSSPE3). *ournal of Open Research Software*, 4(1), 2016.

[22] Daniel S. Katz, Sou-Cheng T. Choi, Nancy Wilkins-Diehr, Neil Chue Hong, Colin C. Venters, James Howison, Frank J. Seinstra, Matthew Jones, Karen Cranston, Thomas L. Clune, Miguel de Val-Borro, and Richard Littauer. Report on the second workshop on sustainable software for science: Practice and experiences (WSSSPE2). *Journal of Open Research Software*, 4(1), 2016.

[23] Da Li. Reliable quasi-Monte Carlo with control variates. Master's thesis, Illinois Institute of Technology, 2016.

[24] Jagadeeswaran Rathinavel. *Fast Automatic Bayesian Cubature Using Matching Kernels and Designs*. PhD thesis, Illinois Institute of Technology, 2019.

[25] Jagadeeswaran Rathinavel and Fred J. Hickernell. Fast automatic Bayesian cubature using lattice sampling. *Stat. Comput.*, 29:1215–1229, 2019.

[26] Lluís Antoni Jiménez Rugama and Fred J. Hickernell. Adaptive multidimensional integration based on rank-1 lattices. In *Monte Carlo and Quasi-Monte Carlo Methods*, pages 407–422. Springer, 2016.

[27] Arfon M Smith, Daniel S Katz, Kyle E Niemeyer, and FORCE11 Software Citation Working Group. Software citation principles. *PeerJ Computer Science*, 2:e86, 2016.

[28] Xin Tong. A guaranteed, adaptive, automatic algorithm for univariate function minimization. Master's thesis, Illinois Institute of Technology, 2014.

[29] Yizhi Zhang. *Guaranteed Adaptive Automatic Algorithms for Univariate Integration: Methods, Costs and Implementations*. PhD thesis, Illinois Institute of Technology, 2018.