QMCPy

Release 1.4.5

Sou-Cheng T. Choi **Fred J. Hickernell**
Michael McCourt **Jagadeeswaran Rathinavel**
Aleksei Sorokin

Nov 07, 2023

CONTENTS

CHAPTER

ONE

ABOUT OUR QMC SOFTWARE COMMUNITY

1.1 Quasi-Monte Carlo Community Software

Quasi-Monte Carlo (QMC) methods are used to approximate multivariate integrals. They have four main components: an integrand, a discrete distribution, summary output data, and stopping criterion. Information about the integrand is obtained as a sequence of values of the function sampled at the data-sites of the discrete distribution. The stopping criterion tells the algorithm when the user-specified error tolerance has been satisfied. We are developing a framework that allows collaborators in the QMC community to develop plug-and-play modules in an effort to produce more efficient and portable QMC software. Each of the above four components is an abstract class. Abstract classes specify the common properties and methods of all subclasses. The ways in which the four kinds of classes interact with each other are also specified. Subclasses then flesh out different integrands, sampling schemes, and stopping criteria. Besides providing developers a way to link their new ideas with those implemented by the rest of the QMC community, we also aim to provide practitioners with state-of-the-art QMC software for their applications.

Homepage ~ Article ~ GitHub ~ Read the Docs ~ PyPI ~ Blogs ~ DockerHub ~ Contributing ~ Issues

---

1.1.1 Installation

```
pip install qmcpy
```

1.1.2 The QMCPy Framework

The central package including the 5 main components as listed below. Each component is implemented as abstract classes with concrete implementations. For example, the lattice and Sobol' sequences are implemented as concrete implementations of the `DiscreteDistribution` abstract class. A complete list of concrete implementations and thorough documentation can be found in the QMCPy Read the Docs.

- **Stopping Criterion:** determines the number of samples necessary to meet an error tolerance.
- **Integrand:** the function/process whose expected value will be approximated.
- **True Measure:** the distribution to be integrated over.
- **Discrete Distribution:** a generator of nodes/sequences that can be either IID (for Monte Carlo) or low-discrepancy (for quasi-Monte Carlo), that mimic a standard distribution.
- **Accumulate Data:** stores and updates data used in the integration process.

1.1.3 Quickstart

Note: If the following mathematics is not rendering try using Google Chrome and installing the Mathjax Plugin for GitHub.

We will approximate the expected value of the d dimensional Keister integrand [18]

$$g(X) = \pi^{d/2} \cos(\|X\|)$$

where $X \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I}/2)$.

We may choose a Sobol' discrete distribution with a corresponding Sobol' cubature stopping criterion to preform quasi-Monte Carlo integration.

```
import qmcpy as qp
from numpy import pi, cos, sqrt, linalg
d = 2
dnb2 = qp.DigitalNetB2(d)
gauss_sobol = qp.Gaussian(dnb2, mean=0, covariance=1/2)
k = qp.CustomFun(
  true_measure = gauss_sobol,
  g = lambda x: pi**(d/2)*cos(linalg.norm(x,axis=1)))
qmc_sobol_algorithm = qp.CubQMCSobolG(k, abs_tol=1e-3)
solution,data = qmc_sobol_algorithm.integrate()
print(data)
```

Running the above code outputs

```
LDTransformData (AccumulateData Object)
    solution        1.808
    error_bound     4.68e-04
    n_total         2^(13)
    time_integrate  0.008
CubQMCSobolG (StoppingCriterion Object)
    abs_tol         0.001
    rel_tol         0
    n_init          2^(10)
    n_max           2^(35)
CustomFun (Integrand Object)
Gaussian (TrueMeasure Object)
    mean            0
    covariance      2^(-1)
    decomp_type     PCA
Sobol (DiscreteDistribution Object)
    d               2^(1)
    dvec            [0 1]
    randomize       LMS_DS
    graycode        0
    entropy         127071403717453177593768120720330942628
    spawn_key       ()
```

A more detailed quickstart can be found in our GitHub repo at `QMCSoftware/demos/quickstart.ipynb` or in this Google Colab quickstart notebook.

We also highly recommend you take a look at Fred Hickernell's tutorial at the Monte Carlo Quasi-Monte Carlo 2020 Conference and the corresponding MCQMC2020 Google Colab notebook.

1.1.4 Community

Please refer to this document for the key roles in the QMCPy community.

1.1.5 Citation

If you find QMCPy helpful in your work, please support us by citing the following work:

```
Choi, S.-C. T., Hickernell, F. J., McCourt, M., Rathinavel, J. & Sorokin, A.
QMCPy: A quasi-Monte Carlo Python Library. Working. 2020.
https://qmcsoftware.github.io/QMCSoftware/
```

BibTex citation available here

V id eo Tu to ri al Pl ea se r ef er to ` th is vi de o <h tt ps :/ /w ww .y ou tu be .c om /w at ch ?v =b Rc Ki LA 2y BQ >` __ f or a q ui ck in tr od uc ti on to QM CP y.
F or a mo re de ta il in tr od uc ti on r ef er to ` th is v id eo  ht tp s: // ww w. yo ut ub e. co m/ wa tc h? v= gL 8M _7 c- YU E> `_ _.
Re fe re nc es
[1] F. Y. K uo a nd D. N uy en s. “A pp li ca ti on of q ua si -M on te C ar lo m et ho ds to el li pt ic PD Es wi th ra nd om d if fu si on co ef fi ci en ts - a su rv ey of an al ys is a nd im pl em en ta ti on ,” F ou nd at io ns of C om pu ta ti on al Ma th em at ic s, 16 (6): 16 31 -1 69 6, 2 01 6. (` sp ri ng er li nk  ht tp s: // li nk .s pr in ge r. co m/ ar ti cl e/ 10 .1 00 7/ s1 02 08 -0 16 -9 32 9- 5> `_ _, `a rx iv l in k <h tt ps :/ /a rx iv .o rg /a bs /1 60 6. 06 61 3> `_ _)
[2] Fr ed J. H ic ke rn el l, L an Ji an g, Yu ew ei Li u, a nd A rt B. O we n, “ Gu ar an te ed co ns er va ti ve f ix ed w id th co nf id en ce i nt er va ls v ia M on te C ar lo sa mp li ng ,” M on te C ar lo a nd Q ua si -M on te C ar lo M et ho ds 20 12 (J. D ic k, F. Y. Ku o, G. W. P et er s, a nd I. H. Sl oa n, ed s.), pp . 10 5- 12 8, Sp ri ng er -V er la g, B er li n, 2 01 4. DO I: 1 0. 10 07 /9 78 -3 -6 42 -4 10 95 -6 _5
[3] S ou -C he ng T. C ho i, Y uh an D in g, Fr ed J. H ic ke rn el l, L an Ji an g, L lu is An to ni J im en ez R ug am a, Da L i, J ag ad ee sw ar an R at hi na ve l, X in T on g, K an Zh an g, Y iz hi Zh an g, a nd Xu an Z ho u, G AI L: Gu ar an te ed A ut om at ic I nt eg ra ti on L ib ra ry (V er si on 2. 3. 1) [MA TL AB So ft wa re], 2 02 0. A va il ab le fr om ht tp :/ /g ai lg it hu b. gi th ub .i o/ GA IL _D ev /.
[4] S ou -C he ng T. C ho i, “ MI NR ES -Q LP Pa ck a nd Re li ab le Re pr od uc ib le Re se ar ch v ia S up po rt ab le Sc ie nt if ic So ft wa re ,” J ou rn al of Op en Re se ar ch S of tw ar e, Vo lu me 2, Nu mb er 1, e2 2, pp . 1- 7, 2 01 4.
[5] S ou -C he ng T. Ch oi a nd Fr ed J. H ic ke rn el l, “I IT MA TH -5 73 Re li ab le Ma th em at ic al S of tw ar e” [Co ur se Sl id es], Il li no is I ns ti tu te of T ec hn ol og y, Ch ic ag o, I L, 2 01 3. A va il ab le fr om ht tp :/ /g ai lg it hu b. gi th ub .i o/ GA IL _D ev /.
[6] Da ni el S. K at z, S ou -C he ng T. C ho i, Hi lm ar L ap p, K et an M ah es hw ar i, F ra nk Lo ff le r, M at th ew T ur k, Ma rc us D. Ha nw el l, N an cy Wi lk in s- Di eh r, J am es H et he ri ng to n, J am es Ho wi so n, Sh el Sw en so n, G ab ri el le D. Al le n, An ne C. E ls te r, B ru ce B er ri ma n, C ol in Ve nt er s, “S um ma ry of t he F ir st Wo rk sh op On S us ta in ab le So ft wa re f or Sc ie nc e: Pr ac ti ce a nd E xp er ie nc es (WS SS PE 1) ,” J ou rn al of Op en Re se ar ch S of tw ar e, Vo lu me 2, Nu mb er 1, e 6, p p. 1 -2 1, 2 01 4.
[7] F an g, K. -T ., a nd W an g, Y. (19 94). Nu mb er -t he or et ic M et ho ds in S ta ti st ic s. L on do n, U K: C HA PM AN & HA LL

continues on next page

Table 1 – continued from previous page

[8] L an Ji an g, Gu ar an te ed Ad ap ti ve M on te C ar lo M et ho ds f or Es ti ma ti ng M ea ns of Ra nd om Va ri ab le s, P hD T he si s, Il li no is I ns ti tu te of T ec hn ol og y, 2 01 6.
[9] L lu is An to ni J im en ez Ru ga ma a nd Fr ed J. H ic ke rn el l, " Ad ap ti ve mu lt id im en si on al i nt eg ra ti on b as ed on ra nk -1 la tt ic es ," M on te C ar lo a nd Q ua si -M on te C ar lo Me th od s: MC QM C, L eu ve n, Be lg iu m, A pr il 20 14 (R. C oo ls a nd D. N uy en s, ed s.), Sp ri ng er P ro ce ed in gs in M at he ma ti cs a nd S ta ti st ic s, v ol . 16 3, Sp ri ng er -V er la g, B er li n, 2 01 6, ar Xi v: 14 11 .1 96 6, pp . 40 7- 42 2.
[1 0] K ai -T ai Fa ng a nd Yu an W an g, Nu mb er -t he or et ic M et ho ds in S ta ti st ic s, C ha pm an & H al l, L on do n, 1 99 4.
[1 1] Fr ed J. Hi ck er ne ll a nd L lu is An to ni J im en ez R ug am a, " Re li ab le ad ap ti ve cu ba tu re u si ng d ig it al s eq ue nc es ," M on te C ar lo a nd Q ua si -M on te C ar lo Me th od s: MC QM C, L eu ve n, Be lg iu m, A pr il 20 14 (R. C oo ls a nd D. N uy en s, ed s.), Sp ri ng er P ro ce ed in gs in M at he ma ti cs a nd S ta ti st ic s, v ol . 16 3, Sp ri ng er -V er la g, B er li n, 2 01 6, a rX iv :1 41 0. 86 15 [m at h. NA], pp . 36 7- 38 3.
[1 2] Ma ri us Ho fe rt a nd Ch ri st ia ne L em ie ux (20 19). q rn g: (R an do mi ze d) Qu as i- Ra nd om Nu mb er G en er at or s. R p ac ka ge v er si on 0. 0- 7. ht tp s: // CR AN .R -p ro je ct .o rg /p ac ka ge =q rn g.
[1 3] Fa ur e, He nr i, a nd Ch ri st ia ne Le mi eu x. " Im pl em en ta ti on of I rr ed uc ib le So bo l' S eq ue nc es in P ri me P ow er B as es ," M at he ma ti cs a nd C om pu te rs in Si mu la ti on 1 61 (20 19): 13 –2 2.
[1 4] M. B. Gi le s. "M ul ti -l ev el M on te C ar lo pa th si mu la ti on ," Op er at io ns R es ea rc h, 56 (3): 60 7- 61 7, 2 00 8. h tt p: // pe op le .m at hs .o x. ac .u k/ ~g il es m/ fi le s/ OP RE _2 00 8. pd f.
[1 5] M. B. Gi le s. " Im pr ov ed mu lt il ev el M on te C ar lo c on ve rg en ce u si ng t he Mi ls te in sc he me ," 34 3- 35 8, in M on te C ar lo a nd Q ua si -M on te C ar lo M et ho ds 2 00 6, S pr in ge r, 2 00 8. h tt p: // pe op le .m at hs .o x. ac .u k/ ~g il es m/ fi le s/ mc qm c0 6. pd f.
[1 6] M. B. G il es a nd B. J. W at er ho us e. " Mu lt il ev el q ua si -M on te C ar lo pa th si mu la ti on ," pp .1 65 -1 81 in Ad va nc ed F in an ci al Mo de ll in g, in R ad on Se ri es on C om pu ta ti on al a nd A pp li ed Ma th em at ic s, de Gr uy te r, 2 00 9. h tt p: // pe op le .m at hs .o x. ac .u k/ ~g il es m/ fi le s/ ra do n. pd f.
[1 7] O we n, A. B. "A ra nd om iz ed Ha lt on a lg or it hm in R ," 2 01 7. ar Xi v: 17 06 .0 28 08 [st at .C O]
[1 8] B. D. Ke is te r, Mu lt id im en si on al Qu ad ra tu re A lg or it hm s, 'C om pu te rs in P hy si cs ', 1 0, pp . 11 9- 12 2, 1 99 6.
[1 9] L 'E cu ye r, Pi er re & M un ge r, Da vi d. (20 15). L at ti ce Bu il de r: A G en er al So ft wa re To ol f or Co ns tr uc ti ng Ra nk -1 L at ti ce Ru le s. A CM Tr an sa ct io ns on Ma th em at ic al S of tw ar e. 4 2. 10 .1 14 5/ 27 54 92 9.
[2 0] Fi sc he r, G re go ry & C ar mo n, Z iv & Za ub er ma n, G al & L 'E cu ye r, P ie rr e. (19 99). Go od Pa ra me te rs a nd I mp le me nt at io ns f or Co mb in ed Mu lt ip le R ec ur si ve Ra nd om Nu mb er G en er at or s. Op er at io ns R es ea rc h. 4 7. 15 9- 16 4. 10 .1 28 7/ op re .4 7. 1. 15 9.
[2 1] I. M. S ob ol ', V. I. Tu rc ha ni no v, Y u. L. Le vi ta n, B. V. S hu kh ma n: " Qu as i- Ra nd om Se qu en ce G en er at or s" K el dy sh I ns ti tu te of A pp li ed Ma th em at ic s, R us si an A ca md ey of S ci en ce s, Mo sc ow (19 92).
[2 2] So bo l, Il ya & As ot sk y, D an il & Kr ei ni n, A le xa nd er & K uc he re nk o, S er ge i. (20 11). Co ns tr uc ti on a nd Co mp ar is on of Hi gh -D im en si on al So bo l' G en er at or s. Wi lm ot t. 2 01 1. 1 0. 10 02 /w il m. 10 05 6.
[2 3] P as zk e, A ., Gr os s, S ., Ma ss a, F ., Le re r, A ., B ra db ur y, J ., C ha na n, G ., … C hi nt al a, S. (20 19). Py To rc h: An Im pe ra ti ve St yl e, Hi gh -P er fo rm an ce De ep Le ar ni ng Li br ar y. In H. Wa ll ac h, H. L ar oc he ll e, A. Be yg el zi me r, F. d ex tq uo te si ng le A lc h' e- Bu c, E. Fo x, & R. G ar ne tt (Ed s.), Ad va nc es in Ne ur al I nf or ma ti on Pr oc es si ng S ys te ms 32 (p p. 8 02 4– 80 35). Cu rr an A ss oc ia te s, In c. R et ri ev ed fr om ht tp :/ /p ap er s. ne ur ip s. cc /p ap er /9 01 5- py to rc h- an -i mp er at iv e- st yl e- hi gh -p er fo rm an ce -d ee p- le ar ni ng -l ib ra ry .p df
[2 4] S. J oe a nd F. Y. Ku o, Co ns tr uc ti ng S ob ol s eq ue nc es wi th be tt er t wo -d im en si on al pr oj ec ti on s, SI AM J. Sc i. C om pu t. 3 0, 2 63 5- 26 54 (20 08).
[2 5] Pa ul B ra tl ey a nd B en ne tt L. Fo x. 1 98 8. A lg or it hm 65 9: Im pl em en ti ng S ob ol 's q ua si ra nd om se qu en ce ge ne ra to r. A CM Tr an s. M at h. So ft w. 1 4, 1 (M ar ch 19 88), 8 8– 10 0. DO I: ht tp s: // do i. or g/ 10 .1 14 5/ 42 28 8. 21 43 72

continues on next page

Table 1 – continued from previous page

[2 6] P. L ’E cu ye r, P. M ar io n, M. Go di n, a nd F. P uc hh am me r, “A To ol f or Cu st om Co ns tr uc ti on of Q MC a nd RQ MC P oi nt Se ts ,” M on te C ar lo a nd Q ua si -M on te C ar lo M et ho ds 2 02 0.
[2 7] P Ku ma ra sw am y, A g en er al iz ed p ro ba bi li ty d en si ty fu nc ti on f or do ub le -b ou nd ed ra nd om pr oc es se s. J. H yd ro l. 4 6, 7 9– 88 (19 80).
[2 8] D L i, Re li ab le q ua si -M on te C ar lo wi th c on tr ol v ar ia te s. Ma st er ’s t he si s, Il li no is I ns ti tu te of Te ch no lo gy (2 01 6)
[2 9] D. H. B ai le y, J. M. Bo rw ei n, R. E. C ra nd al l, B ox in te gr al s, J ou rn al of C om pu ta ti on al a nd A pp li ed Ma th em at ic s, Vo lu me 20 6, I ss ue 1, 2 00 7, P ag es 19 6- 20 8, IS SN 03 77 -0 42 7, ht tp s: // do i. or g/ 10 .1 01 6/ j. ca m. 20 06 .0 6. 01 0.
[3 0] A rt B. Ow en .M on te C ar lo t he or y, m et ho ds a nd e xa mp le s. 2 01 3.

1.1.6 Sponsors

- Illinois Tech
- Kamakura Corporation**, acquired by** SAS Institute Inc. **in June 2022**
- SigOpt, Inc.

CHAPTER

TWO

CONTRIBUTING TO QMCSOFTWARE

2.1 For Contributors

Thank you for your interest in contributing to the QMCPy package!

Please submit **pull requests** to the `develop` branch and **issues** using a template from `.github/ISSUE_TEMPLATE/`

If you develop a new component please consider writing a blog for qmcpy.org

Join team communications by reaching out to us at qmc-software@googlegroups.com

2.1.1 Installation

In a git enabled terminal (e.g. bash for Windows) with miniconda installed and C compilers enabled (Windows users may need to install Microsoft C++ Build Tools), run

```
git clone https://github.com/QMCSoftware/QMCSoftware.git
cd QMCSoftware
git checkout develop
conda env create --file environment.yml
conda activate qmcpy
pip install -e .
```

Doctests and unittests take a few minute to run with

```
make tests_no_docker
```

Optionally, you may install Docker and then run all tests with

```
make tests
```

After installing Pandoc and LaTeX, documentation may be compiled into your preferred format with

```
doc_html
doc_pdf
doc_epub
```

Demos may be run by opening Jupyter using

```
jupyter notebook
```

and then navigating to the desired file in the `demo/` directory. Some of our developers prefer JupyterLab, which may be installed with

```
pip install jupyterlab
```

and then run with

```
jupyter-lab
```

The Developers Tools page on qmcpy.org documents additional tools we have found helpful for mathematical software development and presentation.

2.1.2 Visual Studio Code Tips and Tricks

VSCode is the IDE of choice for many of our developers. Here we compile some helpful notes regarding additional setup for VSCode.

- Run `CMD+p` then `> Python: Select Interpreter` then select the `('qmcpy')` choice from the dropdown to link the qmcpy environment into your workspace. Now when you open a terminal, your command line should read `(qmcpy) username@...` which indicates the qmcpy environment has been automatically activated. Also, when debugging the qmcpy environment will be automatically used.
- Go to `File` and click `Save Workspace as...` to save a qmcpy workspace for future development.

Useful Extensions

- Python
- Jupyter
- Markdown Preview Enhanced
- eps-preview, which requires
  - Postscript Language
  - pdf2svg
- Git Graph
- Code Spell Checker

CHAPTER

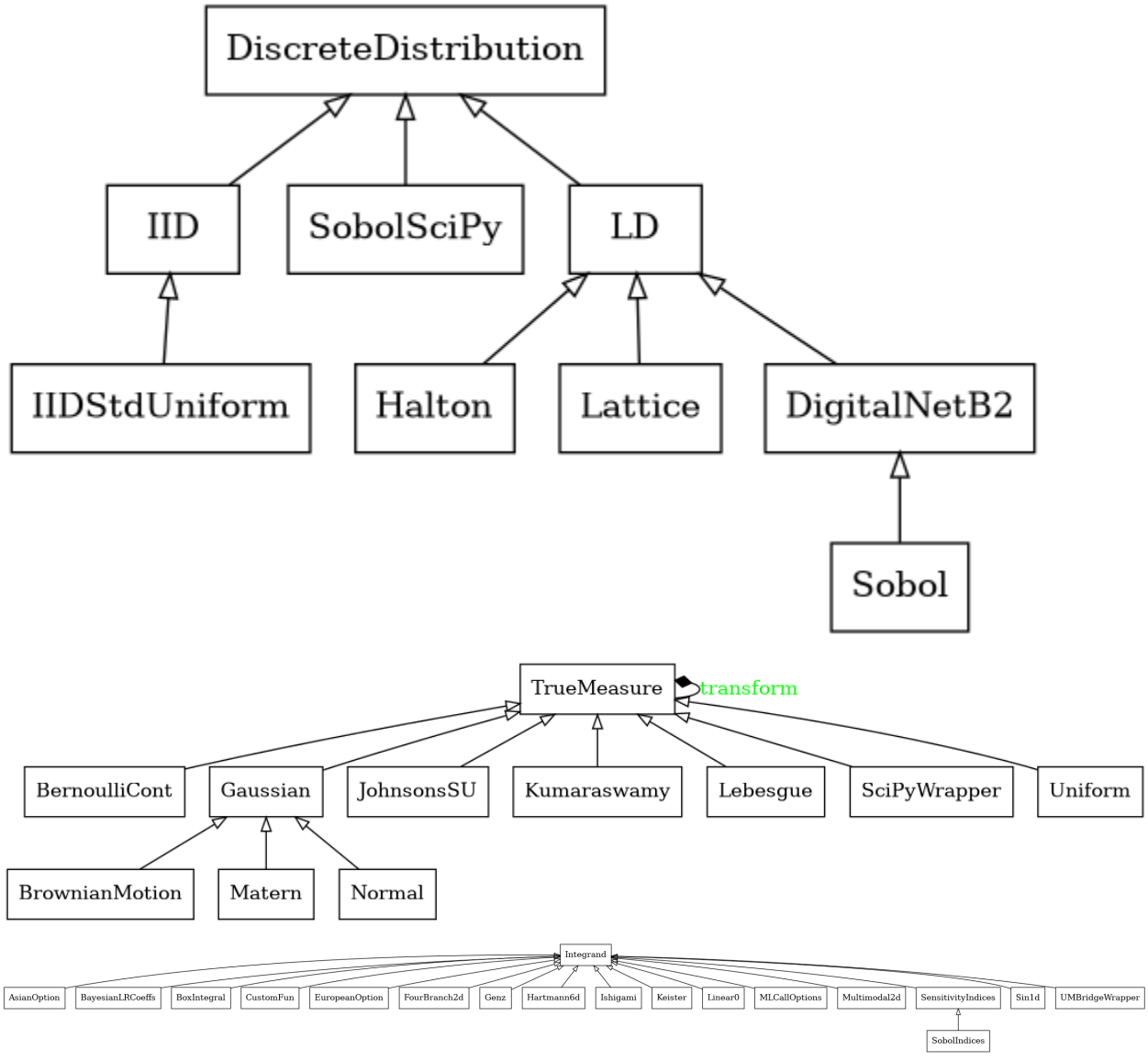
THREE

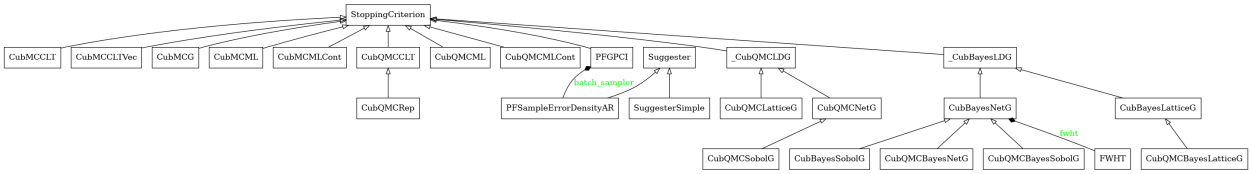
LICENSE

CHAPTER

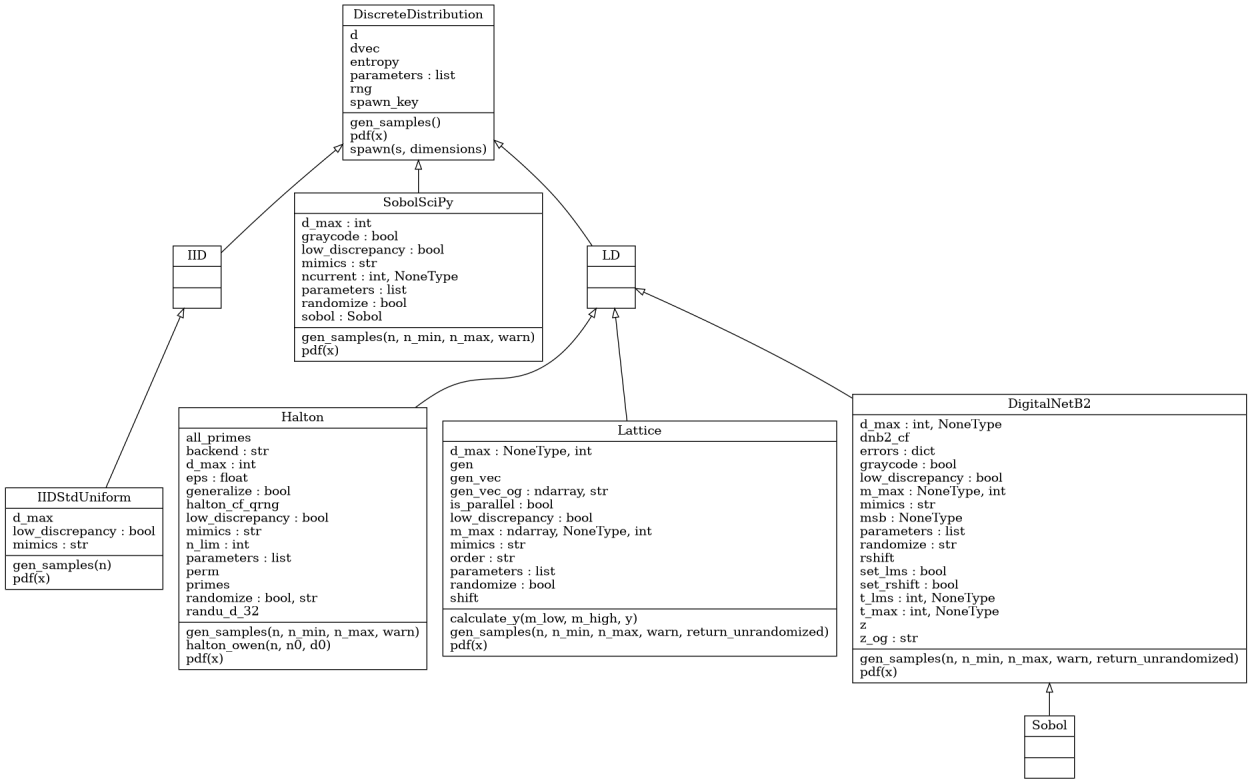
FOUR

QMCPY DOCUMENTATION





4.1 Discrete Distribution Class



4.1.1 Abstract Discrete Distribution Class

class qmcpy.discrete_distribution._discrete_distribution.**DiscreteDistribution**(*dimension*, *seed*)

Discrete Distribution abstract class. DO NOT INSTANTIATE.

__init__(*dimension*, *seed*)

Parameters

- **dimension** (*int or ndarray*) – dimension of the generator. If an int is passed in, use sequence dimensions [0,...,dimensions-1]. If a ndarray is passed in, use these dimension indices in the sequence. Note that this is not relevant for IID generators.
- **seed** (*int or numpy.random.SeedSequence*) – seed to create random number generator

gen_samples(**args*)

ABSTRACT METHOD to generate samples from this discrete distribution.

Parameters
args (*tuple*) – tuple of positional argument. See implementations for details

Returns
n x d array of samples

Return type
ndarray

pdf(*x*)
ABSTRACT METHOD to evaluate pdf of distribution the samples mimic at locations of x.

spawn(*s=1*, *dimensions=None*)
Spawn new instances of the current discrete distribution but with new seeds and dimensions. Developed for multi-level and multi-replication (Q)MC algorithms.

Parameters
- **s** (*int*) – number of spawn
- **dimensions** (*ndarray*) – length s array of dimension for each spawn. Defaults to current dimension

Returns
list of DiscreteDistribution instances with new seeds and dimensions

Return type
list

class qmcpy.discrete_distribution._discrete_distribution.**IID**(*dimension*, *seed*)

class qmcpy.discrete_distribution._discrete_distribution.**LD**(*dimension*, *seed*)

4.1.2 Digital Net Base 2

class qmcpy.discrete_distribution.digital_net_b2.digital_net_b2.**DigitalNetB2**(*dimension=1*, *randomize='LMS_DS'*, *graycode=False*, *seed=None*, *generating_matrices='sobol_mat.21201.3*, *d_max=None*, *t_max=None*, *m_max=None*, *msb=None*, *t_lms=None*, *_verbose=False*)

Quasi-Random digital nets in base 2.

```
>>> dnb2 = DigitalNetB2(2,seed=7)
>>> dnb2.gen_samples(4)
array([[0.56269008, 0.17377997],
       [0.346653  , 0.65070632],
       [0.82074548, 0.95490574],
```

(continues on next page)

(continued from previous page)

```
       [0.10422261, 0.49458097]])
>>> dnb2.gen_samples(1)
array([[0.56269008, 0.17377997]])
>>> dnb2
DigitalNetB2 (DiscreteDistribution Object)
    d               2^(1)
    dvec            [0 1]
    randomize       LMS_DS
    graycode        0
    entropy         7
    spawn_key       ()
>>> DigitalNetB2(dimension=2,randomize=False,graycode=True).gen_samples(n_min=2,n_
↪max=4)
array([[0.75, 0.25],
       [0.25, 0.75]])
>>> DigitalNetB2(dimension=2,randomize=False,graycode=False).gen_samples(n_min=2,n_
↪max=4)
array([[0.25, 0.75],
       [0.75, 0.25]])
>>> dnb2_alpha2 = DigitalNetB2(5,randomize=False,generating_matrices='sobol_mat_
↪alpha2.10600.64.32.lsb.npy')
>>> dnb2_alpha2.gen_samples(8,warn=False)
array([[0.       , 0.       , 0.       , 0.       , 0.       ],
       [0.75     , 0.75     , 0.75     , 0.75     , 0.75     ],
       [0.4375   , 0.9375   , 0.1875   , 0.6875   , 0.1875   ],
       [0.6875   , 0.1875   , 0.9375   , 0.4375   , 0.9375   ],
       [0.296875, 0.171875, 0.109375, 0.796875, 0.859375],
       [0.546875, 0.921875, 0.859375, 0.046875, 0.109375],
       [0.234375, 0.859375, 0.171875, 0.484375, 0.921875],
       [0.984375, 0.109375, 0.921875, 0.734375, 0.171875]])
```

References

[1] Marius Hofert and Christiane Lemieux (2019). qrng: (Randomized) Quasi-Random Number Generators. R package version 0.0-7. https://CRAN.R-project.org/package=qrng.

[2] Faure, Henri, and Christiane Lemieux. "Implementation of Irreducible Sobol' Sequences in Prime Power Bases." Mathematics and Computers in Simulation 161 (2019): 13-22. Crossref. Web.

[3] F.Y. Kuo & D. Nuyens. Application of quasi-Monte Carlo methods to elliptic PDEs with random diffusion coefficients - a survey of analysis and implementation, Foundations of Computational Mathematics, 16(6):1631-1696, 2016. springer link: https://link.springer.com/article/10.1007/s10208-016-9329-5 arxiv link: https://arxiv.org/abs/1606.06613

[4] D. Nuyens, *The Magic Point Shop of QMC point generators and generating vectors.* MATLAB and Python software, 2018. Available from https://people.cs.kuleuven.be/~dirk.nuyens/ https://bitbucket.org/dnuyens/qmc-generators/src/cb0f2fb10fa9c9f2665e41419097781b611daa1e/cpp/digitalseq_b2g.hpp

[5] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., … Chintala, S. (2019). PyTorch: An Imperative Style, High-Performance Deep Learning Library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d extquotesingle Alch'e-Buc, E. Fox, & R. Garnett (Eds.), Advances in Neural Information Processing Systems 32 (pp. 8024-8035). Curran Associates, Inc. Retrieved from http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[6] I.M. Sobol', V.I. Turchaninov, Yu.L. Levitan, B.V. Shukhman: "Quasi-Random Sequence Generators" Keldysh Institute of Applied Mathematics, Russian Acamdey of Sciences, Moscow (1992).

[7] Sobol, Ilya & Asotsky, Danil & Kreinin, Alexander & Kucherenko, Sergei. (2011). Construction and Comparison of High-Dimensional Sobol' Generators. Wilmott. 2011. 10.1002/wilm.10056.

[8] Paul Bratley and Bennett L. Fox. 1988. Algorithm 659: Implementing Sobol's quasirandom sequence generator. ACM Trans. Math. Softw. 14, 1 (March 1988), 88-100. DOI:https://doi.org/10.1145/42288.214372

__init__(*dimension=1*, *randomize='LMS_DS'*, *graycode=False*, *seed=None*, *generating_matrices='sobol_mat.21201.32.32.msb.npy'*, *d_max=None*, *t_max=None*, *m_max=None*, *msb=None*, *t_lms=None*, *_verbose=False*)

Parameters

- **dimension** (*int or ndarray*) – dimension of the generator. If an int is passed in, use sequence dimensions [0,...,dimensions-1]. If a ndarray is passed in, use these dimension indices in the sequence.
- **randomize** (*bool*) – apply randomization? True defaults to LMS_DS. Can also explicitly pass in 'LMS_DS': linear matrix scramble with digital shift 'LMS': linear matrix scramble only 'DS': digital shift only
- **graycode** (*bool*) – indicator to use graycode ordering (True) or natural ordering (False)
- **seed** (*list*) – int seed of list of seeds, one for each dimension.
- **generating_matrices** (*ndarray or str*) – generating matrices or path to generating matrices. ndarray should have shape (d_max, m_max) where each int has t_max bits generating_matrices should be formatted like *gen_mat.21201.32.32.msb.npy* with name.d_max.t_max.m_max.{msb,lsb}.npy
- **d_max** (*int*) – max dimension
- **t_max** (*int*) – number of bits in each int of each generating matrix. aka: number of rows in a generating matrix with ints expanded into columns
- **m_max** (*int*) – 2^m_max is the number of samples supported. aka: number of columns in a generating matrix with ints expanded into columns
- **msb** (*bool*) – bit storage as ints. e.g. if t_max=3, then 6 is [1 1 0] in MSB (True) and [0 1 1] in LSB (False)
- **t_lms** (*int*) – LMS scrambling matrix will be t_lms x t_max for generating matrix of shape t_max x m_max
- **_verbose** (*bool*) – print randomization details

gen_samples(*n=None*, *n_min=0*, *n_max=8*, *warn=True*, *return_unrandomized=False*)

Generate samples

Parameters

- **n** (*int*) – if n is supplied, generate from n_min=0 to n_max=n samples. Otherwise use the n_min and n_max explicitly supplied as the following 2 arguments
- **n_min** (*int*) – Starting index of sequence.
- **n_max** (*int*) – Final index of sequence.

- **return_unrandomized** (*bool*) – return unrandomized samples as well? If True, return randomized_samples,unrandomized_samples. Note that this only applies when randomize includes Digital Shift. Also note that unrandomized samples included linear matrix scrambling if applicable.

Returns
(n_max-n_min) x d (dimension) array of samples

Return type
ndarray

pdf(*x*)

pdf of a standard uniform

class qmcpy.discrete_distribution.digital_net_b2.digital_net_b2.**Sobol**(*dimension=1*, *randomize='LMS_DS'*, *graycode=False*, *seed=None*, *generating_matrices='sobol_mat.21201.32.32.msb*, *d_max=None*, *t_max=None*, *m_max=None*, *msb=None*, *t_lms=None*, *_verbose=False*)

4.1.3 Lattice

class qmcpy.discrete_distribution.lattice.lattice.**Lattice**(*dimension=1*, *randomize=True*, *order='natural'*, *seed=None*, *generating_vector='lattice_vec.3600.20.npy'*, *d_max=None*, *m_max=None*, *is_parallel=True*)

Quasi-Random Lattice nets in base 2.

```
>>> l = Lattice(2,seed=7)
>>> l.gen_samples(4)
array([[0.04386058, 0.58727432],
       [0.54386058, 0.08727432],
       [0.29386058, 0.33727432],
       [0.79386058, 0.83727432]])
>>> l.gen_samples(1)
array([[0.04386058, 0.58727432]])
>>> l
Lattice (DiscreteDistribution Object)
    d               2^(1)
    dvec            [0 1]
    randomize       1
    order           natural
    gen_vec         [     1 182667]
    entropy         7
    spawn_key       ()
>>> Lattice(dimension=2,randomize=False,order='natural').gen_samples(4, warn=False)
array([[0.  , 0.  ],
```

(continues on next page)

(continued from previous page)

```
       [0.5 , 0.5 ],
       [0.25, 0.75],
       [0.75, 0.25]])
>>> Lattice(dimension=2,randomize=False,order='linear').gen_samples(4, warn=False)
array([[0.  , 0.  ],
       [0.25, 0.75],
       [0.5 , 0.5 ],
       [0.75, 0.25]])
>>> Lattice(dimension=2,randomize=False,order='mps').gen_samples(4, warn=False)
array([[0.  , 0.  ],
       [0.5 , 0.5 ],
       [0.25, 0.75],
       [0.75, 0.25]])
>>> l = Lattice(2,generating_vector=25,seed=55)
>>> l.gen_samples(4)
array([[0.84489224, 0.30534549],
       [0.34489224, 0.80534549],
       [0.09489224, 0.05534549],
       [0.59489224, 0.55534549]])
>>> l
Lattice (DiscreteDistribution Object)
    d               2^(1)
    dvec            [0 1]
    randomize       1
    order           natural
    gen_vec         [       1 11961679]
    entropy         55
    spawn_key       ()
>>> Lattice(dimension=4,randomize=False,seed=353,generating_vector=26).gen_
↪samples(8,warn=False)
array([[0.   , 0.   , 0.   , 0.   ],
       [0.5  , 0.5  , 0.5  , 0.5  ],
       [0.25 , 0.25 , 0.75 , 0.75 ],
       [0.75 , 0.75 , 0.25 , 0.25 ],
       [0.125, 0.125, 0.875, 0.875],
       [0.625, 0.625, 0.375, 0.375],
       [0.375, 0.375, 0.625, 0.625],
       [0.875, 0.875, 0.125, 0.125]])
>>> Lattice(dimension=4,randomize=False,seed=353,generating_vector=26,is_
↪parallel=True).gen_samples(8,warn=False)
array([[0.   , 0.   , 0.   , 0.   ],
       [0.5  , 0.5  , 0.5  , 0.5  ],
       [0.25 , 0.25 , 0.75 , 0.75 ],
       [0.75 , 0.75 , 0.25 , 0.25 ],
       [0.125, 0.125, 0.875, 0.875],
       [0.625, 0.625, 0.375, 0.375],
       [0.375, 0.375, 0.625, 0.625],
       [0.875, 0.875, 0.125, 0.125]])
```

References

[1] Sou-Cheng T. Choi, Yuhan Ding, Fred J. Hickernell, Lan Jiang, Lluis Antoni Jimenez Rugama, Da Li, Jagadeeswaran Rathinavel, Xin Tong, Kan Zhang, Yizhi Zhang, and Xuan Zhou, GAIL: Guaranteed Automatic Integration Library (Version 2.3) [MATLAB Software], 2019. Available from http://gailgithub.github.io/GAIL_Dev/

[2] F.Y. Kuo & D. Nuyens. Application of quasi-Monte Carlo methods to elliptic PDEs with random diffusion coefficients - a survey of analysis and implementation, Foundations of Computational Mathematics, 16(6):1631-1696, 2016. springer link: https://link.springer.com/article/10.1007/s10208-016-9329-5 arxiv link: https://arxiv.org/abs/1606.06613

[3] D. Nuyens, *The Magic Point Shop of QMC point generators and generating vectors.* MATLAB and Python software, 2018. Available from https://people.cs.kuleuven.be/~dirk.nuyens/

[4] Constructing embedded lattice rules for multivariate integration R Cools, FY Kuo, D Nuyens - SIAM J. Sci. Comput., 28(6), 2162-2188.

[5] L'Ecuyer, Pierre & Munger, David. (2015). LatticeBuilder: A General Software Tool for Constructing Rank-1 Lattice Rules. ACM Transactions on Mathematical Software. 42. 10.1145/2754929.

__init__(*dimension=1*, *randomize=True*, *order='natural'*, *seed=None*, *generating_vector='lattice_vec.3600.20.npy'*, *d_max=None*, *m_max=None*, *is_parallel=True*)

Parameters

- **dimension** (*int or ndarray*) – dimension of the generator. If an int is passed in, use sequence dimensions [0,...,dimensions-1]. If a ndarray is passed in, use these dimension indices in the sequence.
- **randomize** (*bool*) – If True, apply shift to generated samples. Note: Non-randomized lattice sequence includes the origin.
- **order** (*str*) – 'linear', 'natural', or 'mps' ordering.
- **seed** (*None or int or numpy.random.SeedSeq*) – seed the random number generator for reproducibility
- **generating_vector** (*ndarray, str or int*) – generating matrix or path to generating matrices. ndarray should have shape (d_max). a string generating_vector should be formatted like 'lattice_vec.3600.20.npy' where 'name.d_max.m_max.npy' an integer should be an odd larger than 1; passing an integer M would create a random generating vector supporting up to 2^M points. M is restricted between 2 and 26 for numerical percision. The generating vector is [1,v_1,v_2,...,v_dimension], where v_i is an integer in {3,5,...,2*M-1}.
- **d_max** (*int*) – maximum dimension
- **m_max** (*int*) – 2^m_max is the max number of supported samples
- **is_parallel** (*bool*) – Default to True to perform parallel computations, False serial

Note: d_max and m_max are required if generating_vector is a ndarray. If generating_vector is an string (path), d_max and m_max can be taken from the file name if None

gen_samples(*n=None*, *n_min=0*, *n_max=8*, *warn=True*, *return_unrandomized=False*)

Generate lattice samples

Parameters

- **n** (*int*) – if n is supplied, generate from n_min=0 to n_max=n samples. Otherwise use the n_min and n_max explicitly supplied as the following 2 arguments
- **n_min** (*int*) – Starting index of sequence.
- **n_max** (*int*) – Final index of sequence.
- **return_unrandomized** (*bool*) – return samples without randomization as 2nd return value. Will not be returned if randomize=False.

Returns
(n_max-n_min) x d (dimension) array of samples

Return type
ndarray

Note: Lattice generates in blocks from 2**m to 2**(m+1) so generating n_min=3 to n_max=9 requires necessarily produces samples from n_min=2 to n_max=16 and automatically subsets. May be inefficient for non-powers-of-2 samples sizes.

---

pdf(*x*)

pdf of a standard uniform

4.1.4 Halton

class qmcpy.discrete_distribution.halton.**Halton**(*dimension=1*, *randomize=True*, *generalize=True*, *seed=None*)

Quasi-Random Halton nets.

```
>>> h_qrng = Halton(2,randomize='QRNG',generalize=True,seed=7)
>>> h_qrng.gen_samples(4)
array([[0.35362988, 0.38733489],
       [0.85362988, 0.72066823],
       [0.10362988, 0.05400156],
       [0.60362988, 0.498446  ]])
>>> h_qrng.gen_samples(1)
array([[0.35362988, 0.38733489]])
>>> h_qrng
Halton (DiscreteDistribution Object)
    d               2^(1)
    dvec            [0 1]
    randomize       QRNG
    generalize      1
    entropy         7
    spawn_key       ()
>>> h_owen = Halton(2,randomize='OWEN',generalize=False,seed=7)
>>> h_owen.gen_samples(4)
array([[0.64637012, 0.48226667],
       [0.14637012, 0.81560001],
       [0.89637012, 0.14893334],
       [0.39637012, 0.59337779]])
>>> h_owen
Halton (DiscreteDistribution Object)
```

(continues on next page)

(continued from previous page)

```
    d               2^(1)
    dvec            [0 1]
    randomize       OWEN
    generalize      0
    entropy         7
    spawn_key       ()
```

References

[1] Marius Hofert and Christiane Lemieux (2019). qrng: (Randomized) Quasi-Random Number Generators. R package version 0.0-7. https://CRAN.R-project.org/package=qrng.

[2] Owen, A. B. "A randomized Halton algorithm in R," 2017. arXiv:1706.02808 [stat.CO]

__init__(*dimension=1*, *randomize=True*, *generalize=True*, *seed=None*)

Parameters

- **dimension** (*int or ndarray*) – dimension of the generator. If an int is passed in, use sequence dimensions [0,…,dimensions-1]. If a ndarray is passed in, use these dimension indices in the sequence.
- **randomize** (*str/bool*) – select randomization method from 'QRNG" [1], (max dimension = 360, supports generalize=True, default if randomize=True) or 'OWEN' [2], (max dimension = 1000),
- **generalize** (*bool*) – generalize flag, only applicable to the QRNG generator
- **seed** (*None or int or numpy.random.SeedSeq*) – seed the random number generator for reproducibility

gen_samples(*n=None*, *n_min=0*, *n_max=8*, *warn=True*)

Generate samples

Parameters

- **n** (*int*) – if n is supplied, generate from n_min=0 to n_max=n samples. Otherwise use the n_min and n_max explicitly supplied as the following 2 arguments
- **n_min** (*int*) – Starting index of sequence.
- **n_max** (*int*) – Final index of sequence.

Returns
(n_max-n_min) x d (dimension) array of samples

Return type
ndarray

halton_owen(*n*, *n0*, *d0=0*)

see gen_samples method and [1] Owen, A. B.A randomized Halton algorithm in R2017. arXiv:1706.02808 [stat.CO].

pdf(*x*)

ABSTRACT METHOD to evaluate pdf of distribution the samples mimic at locations of x.

4.1.5 IID Standard Uniform

class qmcpy.discrete_distribution.iid_std_uniform.**IIDStdUniform**(*dimension=1*, *seed=None*)

A wrapper around NumPy's IID Standard Uniform generator *numpy.random.rand*.

```
>>> dd = IIDStdUniform(dimension=2,seed=7)
>>> dd.gen_samples(4)
array([[0.04386058, 0.58727432],
       [0.3691824 , 0.65212985],
       [0.69669968, 0.10605352],
       [0.63025643, 0.13630282]])
>>> dd
IIDStdUniform (DiscreteDistribution Object)
    d               2^(1)
    entropy         7
    spawn_key       ()
```

__init__(*dimension=1*, *seed=None*)

Parameters

- **dimension** (*int*) – dimension of samples
- **seed** (*None or int or numpy.random.SeedSeq*) – seed the random number generator for reproducibility

gen_samples(*n*)

Generate samples

Parameters
n (*int*) – Number of observations to generate

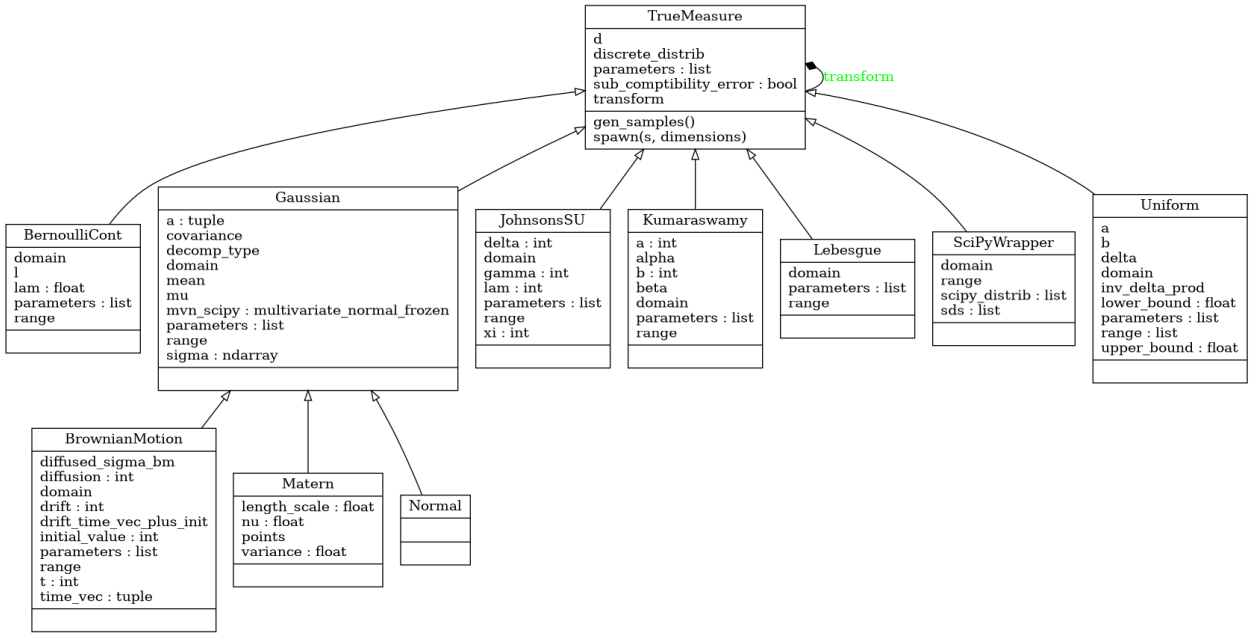
Returns
n x self.d array of samples

Return type
ndarray

pdf(*x*)

ABSTRACT METHOD to evaluate pdf of distribution the samples mimic at locations of x.

4.2 True Measure Class



4.2.1 Abstract Measure Class

class qmcpy.true_measure._true_measure.**TrueMeasure**

True Measure abstract class. DO NOT INSTANTIATE.

gen_samples(**args*, ***kwargs*)

Generate samples from the discrete distribution and transform them via the transform method.

Parameters

- **args** (*tuple*) – positional arguments to the discrete distributions gen_samples method
- **kwargs** (*dict*) – keyword arguments to the discrete distributions gen_samples method

Returns

n x d matrix of transformed samples

Return type

ndarray

spawn(*s=1*, *dimensions=None*)

Spawn new instances of the current discrete distribution but with new seeds and dimensions. Developed for multi-level and multi-replication (Q)MC algorithms.

Parameters

- **s** (*int*) – number of spawn
- **dimensions** (*ndarray*) – length s array of dimension for each spawn. Defaults to current dimension

Returns

list of TrueMeasures linked to newly spawned DiscreteDistributions

Return type
list

4.2.2 Uniform

class qmcpy.true_measure.uniform.**Uniform**(*sampler*, *lower_bound=0.0*, *upper_bound=1.0*)

```
>>> u = Uniform(DigitalNetB2(2,seed=7),lower_bound=[0,.5],upper_bound=[2,3])
>>> u.gen_samples(4)
array([[1.12538017, 0.93444992],
       [0.693306  , 2.12676579],
       [1.64149095, 2.88726434],
       [0.20844522, 1.73645241]])
>>> u
Uniform (TrueMeasure Object)
    lower_bound     [0.  0.5]
    upper_bound     [2 3]
```

__init__(*sampler*, *lower_bound=0.0*, *upper_bound=1.0*)

Parameters

- **sampler** (*DiscreteDistribution/TrueMeasure*) – A discrete distribution from which to transform samples or a true measure by which to compose a transform
- **lower_bound** (*float*) – a for Uniform(a,b)
- **upper_bound** (*float*) – b for Uniform(a,b)

4.2.3 Gaussian

class qmcpy.true_measure.gaussian.**Gaussian**(*sampler*, *mean=0.0*, *covariance=1.0*, *decomp_type='PCA'*)

Normal Measure.

```
>>> g = Gaussian(DigitalNetB2(2,seed=7),mean=[1,2],covariance=[[9,4],[4,5]])
>>> g.gen_samples(4)
array([[-0.23979685,  2.98944192],
       [ 2.45994002,  2.17853622],
       [-0.22923897, -1.92667105],
       [ 4.6127697 ,  4.25820377]])
>>> g
Gaussian (TrueMeasure Object)
    mean            [1 2]
    covariance      [[9 4]
                     [4 5]]
    decomp_type     PCA
```

__init__(*sampler*, *mean=0.0*, *covariance=1.0*, *decomp_type='PCA'*)

Parameters

- **sampler** (*DiscreteDistribution/TrueMeasure*) – A discrete distribution from which to transform samples or a true measure by which to compose a transform
- **mean** (*float*) – mu for Normal(mu,sigma^2)

- **covariance** (*ndarray*) – sigma^2 for Normal(mu,sigma^2). A float or d (dimension) vector input will be extended to covariance*eye(d)
- **decomp_type** (*str*) – method of decomposition either "PCA" for principal component analysis or "Cholesky" for cholesky decomposition.

class qmcpy.true_measure.gaussian.**Normal**(*sampler*, *mean=0.0*, *covariance=1.0*, *decomp_type='PCA'*)

4.2.4 Brownian Motion

class qmcpy.true_measure.brownian_motion.**BrownianMotion**(*sampler*, *t_final=1*, *initial_value=0*, *drift=0*, *diffusion=1*, *decomp_type='PCA'*)

Geometric Brownian Motion.

```
>>> bm = BrownianMotion(DigitalNetB2(4,seed=7),t_final=2,drift=2)
>>> bm.gen_samples(2)
array([[0.44018403, 1.62690477, 2.69418273, 4.21753174],
       [1.97549563, 2.27002956, 2.92802765, 4.77126959]])
>>> bm
BrownianMotion (TrueMeasure Object)
    time_vec        [0.5 1.  1.5 2. ]
    drift           2^(1)
    mean            [1. 2. 3. 4.]
    covariance      [[0.5 0.5 0.5 0.5]
                    [0.5 1.  1.  1. ]
                    [0.5 1.  1.5 1.5]
                    [0.5 1.  1.5 2. ]]
    decomp_type     PCA
```

__init__(*sampler*, *t_final=1*, *initial_value=0*, *drift=0*, *diffusion=1*, *decomp_type='PCA'*)

BrowianMotion(t) = (initial_value) + (drift)*t + sqrt{diffusion}*StandardBrownianMotion(t)

Parameters

- **sampler** (*DiscreteDistribution/TrueMeasure*) – A discrete distribution from which to transform samples or a true measure by which to compose a transform
- **t_final** (*float*) – end time for the Brownian Motion.
- **initial_value** (*float*) – See above formula
- **drift** (*int*) – See above formula
- **diffusion** (*int*) – See above formula
- **decomp_type** (*str*) – method of decomposition either "PCA" for principal component analysis or "Cholesky" for cholesky decomposition.

4.2.5 Lebesgue

class qmcpy.true_measure.lebesgue.**Lebesgue**(*sampler*)

```
>>> Lebesgue(Gaussian(DigitalNetB2(2,seed=7)))
Lebesgue (TrueMeasure Object)
    transform       Gaussian (TrueMeasure Object)
                       mean            0
                       covariance      1
                       decomp_type     PCA
>>> Lebesgue(Uniform(DigitalNetB2(2,seed=7)))
Lebesgue (TrueMeasure Object)
    transform       Uniform (TrueMeasure Object)
                       lower_bound     0
                       upper_bound     1
```

__init__(*sampler*)

Parameters

sampler (TrueMeasure) – A true measure by which to compose a transform.

4.2.6 Continuous Bernoulli

class qmcpy.true_measure.bernoulli_cont.**BernoulliCont**(*sampler*, *lam=0.5*)

```
>>> bc = BernoulliCont(DigitalNetB2(2,seed=7),lam=.2)
>>> bc.gen_samples(4)
array([[0.39545122, 0.10073414],
       [0.21719142, 0.48293404],
       [0.68958314, 0.90847415],
       [0.05871131, 0.33436033]])
>>> bc
BernoulliCont (TrueMeasure Object)
    lam             0.200
```

See https://en.wikipedia.org/wiki/Continuous_Bernoulli_distribution

__init__(*sampler*, *lam=0.5*)

Parameters

- **sampler** (*DiscreteDistribution/TrueMeasure*) – A discrete distribution from which to transform samples or a true measure by which to compose a transform
- **lam** (*ndarray*) – 0 < lambda < 1, a shape parameter, independent for each dimension

4.2.7 Johnson's SU

class qmcpy.true_measure.johnsons_su.**JohnsonsSU**(*sampler*, *gamma=1*, *xi=1*, *delta=2*, *lam=2*)

```
>>> jsu = JohnsonsSU(DigitalNetB2(2,seed=7),gamma=1,xi=2,delta=3,lam=4)
>>> jsu.gen_samples(4)
array([[ 0.86224892, -0.76967276],
       [ 0.07317047,  1.17727769],
       [ 1.89093286,  2.9341619 ],
       [-1.30283298,  0.62269632]])
>>> jsu
JohnsonsSU (TrueMeasure Object)
    gamma           1
    xi              2^(1)
    delta           3
    lam             2^(2)
```

See https://en.wikipedia.org/wiki/Johnson%27s_SU-distribution

__init__(*sampler*, *gamma=1*, *xi=1*, *delta=2*, *lam=2*)

Parameters

- **sampler** (*DiscreteDistribution/TrueMeasure*) – A discrete distribution from which to transform samples or a true measure by which to compose a transform
- **gamma** (*ndarray*) – gamma
- **xi** (*ndarray*) – xi
- **delta** (*ndarray*) – delta > 0
- **lam** (*ndarray*) – lambda > 0

4.2.8 Kumaraswamy

class qmcpy.true_measure.kumaraswamy.**Kumaraswamy**(*sampler*, *a=2*, *b=2*)

```
>>> k = Kumaraswamy(DigitalNetB2(2,seed=7),a=[1,2],b=[3,4])
>>> k.gen_samples(4)
array([[0.24096272, 0.21587652],
       [0.13227662, 0.4808615 ],
       [0.43615893, 0.73428949],
       [0.03602294, 0.39602319]])
>>> k
Kumaraswamy (TrueMeasure Object)
    a               [1 2]
    b               [3 4]
```

See https://en.wikipedia.org/wiki/Kumaraswamy_distribution

__init__(*sampler*, *a=2*, *b=2*)

Parameters

- **sampler** (*DiscreteDistribution/TrueMeasure*) – A discrete distribution from which to transform samples or a true measure by which to compose a transform

- **a** (*ndarray*) – alpha > 0
- **b** (*ndarray*) – beta > 0

4.2.9 SciPy Wrapper

class qmcpy.true_measure.scipy_wrapper.**SciPyWrapper**(*discrete_distrib*, *scipy_distribs*)

Multivariate True Measure from Independent SciPy 1 Dimensional Marginals

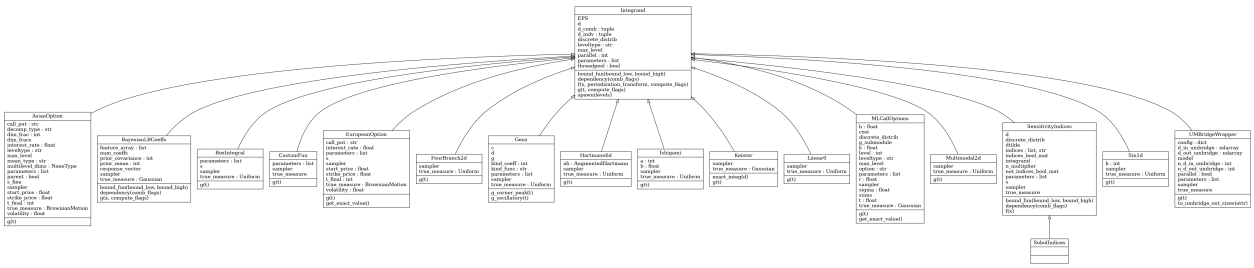
```
>>> unif_gauss_gamma = SciPyWrapper(
...     discrete_distrib = DigitalNetB2(3,seed=7),
...     scipy_distribs = [
...         scipy.stats.uniform(loc=1,scale=2),
...         scipy.stats.norm(loc=3,scale=4),
...         scipy.stats.gamma(a=5,loc=6,scale=7)])
>>> unif_gauss_gamma.range
array([[  1.,   3.],
       [-inf,  inf],
       [  6.,  inf]])
>>> unif_gauss_gamma.gen_samples(4)
array([[ 2.12538017, -0.75733093, 38.28471046],
       [ 1.693306  ,  4.54891231, 80.23287215],
       [ 2.64149095,  9.77761625, 43.6883765 ],
       [ 1.20844522,  2.94566431, 22.68122716]])
>>> betas_2d = SciPyWrapper(discrete_distrib=DigitalNetB2(2,seed=7),scipy_
↪distribs=scipy.stats.beta(a=5,b=1))
>>> betas_2d.gen_samples(4)
array([[0.89136146, 0.70469298],
       [0.80905676, 0.91764986],
       [0.96126183, 0.99081392],
       [0.63619813, 0.86865531]])
```

__init__(*discrete_distrib*, *scipy_distribs*)

Parameters

- **sampler** (*DiscreteDistribution/TrueMeasure*) – A discrete distribution from which to transform samples or a true measure by which to compose a transform
- **scipy_distribs** (*list*) – instantiated CONTINUOUS UNIVARIATE scipy.stats distributions https://docs.scipy.org/doc/scipy/reference/stats.html#continuous-distributions

4.3 Integrand Class



4.3.1 Abstract Integrand Class

class qmcpy.integrand._integrand.**Integrand**(*dimension_indv*, *dimension_comb*, *parallel*, *threadpool=False*)

Integrand abstract class. DO NOT INSTANTIATE.

__init__(*dimension_indv*, *dimension_comb*, *parallel*, *threadpool=False*)

Parameters

- **dimension_indv** (*tuple*) – individual solution shape.
- **dimension_comb** (*tuple*) – combined solution shape.
- **parallel** (*int*) – If parallel is False, 0, or 1: function evaluation is done in serial fashion. Otherwise, parallel specifies the number of processes used by multiprocessing.Pool or multiprocessing.pool.ThreadPool. Passing parallel=True sets processes = os.cpu_count().
- **threadpool** (*bool*) – When parallel > 1, if threadpool = True then use multiprocessing.pool.ThreadPool else use multiprocessing.Pool.

bound_fun(*bound_low*, *bound_high*)

Compute the bounds on the combined function based on bounds for the individual functions. Defaults to the identity where we essentially do not combine integrands, but instead integrate each function individually.

Parameters

- **bound_low** (*ndarray*) – length Integrand.d_indv lower error bound
- **bound_high** (*ndarray*) – length Integrand.d_indv upper error bound

Returns

(tuple) containing

- (ndarray): lower bound on function combining estimates
- (ndarray): upper bound on function combining estimates
- (ndarray): bool flags to override sufficient combined integrand estimation, e.g., when approximating a ratio of integrals, if the denominator's bounds straddle 0, then returning True here forces ratio to be flagged as insufficiently approximated.

dependency(*comb_flags*)

Takes a vector of indicators of weather of not the error bound is satisfied for combined integrands and which returns flags for individual integrands. For example, if we are taking the ratio of 2 individual integrands, then getting flag_comb=True means the ratio has not been approximated to within the tolerance, so the dependency function should return [True,True] indicating that both the numerator and denominator integrands need to be better approximated. :param comb_flags: flags indicating weather the combined integrals are insufficiently approximated :type comb_flags: bool ndarray

Returns
length (Integrand.d_indv) flags for individual integrands

Return type
(bool ndarray)

f(*x*, *periodization_transform='NONE'*, *compute_flags=None*, **args*, ***kwargs*)

Evaluate transformed integrand based on true measures and discrete distribution

Parameters

- **x** (*ndarray*) – n x d array of samples from a discrete distribution

- **periodization_transform** (*str*) – periodization transform
- **compute_flags** (*ndarray*) – outputs that require computation. For example, if the vector function has 3 outputs and compute_flags = [False, True, False], then the function is only required to compute the second output and may leave the remaining outputs as e.g. 0. The False outputs will not be used in the computation since those integrals have been sufficiently approximated.
- ***args** – other ordered args to g
- ****kwargs** (*dict*) – other keyword args to g

Returns
length n vector of function evaluations

Return type
ndarray

g(*t*, *compute_flags=None*, **args*, ***kwargs*)

ABSTRACT METHOD for original integrand to be integrated.

Parameters

- **t** (*ndarray*) – n x d array of samples to be input into original integrand.
- **compute_flags** (*ndarray*) – outputs that require computation. For example, if the vector function has 3 outputs and compute_flags = [False, True, False], then the function is only required to compute the second output and may leave the remaining outputs as e.g. 0. The False outputs will not be used in the computation since those integrals have been sufficiently approximated.

Returns
n vector of function evaluations

Return type
ndarray

spawn(*levels*)

Spawn new instances of the current integrand at the specified levels.

Parameters
levels (*ndarray*) – array of levels at which to spawn new integrands

Returns
list of Integrands linked to newly spawned TrueMeasures and DiscreteDistributions

Return type
list

4.3.2 Custom Function

class qmcpy.integrand.custom_fun.**CustomFun**(*true_measure*, *g*, *dimension_indv=1*, *parallel=False*)

Integrand wrapper for a user's function

```
>>> cf = CustomFun(
...     true_measure = Gaussian(DigitalNetB2(2,seed=7),mean=[1,2]),
...     g = lambda x: x[:,0]**2*x[:,1],
...     dimension_indv = 1)
>>> x = cf.discrete_distrib.gen_samples(2**10)
```

(continues on next page)

(continued from previous page)

```
>>> y = cf.f(x)
>>> y.shape
(1024, 1)
>>> y.mean()
3.995...
>>> cf = CustomFun(
...     true_measure = Uniform(DigitalNetB2(3,seed=7),lower_bound=[2,3,4],upper_
↪bound=[4,5,6]),
...     g = lambda x,compute_flags=None: x,
...     dimension_indv = 3)
>>> x = cf.discrete_distrib.gen_samples(2**10)
>>> y = cf.f(x)
>>> y.shape
(1024, 3)
>>> y.mean(0)
array([3., 4., 5.])
```

__init__(*true_measure*, *g*, *dimension_indv=1*, *parallel=False*)

Parameters

- **true_measure** (TrueMeasure) – a TrueMeasure instance.
- **g** (*function*) – a function handle.
- **dimension_indv** (*tuple*) – individual solution dimensions.
- **parallel** (*int*) – If parallel is False, 0, or 1: function evaluation is done in serial fashion. Otherwise, parallel specifies the number of CPUs used by multiprocessing.Pool. Passing parallel=True sets the number of CPUs equal to os.cpu_count(). Do NOT set g to a lambda function when doing parallel computation

g(*t*, **args*, ***kwargs*)

ABSTRACT METHOD for original integrand to be integrated.

Parameters

- **t** (*ndarray*) – n x d array of samples to be input into original integrand.
- **compute_flags** (*ndarray*) – outputs that require computation. For example, if the vector function has 3 outputs and compute_flags = [False, True, False], then the function is only required to compute the second output and may leave the remaining outputs as e.g. 0. The False outputs will not be used in the computation since those integrals have been sufficiently approximated.

Returns

n vector of function evaluations

Return type

ndarray

4.3.3 Keister Function

class qmcpy.integrand.keister.**Keister**(*sampler*)

$f(\boldsymbol{t}) = \pi^{d/2} \cos(\|\boldsymbol{t}\|)$.

The standard example integrates the Keister integrand with respect to an IID Gaussian distribution with variance 1./2.

```
>>> k = Keister(DigitalNetB2(2,seed=7))
>>> x = k.discrete_distrib.gen_samples(2**10)
>>> y = k.f(x)
>>> y.mean()
1.808...
>>> k.true_measure
Gaussian (TrueMeasure Object)
    mean            0
    covariance      2^(-1)
    decomp_type     PCA
>>> k = Keister(Gaussian(DigitalNetB2(2,seed=7),mean=0,covariance=2))
>>> x = k.discrete_distrib.gen_samples(2**12)
>>> y = k.f(x)
>>> y.mean()
1.808...
>>> yp = k.f(x,periodization_transform='c2sin')
>>> yp.mean()
1.807...
```

References

[1] B. D. Keister, Multidimensional Quadrature Algorithms, *Computers in Physics*, *10*, pp. 119-122, 1996.

__init__(*sampler*)

Parameters

sampler (*DiscreteDistribution/TrueMeasure*) – A discrete distribution from which to transform samples or a true measure by which to compose a transform

exact_integ(*d*)

computes the true value of the Keister integral in dimension d. Accuracy might degrade as d increases due to round-off error. :param d: :return: true_integral

g(*t*)

ABSTRACT METHOD for original integrand to be integrated.

Parameters

- **t** (*ndarray*) – n x d array of samples to be input into original integrand.
- **compute_flags** (*ndarray*) – outputs that require computation. For example, if the vector function has 3 outputs and compute_flags = [False, True, False], then the function is only required to compute the second output and may leave the remaining outputs as e.g. 0. The False outputs will not be used in the computation since those integrals have been sufficiently approximated.

Returns

n vector of function evaluations

Return type
ndarray

4.3.4 Box Integral

class qmcpy.integrand.box_integral.**BoxIntegral**(*sampler*, *s=array([1, 2])*)

$B_s(x) = \left(\sum_{j=1}^{d} x_j^2\right)^{s/2}$

```
>>> l1 = BoxIntegral(DigitalNetB2(2,seed=7), s=[7])
>>> x1 = l1.discrete_distrib.gen_samples(2**10)
>>> y1 = l1.f(x1)
>>> y1.shape
(1024, 1)
>>> y1.mean(0)
array([0.75156724])
>>> l2 = BoxIntegral(DigitalNetB2(5,seed=7), s=[-7,7])
>>> x2 = l2.discrete_distrib.gen_samples(2**10)
>>> y2 = l2.f(x2,compute_flags=[1,1])
>>> y2.shape
(1024, 2)
>>> y2.mean(0)
array([ 6.67548708, 10.52267786])
```

References:

[1] D.H. Bailey, J.M. Borwein, R.E. Crandall,Box integrals, Journal of Computational and Applied Mathematics, Volume 206, Issue 1, 2007, Pages 196-208, ISSN 0377-0427, https://doi.org/10.1016/j.cam.2006.06.010. (https://www.sciencedirect.com/science/article/pii/S0377042706004250)

[2] https://www.davidhbailey.com/dhbpapers/boxintegrals.pdf

__init__(*sampler*, *s=array([1, 2])*)

Parameters

- **sampler** (*DiscreteDistribution/TrueMeasure*) – A discrete distribution from which to transform samples or a true measure by which to compose a transform
- **s** (*list or ndarray*) – vectorized s parameter, len(s) is the number of vectorized integrals to evaluate.

g(*t*, ***kwargs*)

ABSTRACT METHOD for original integrand to be integrated.

Parameters

- **t** (*ndarray*) – n x d array of samples to be input into original integrand.
- **compute_flags** (*ndarray*) – outputs that require computation. For example, if the vector function has 3 outputs and compute_flags = [False, True, False], then the function is only required to compute the second output and may leave the remaining outputs as e.g. 0. The False outputs will not be used in the computation since those integrals have been sufficiently approximated.

Returns
n vector of function evaluations

Return type
ndarray

4.3.5 European Option

class qmcpy.integrand.european_option.**EuropeanOption**(*sampler*, *volatility=0.5*, *start_price=30*, *strike_price=35*, *interest_rate=0*, *t_final=1*, *call_put='call'*)

European financial option.

```
>>> eo = EuropeanOption(DigitalNetB2(4,seed=7),call_put='put')
>>> eo
EuropeanOption (Integrand Object)
    volatility      2^(-1)
    call_put        put
    start_price     30
    strike_price    35
    interest_rate   0
>>> x = eo.discrete_distrib.gen_samples(2**12)
>>> y = eo.f(x)
>>> y.mean()
9.209...
>>> eo = EuropeanOption(BrownianMotion(DigitalNetB2(4,seed=7),drift=1),call_put='put
↪')
>>> x = eo.discrete_distrib.gen_samples(2**12)
>>> y = eo.f(x)
>>> y.mean()
9.162...
>>> eo.get_exact_value()
9.211452976234058
```

__init__(*sampler*, *volatility=0.5*, *start_price=30*, *strike_price=35*, *interest_rate=0*, *t_final=1*, *call_put='call'*)

Parameters

- **sampler** (*DiscreteDistribution/TrueMeasure*) – A discrete distribution from which to transform samples or a true measure by which to compose a transform
- **volatility** (*float*) – sigma, the volatility of the asset
- **start_price** (*float*) – S(0), the asset value at t=0
- **strike_price** (*float*) – strike_price, the call/put offer
- **interest_rate** (*float*) – r, the annual interest rate
- **t_final** (*float*) – exercise time
- **call_put** (*str*) – 'call' or 'put' option

g(*t*)

See abstract method.

get_exact_value()

Get the fair price of a European call/put option.

Returns
fair price

Return type
float

4.3.6 Asian Option

class qmcpy.integrand.asian_option.**AsianOption**(*sampler*, *volatility=0.5*, *start_price=30.0*, *strike_price=35.0*, *interest_rate=0.0*, *t_final=1*, *call_put='call'*, *mean_type='arithmetic'*, *multilevel_dims=None*, *decomp_type='PCA'*, *_dim_frac=0*)

Asian financial option.

```
>>> ac = AsianOption(DigitalNetB2(4,seed=7))
>>> ac
AsianOption (Integrand Object)
    volatility      2^(-1)
    call_put        call
    start_price     30
    strike_price    35
    interest_rate   0
    mean_type       arithmetic
    dim_frac        0
>>> x = ac.discrete_distrib.gen_samples(2**12)
>>> y = ac.f(x)
>>> y.mean()
1.768...
>>> level_dims = [2,4,8]
>>> ac2_multilevel = AsianOption(DigitalNetB2(seed=7),multilevel_dims=level_dims)
>>> levels_to_spawn = arange(ac2_multilevel.max_level+1)
>>> ac2_single_levels = ac2_multilevel.spawn(levels_to_spawn)
>>> yml = 0
>>> for ac2_single_level in ac2_single_levels:
...     x = ac2_single_level.discrete_distrib.gen_samples(2**12)
...     level_est = ac2_single_level.f(x).mean()
...     yml += level_est
>>> yml
1.779...
```

__init__(*sampler*, *volatility=0.5*, *start_price=30.0*, *strike_price=35.0*, *interest_rate=0.0*, *t_final=1*, *call_put='call'*, *mean_type='arithmetic'*, *multilevel_dims=None*, *decomp_type='PCA'*, *_dim_frac=0*)

Parameters

- **sampler** (*DiscreteDistribution/TrueMeasure*) – A discrete distribution from which to transform samples or a true measure by which to compose a transform
- **volatility** (*float*) – sigma, the volatility of the asset
- **start_price** (*float*) – S(0), the asset value at t=0
- **strike_price** (*float*) – strike_price, the call/put offer

- **interest_rate** (*float*) – r, the annual interest rate
- **t_final** (*float*) – exercise time
- **mean_type** (*string*) – 'arithmetic' or 'geometric' mean
- **multilevel_dims** (*list of ints*) – list of dimensions at each level. Leave as None for single-level problems
- **_dim_frac** (*float*) – for internal use only, users should not set this parameter.

g(*t*)

ABSTRACT METHOD for original integrand to be integrated.

Parameters

- **t** (*ndarray*) – n x d array of samples to be input into original integrand.
- **compute_flags** (*ndarray*) – outputs that require computation. For example, if the vector function has 3 outputs and compute_flags = [False, True, False], then the function is only required to compute the second output and may leave the remaining outputs as e.g. 0. The False outputs will not be used in the computation since those integrals have been sufficiently approximated.

Returns

n vector of function evaluations

Return type

ndarray

4.3.7 Multilevel Call Options with Milstein Discretization

class qmcpy.integrand.ml_call_options.**MLCallOptions**(*sampler*, *option='european'*, *volatility=0.2*, *start_strike_price=100.0*, *interest_rate=0.05*, *t_final=1.0*, *_level=0*)

Various call options from finance using Milstein discretization with 2^l timesteps on level l.

```
>>> mlco_original = MLCallOptions(DigitalNetB2(seed=7))
>>> mlco_original
MLCallOptions (Integrand Object)
    option          european
    sigma           0.200
    k               100
    r               0.050
    t               1
    b               85
    level           0
>>> mlco_ml_dims = mlco_original.spawn(levels=arange(4))
>>> yml = 0
>>> for mlco in mlco_ml_dims:
...     x = mlco.discrete_distrib.gen_samples(2**10)
...     yml += mlco.f(x).mean()
>>> yml
10.393...
```

References:

[1] M.B. Giles. Improved multilevel Monte Carlo convergence using the Milstein scheme. 343-358, in Monte Carlo and Quasi-Monte Carlo Methods 2006, Springer, 2008. http://people.maths.ox.ac.uk/~gilesm/files/mcqmc06.pdf.

__init__(*sampler*, *option='european'*, *volatility=0.2*, *start_strike_price=100.0*, *interest_rate=0.05*, *t_final=1.0*, *_level=0*)

Parameters

- **sampler** (*DiscreteDistribution/TrueMeasure*) – A discrete distribution from which to transform samples or a true measure by which to compose a transform
- **option_type** (*str*) – type of option in ["European","Asian"]
- **volatility** (*float*) – sigma, the volatility of the asset
- **start_strike_price** (*float*) – S(0), the asset value at t=0, and K, the strike price. Assume start_price = strike_price
- **interest_rate** (*float*) – r, the annual interest rate
- **t_final** (*float*) – exercise time
- **_level** (*int*) – for internal use only, users should not set this parameter.

g(*t*)

Parameters
t (*ndarray*) – Gaussian(0,1^2) samples

Returns
First, an ndarray of length 6 vector of summary statistic sums. Second, a float of cost on this level.

Return type
tuple

get_exact_value()

Print exact analytic value, based on s0=k.

4.3.8 Linear Function

class qmcpy.integrand.linear0.**Linear0**(*sampler*)

```
>>> l = Linear0(DigitalNetB2(100,seed=7))
>>> x = l.discrete_distrib.gen_samples(2**10)
>>> y = l.f(x)
>>> y.mean()
-1.175...e-08
>>> ytf = l.f(x,periodization_transform='C1SIN')
>>> ytf.mean()
-4.050...e-12
```

__init__(*sampler*)

Parameters
sampler (*DiscreteDistribution/TrueMeasure*) – A discrete distribution from which to transform samples or a true measure by which to compose a transform

g(*t*)

ABSTRACT METHOD for original integrand to be integrated.

Parameters

- **t** (*ndarray*) – n x d array of samples to be input into original integrand.
- **compute_flags** (*ndarray*) – outputs that require computation. For example, if the vector function has 3 outputs and compute_flags = [False, True, False], then the function is only required to compute the second output and may leave the remaining outputs as e.g. 0. The False outputs will not be used in the computation since those integrals have been sufficiently approximated.

Returns

n vector of function evaluations

Return type

ndarray

4.3.9 Bayesian Logistic Regression

class qmcpy.integrand.bayesian_lr_coeffs.**BayesianLRCoeffs**(*sampler*, *feature_array*, *response_vector*, *prior_mean=0*, *prior_covariance=10*)

Logistic Regression Coefficients computed as the posterior mean in a Bayesian framework.

```
>>> blrcoeffs = BayesianLRCoeffs(DigitalNetB2(3,seed=7),feature_array=arange(8).
↪reshape((4,2)),response_vector=[0,0,1,1])
>>> x = blrcoeffs.discrete_distrib.gen_samples(2**10)
>>> y = blrcoeffs.f(x)
>>> y.shape
(1024, 2, 3)
>>> y.mean(0)
array([[ 0.04639394, -0.01440543, -0.05498496],
       [ 0.02176581,  0.02176581,  0.02176581]])
```

__init__(*sampler*, *feature_array*, *response_vector*, *prior_mean=0*, *prior_covariance=10*)

Parameters

- **sampler** (*DiscreteDistribution/TrueMeasure*) – A discrete distribution from which to transform samples or a true measure by which to compose a transform
- **feature_array** (*ndarray*) – N samples by d-1 dimensions array of input features
- **response_vector** (*ndarray*) – length N array of binary responses corresponding to each
- **prior_mean** (*ndarray*) – length d array of prior means, one for each coefficient. The first d-1 inputs correspond to the d-1 features. The last input corresponds to the intercept coefficient.
- **prior_covariance** (*ndarray*) – d x d prior covariance array whose indexing is consistent with the prior mean.

bound_fun(*bound_low*, *bound_high*)

Compute the bounds on the combined function based on bounds for the individual functions. Defaults to the identity where we essentially do not combine integrands, but instead integrate each function individually.

Parameters

- **bound_low** (*ndarray*) – length Integrand.d_indv lower error bound
- **bound_high** (*ndarray*) – length Integrand.d_indv upper error bound

Returns

(tuple) containing

- (ndarray): lower bound on function combining estimates
- (ndarray): upper bound on function combining estimates
- (ndarray): bool flags to override sufficient combined integrand estimation, e.g., when approximating a ratio of integrals, if the denominator's bounds straddle 0, then returning True here forces ratio to be flagged as insufficiently approximated.

dependency(*comb_flags*)

Takes a vector of indicators of weather of not the error bound is satisfied for combined integrands and which returns flags for individual integrands. For example, if we are taking the ratio of 2 individual integrands, then getting flag_comb=True means the ratio has not been approximated to within the tolerance, so the dependency function should return [True,True] indicating that both the numerator and denominator integrands need to be better approximated. :param comb_flags: flags indicating weather the combined integrals are insufficiently approximated :type comb_flags: bool ndarray

Returns
length (Integrand.d_indv) flags for individual integrands

Return type
(bool ndarray)

g(*x*, *compute_flags*)

ABSTRACT METHOD for original integrand to be integrated.

Parameters

- **t** (*ndarray*) – n x d array of samples to be input into original integrand.
- **compute_flags** (*ndarray*) – outputs that require computation. For example, if the vector function has 3 outputs and compute_flags = [False, True, False], then the function is only required to compute the second output and may leave the remaining outputs as e.g. 0. The False outputs will not be used in the computation since those integrals have been sufficiently approximated.

Returns
n vector of function evaluations

Return type
ndarray

4.3.10 Genz Function

class qmcpy.integrand.genz.**Genz**(*sampler*, *kind_func='oscilatory'*, *kind_coeff=1*)

https://dakota.sandia.gov/sites/default/files/docs/6.17.0-release/user-html/usingdakota/examples/additionalexamples.html?highlight=genz#genz-functions

```
>>> for kind_func in ['oscilatory','corner-peak']:
...     for kind_coeff in [1,2,3]:
...         g = Genz(DigitalNetB2(2,seed=7),kind_func=kind_func,kind_coeff=kind_
↪coeff)
```

(continues on next page)

(continued from previous page)

```
...         x = g.discrete_distrib.gen_samples(2**14)
...         y = g.f(x)
...         mu_hat = y.mean()
...         print('%-15s %-3d %.3f'%(kind_func,kind_coeff,mu_hat))
oscilatory      1   -0.351
oscilatory      2   -0.380
oscilatory      3   -0.217
corner-peak     1   0.713
corner-peak     2   0.712
corner-peak     3   0.720
```

__init__(*sampler*, *kind_func='oscilatory'*, *kind_coeff=1*)

Parameters

- **sampler** (*DiscreteDistribution/TrueMeasure*) – A discrete distribution from which to transform samples or a true measure by which to compose a transform
- **kind_func** (*str*) – 'oscilatory' or 'corner-peak'
- **kind_coeff** (*int*) – 1, 2, or 3 for choice of coefficients

4.3.11 Ishigami Function

class qmcpy.integrand.ishigami.**Ishigami**(*sampler*, *a=7*, *b=0.1*)

$g(\boldsymbol{t}) = (1 + bt_2^4)\sin(t_0) + a\sin^2(t_1), \qquad T \sim \mathcal{U}(-\pi, \pi)^3$

https://www.sfu.ca/~ssurjano/ishigami.html

```
>>> ishigami = Ishigami(DigitalNetB2(3,seed=7))
>>> x = ishigami.discrete_distrib.gen_samples(2**10)
>>> y = ishigami.f(x)
>>> y.mean()
3.499...
>>> ishigami.true_measure
Uniform (TrueMeasure Object)
    lower_bound     -3.142
    upper_bound     3.142
```

References

[1] Ishigami, T., & Homma, T. (1990, December). An importance quantification technique in uncertainty analysis for computer models. In Uncertainty Modeling and Analysis, 1990. Proceedings., First International Symposium on (pp. 398-403). IEEE.

__init__(*sampler*, *a=7*, *b=0.1*)

Parameters

- **sampler** (*DiscreteDistribution/TrueMeasure*) – A discrete distribution from which to transform samples or a true measure by which to compose a transform
- **a** (*float*) – fixed parameters in above equation
- **b** (*float*) – fixed parameters in above equation

`g`(*t*)

ABSTRACT METHOD for original integrand to be integrated.

Parameters

- **t** (*ndarray*) – n x d array of samples to be input into original integrand.
- **compute_flags** (*ndarray*) – outputs that require computation. For example, if the vector function has 3 outputs and compute_flags = [False, True, False], then the function is only required to compute the second output and may leave the remaining outputs as e.g. 0. The False outputs will not be used in the computation since those integrals have been sufficiently approximated.

Returns

n vector of function evaluations

Return type

ndarray

4.3.12 Sensitivity Indices

class qmcpy.integrand.sensitivity_indices.**SensitivityIndices**(*integrand*, *indices='singletons'*)

Sensitivity' Indicies, normalized Sobol' Indices.

```
>>> dnb2 = DigitalNetB2(dimension=3,seed=7)
>>> keister_d = Keister(dnb2)
>>> keister_indices = SensitivityIndices(keister_d,indices='singletons')
>>> sc = CubQMCNetG(keister_indices,abs_tol=1e-3)
>>> solution,data = sc.integrate()
>>> solution.squeeze()
array([[0.32803639, 0.32795358, 0.32807359],
       [0.33884667, 0.33857811, 0.33884115]])
>>> data
LDTransformData (AccumulateData Object)
    solution        [[0.328 0.328 0.328]
                    [0.339 0.339 0.339]]
    comb_bound_low  [[0.327 0.327 0.328]
                    [0.338 0.338 0.338]]
    comb_bound_high [[0.329 0.329 0.329]
                    [0.34  0.339 0.34 ]]
    comb_flags      [[ True  True  True]
                    [ True  True  True]]
    n_total         2^(16)
    n               [[[65536. 65536. 65536.]
                      [65536. 65536. 65536.]
                      [65536. 65536. 65536.]]

                     [[32768. 32768. 32768.]
                      [32768. 32768. 32768.]
                      [32768. 32768. 32768.]]]
    time_integrate  ...
CubQMCNetG (StoppingCriterion Object)
    abs_tol         0.001
    rel_tol         0
```

(continues on next page)

(continued from previous page)

```
    n_init          2^(10)
    n_max           2^(35)
SensitivityIndices (Integrand Object)
    indices         [[0]
                    [1]
                    [2]]
    n_multiplier    3
Gaussian (TrueMeasure Object)
    mean            0
    covariance      2^(-1)
    decomp_type     PCA
DigitalNetB2 (DiscreteDistribution Object)
    d               6
    dvec            [0 1 2 3 4 5]
    randomize       LMS_DS
    graycode        0
    entropy         7
    spawn_key       (0,)
>>> sc = CubQMCNetG(SobolIndices(BoxIntegral(DigitalNetB2(3,seed=7)),indices='all'),
↪abs_tol=.01)
>>> sol,data = sc.integrate()
>>> print(sol)
[[[0.32312991 0.33340559]
  [0.32331463 0.33342669]
  [0.32160276 0.33318619]
  [0.65559598 0.6667154 ]
  [0.65551702 0.66670251]
  [0.6556618  0.66672429]]

 [[0.3440018  0.33341845]
  [0.34501082 0.33347005]
  [0.34504829 0.33345212]
  [0.67659368 0.6667021 ]
  [0.67725088 0.66667925]
  [0.67802866 0.66672587]]]
```

References

[1] Art B. Owen.Monte Carlo theory, methods and examples. 2013. Appendix A.

__init__(*integrand*, *indices='singletons'*)

Parameters

- **integrand** (Integrand) – integrand to find Sobol' indices of
- **indices** (*list of lists*) – each element of indices should be a list of indices, u, at which to compute the Sobol' indices. The default indices='singletons' sets indices=[[0],[1],...[d-1]]. Should not include [], the null set, or [0,...,d-1], the set of all indices. Setting indices='all' will compute all sensitivity indices

bound_fun(*bound_low*, *bound_high*)

Compute the bounds on the combined function based on bounds for the individual functions. Defaults to the identity where we essentially do not combine integrands, but instead integrate each function individually.

Parameters

- **bound_low** (*ndarray*) – length Integrand.d_indv lower error bound
- **bound_high** (*ndarray*) – length Integrand.d_indv upper error bound

Returns

(tuple) containing

- (ndarray): lower bound on function combining estimates
- (ndarray): upper bound on function combining estimates
- (ndarray): bool flags to override sufficient combined integrand estimation, e.g., when approximating a ratio of integrals, if the denominator's bounds straddle 0, then returning True here forces ratio to be flagged as insufficiently approximated.

dependency(*comb_flags*)

Takes a vector of indicators of weather of not the error bound is satisfied for combined integrands and which returns flags for individual integrands. For example, if we are taking the ratio of 2 individual integrands, then getting flag_comb=True means the ratio has not been approximated to within the tolerance, so the dependency function should return [True,True] indicating that both the numerator and denominator integrands need to be better approximated. :param comb_flags: flags indicating weather the combined integrals are insufficiently approximated :type comb_flags: bool ndarray

Returns
length (Integrand.d_indv) flags for individual integrands

Return type
(bool ndarray)

f(*x*, **args*, ***kwargs*)

Evaluate transformed integrand based on true measures and discrete distribution

Parameters

- **x** (*ndarray*) – n x d array of samples from a discrete distribution
- **periodization_transform** (*str*) – periodization transform
- **compute_flags** (*ndarray*) – outputs that require computation. For example, if the vector function has 3 outputs and compute_flags = [False, True, False], then the function is only required to compute the second output and may leave the remaining outputs as e.g. 0. The False outputs will not be used in the computation since those integrals have been sufficiently approximated.
- ***args** – other ordered args to g
- ****kwargs** (*dict*) – other keyword args to g

Returns
length n vector of function evaluations

Return type
ndarray

class qmcpy.integrand.sensitivity_indices.**SobolIndices**(*integrand*, *indices='singletons'*)

Normalized Sobol' Indices, an alias for SensitivityIndices.

4.3.13 UM-Bridge Wrapper

class qmcpy.integrand.um_bridge_wrapper.**UMBridgeWrapper**(*true_measure*, *model*, *config={}*, *parallel=False*)

UM-Bridge Model Wrapper. Requires Docker be installed, see https://www.docker.com/.

```
>>> _ = os.system('docker run --name muqbp -dit -p 4243:4243 linusseelinger/
↪benchmark-muq-beam-propagation:latest > /dev/null')
>>> import umbridge
>>> dnb2 = DigitalNetB2(dimension=3,seed=7)
>>> distribution = Uniform(dnb2,lower_bound=1,upper_bound=1.05)
>>> model = umbridge.HTTPModel('http://localhost:4243','forward')
>>> umbridge_config = {"d": dnb2.d}
>>> um_bridge_integrand = UMBridgeWrapper(distribution,model,umbridge_config,
↪parallel=False)
>>> solution,data = CubQMCNetG(um_bridge_integrand,abs_tol=5e-2).integrate()
>>> print(data)
LDTransformData (AccumulateData Object)
    solution        [  0.      3.855  14.69  ... 898.921 935.383 971.884]
    comb_bound_low  [  0.      3.854  14.688 ... 898.901 935.363 971.863]
    comb_bound_high [  0.      3.855  14.691 ... 898.941 935.404 971.906]
    comb_flags      [ True  True  True ...  True  True  True]
    n_total         2^(11)
    n               [1024. 1024. 1024. ... 2048. 2048. 2048.]
    time_integrate  ...
CubQMCNetG (StoppingCriterion Object)
    abs_tol         0.050
    rel_tol         0
    n_init          2^(10)
    n_max           2^(35)
UMBridgeWrapper (Integrand Object)
Uniform (TrueMeasure Object)
    lower_bound     1
    upper_bound     1.050
DigitalNetB2 (DiscreteDistribution Object)
    d               3
    dvec            [0 1 2]
    randomize       LMS_DS
    graycode        0
    entropy         7
    spawn_key       ()
>>> _ = os.system('docker rm -f muqbp > /dev/null')
>>> class TestModel(umbridge.Model):
...     def __init__(self):
...         super().__init__("forward")
...     def get_input_sizes(self, config):
...         return [1,2,3]
...     def get_output_sizes(self, config):
...         return [3,2,1]
...     def __call__(self, parameters, config):
...         out0 = [parameters[2][0],sum(parameters[2][:2]),sum(parameters[2])]
...         out1 = [parameters[1][0],sum(parameters[1])]
...         out2 = [parameters[0]]
```

(continues on next page)

(continued from previous page)

```
...         return [out0,out1,out2]
...     def supports_evaluate(self):
...         return True
>>> my_model = TestModel()
>>> my_distribution = Uniform(
...     sampler = DigitalNetB2(dimension=sum(my_model.get_input_sizes(config={})),
↪seed=7),
...     lower_bound = -1,
...     upper_bound = 1)
>>> my_integrand = UMBridgeWrapper(my_distribution,my_model)
>>> my_solution,my_data = CubQMCNetG(my_integrand,abs_tol=5e-2).integrate()
>>> my_data
LDTransformData (AccumulateData Object)
    solution        [-2.328e-10 -4.657e-10 -6.985e-10 -2.328e-10 -4.657e-10 -2.328e-
↪10]
    comb_bound_low  [-9.649e-06 -1.765e-04 -2.352e-04 -3.053e-07 -1.997e-05 -1.952e-
↪05]
    comb_bound_high [9.649e-06 1.765e-04 2.352e-04 3.048e-07 1.997e-05 1.952e-05]
    comb_flags      [ True  True  True  True  True  True]
    n_total         2^(10)
    n               [1024. 1024. 1024. 1024. 1024. 1024.]
    time_integrate  ...
CubQMCNetG (StoppingCriterion Object)
    abs_tol         0.050
    rel_tol         0
    n_init          2^(10)
    n_max           2^(35)
UMBridgeWrapper (Integrand Object)
Uniform (TrueMeasure Object)
    lower_bound     -1
    upper_bound     1
DigitalNetB2 (DiscreteDistribution Object)
    d               6
    dvec            [0 1 2 3 4 5]
    randomize       LMS_DS
    graycode        0
    entropy         7
    spawn_key       ()
>>> my_integrand.to_umbridge_out_sizes(my_solution)
[[-2.3283064365386963e-10, -4.656612873077393e-10, -6.984919309616089e-10], [-2.
↪3283064365386963e-10, -4.656612873077393e-10], [-2.3283064365386963e-10]]
>>> my_integrand.to_umbridge_out_sizes(my_data.comb_bound_low)
[[-9.649316780269146e-06, -0.00017654551993473433, -0.00023524149401055183], [-3.
↪0527962735504843e-07, -1.997367189687793e-05], [-1.9521190552040935e-05]]
>>> my_integrand.to_umbridge_out_sizes(my_data.comb_bound_high)
[[9.648851118981838e-06, 0.00017654458861215971, 0.00023524000970266899], [3.
↪048139660677407e-07, 1.9972740574303316e-05], [1.9520724890753627e-05]]
```

References

[1] UM-Bridge documentation. https://um-bridge-benchmarks.readthedocs.io/en/docs/index.html

__init__(*true_measure*, *model*, *config={}*, *parallel=False*)

See https://um-bridge-benchmarks.readthedocs.io/en/docs/umbridge/clients.html

Parameters

- **true_measure** (TrueMeasure) – a TrueMeasure instance.
- **model** (*umbridge.HTTPModel*) – a UM-Bridge model
- **config** (*dict*) – config keyword argument to umbridge.HTTPModel(url,name).__call__
- **parallel** (*int*) – If parallel is False, 0, or 1: function evaluation is done in serial fashion. Otherwise, parallel specifies the number of processes used by multiprocessing.Pool or multiprocessing.pool.ThreadPool. Passing parallel=True sets processes = os.cpu_count().

g(*t*, ***kwargs*)

ABSTRACT METHOD for original integrand to be integrated.

Parameters

- **t** (*ndarray*) – n x d array of samples to be input into original integrand.
- **compute_flags** (*ndarray*) – outputs that require computation. For example, if the vector function has 3 outputs and compute_flags = [False, True, False], then the function is only required to compute the second output and may leave the remaining outputs as e.g. 0. The False outputs will not be used in the computation since those integrals have been sufficiently approximated.

Returns

n vector of function evaluations

Return type

ndarray

to_umbridge_out_sizes(*attr*)

Convert a data attribute to UM-Bridge output sized list of lists.

Parameters

attr (*ndarray*) – array of length sum(model.get_output_sizes(self.config))

Returns

list of lists with sub-list lengths specified by model.get_output_sizes(self.config)

Return type

list

4.3.14 Sin 1d

class qmcpy.integrand.sin1d.**Sin1d**(*sampler*, *k=1*)

```
>>> sin1d = Sin1d(DigitalNetB2(1,seed=7))
>>> x = sin1d.discrete_distrib.gen_samples(2**10)
>>> y = sin1d.f(x)
>>> y.mean()
7.33449186...e-08
```

(continues on next page)

(continued from previous page)

```
>>> sin1d.true_measure
Uniform (TrueMeasure Object)
    lower_bound     0
    upper_bound     6.283
```

g(*t*)

ABSTRACT METHOD for original integrand to be integrated.

Parameters

- **t** (*ndarray*) – n x d array of samples to be input into original integrand.
- **compute_flags** (*ndarray*) – outputs that require computation. For example, if the vector function has 3 outputs and compute_flags = [False, True, False], then the function is only required to compute the second output and may leave the remaining outputs as e.g. 0. The False outputs will not be used in the computation since those integrals have been sufficiently approximated.

Returns

n vector of function evaluations

Return type

ndarray

4.3.15 Multimodal 2d

class qmcpy.integrand.multimodal2d.**Multimodal2d**(*sampler*)

```
>>> mm2d = Multimodal2d(DigitalNetB2(2,seed=7))
>>> x = mm2d.discrete_distrib.gen_samples(2**10)
>>> y = mm2d.f(x)
>>> y.mean()
-0.7365118306607449
>>> mm2d.true_measure
Uniform (TrueMeasure Object)
    lower_bound     [-4 -3]
    upper_bound     [7 8]
```

g(*t*)

ABSTRACT METHOD for original integrand to be integrated.

Parameters

- **t** (*ndarray*) – n x d array of samples to be input into original integrand.
- **compute_flags** (*ndarray*) – outputs that require computation. For example, if the vector function has 3 outputs and compute_flags = [False, True, False], then the function is only required to compute the second output and may leave the remaining outputs as e.g. 0. The False outputs will not be used in the computation since those integrals have been sufficiently approximated.

Returns

n vector of function evaluations

Return type

ndarray

4.3.16 Four Branch 2d

class qmcpy.integrand.fourbranch2d.**FourBranch2d**(*sampler*)

```
>>> fb2d = FourBranch2d(DigitalNetB2(2,seed=7))
>>> x = fb2d.discrete_distrib.gen_samples(2**10)
>>> y = fb2d.f(x)
>>> y.mean()
-2.5003746135324247
>>> fb2d.true_measure
Uniform (TrueMeasure Object)
    lower_bound     -8
    upper_bound     2^(3)
```

g(*t*)

ABSTRACT METHOD for original integrand to be integrated.

Parameters

- **t** (*ndarray*) – n x d array of samples to be input into original integrand.
- **compute_flags** (*ndarray*) – outputs that require computation. For example, if the vector function has 3 outputs and compute_flags = [False, True, False], then the function is only required to compute the second output and may leave the remaining outputs as e.g. 0. The False outputs will not be used in the computation since those integrals have been sufficiently approximated.

Returns
n vector of function evaluations

Return type
ndarray

4.3.17 Hartmann 6d

class qmcpy.integrand.hartmann6d.**Hartmann6d**(*sampler*)

```
>>> h6d = Hartmann6d(DigitalNetB2(6,seed=7))
>>> x = h6d.discrete_distrib.gen_samples(2**10)
>>> y = h6d.f(x)
>>> y.mean()
-0.2613140309713834
>>> h6d.true_measure
Uniform (TrueMeasure Object)
    lower_bound     0
    upper_bound     1
```

g(*t*)

ABSTRACT METHOD for original integrand to be integrated.

Parameters

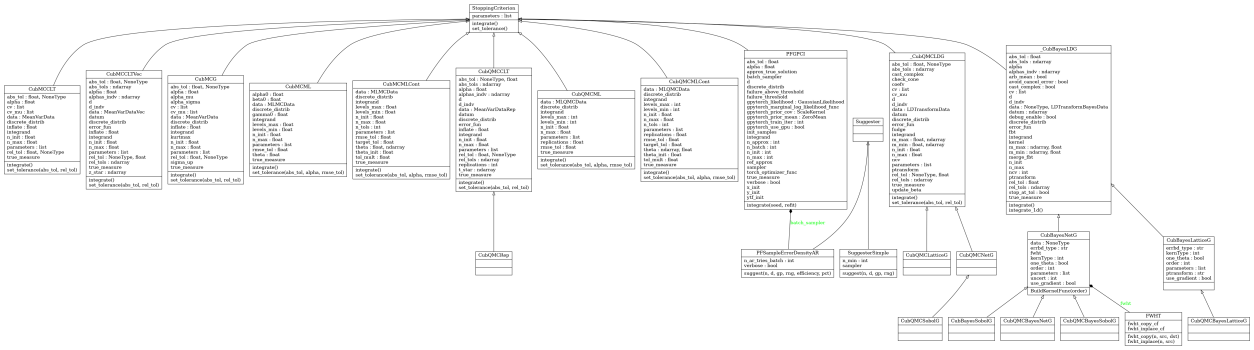
- **t** (*ndarray*) – n x d array of samples to be input into original integrand.
- **compute_flags** (*ndarray*) – outputs that require computation. For example, if the vector function has 3 outputs and compute_flags = [False, True, False], then the function is only

required to compute the second output and may leave the remaining outputs as e.g. 0. The False outputs will not be used in the computation since those integrals have been sufficiently approximated.

Returns
n vector of function evaluations

Return type
ndarray

4.4 Stopping Criterion Algorithms



4.4.1 Abstract Stopping Criterion Class

class qmcpy.stopping_criterion._stopping_criterion.**StoppingCriterion**(*allowed_levels*, *allowed_distribs*, *allow_vectorized_integrals*)

Stopping Criterion abstract class. DO NOT INSTANTIATE.

__init__(*allowed_levels*, *allowed_distribs*, *allow_vectorized_integrals*)

Parameters

- **distribution** (DiscreteDistribution) – a DiscreteDistribution
- **allowed_levels** (*list*) – which integrand types are supported: ‘single’, ‘fixed-multi’, ‘adaptive-multi’
- **allowed_distribs** (*list*) – list of compatible DiscreteDistribution classes

integrate()

ABSTRACT METHOD to determine the number of samples needed to satisfy the tolerance.

Returns

tuple containing:

- solution (float): approximation to the integral
- data (AccumulateData): an AccumulateData object

Return type
tuple

set_tolerance(**args*, ***kwargs*)

ABSTRACT METHOD to reset the absolute tolerance.

4.4.2 Guaranteed Digital Net Cubature (QMC)

class qmcpy.stopping_criterion.cub_qmc_net_g.**CubQMCNetG**(*integrand*, *abs_tol=0.01*, *rel_tol=0.0*, *n_init=1024.0*, *n_max=34359738368.0*, *fudge=<function CubQMCNetG.<lambda>>*, *check_cone=False*, *control_variates=[]*, *control_variate_means=[]*, *update_beta=False*, *error_fun=<function CubQMCNetG.<lambda>>*)

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.

```
>>> k = Keister(DigitalNetB2(2,seed=7))
>>> sc = CubQMCNetG(k,abs_tol=.05)
>>> solution,data = sc.integrate()
>>> data
LDTransformData (AccumulateData Object)
    solution        1.809
    comb_bound_low  1.804
    comb_bound_high 1.814
    comb_flags      1
    n_total         2^(10)
    n               2^(10)
    time_integrate  ...
CubQMCNetG (StoppingCriterion Object)
    abs_tol         0.050
    rel_tol         0
    n_init          2^(10)
    n_max           2^(35)
Keister (Integrand Object)
Gaussian (TrueMeasure Object)
    mean            0
    covariance      2^(-1)
    decomp_type     PCA
DigitalNetB2 (DiscreteDistribution Object)
    d               2^(1)
    dvec            [0 1]
    randomize       LMS_DS
    graycode        0
    entropy         7
    spawn_key       ()
>>> dd = DigitalNetB2(3,seed=7)
>>> g1 = CustomFun(Uniform(dd,0,2),lambda t: 10*t[:,0]-5*t[:,1]**2+t[:,2]**3)
>>> cv1 = CustomFun(Uniform(dd,0,2),lambda t: t[:,0])
>>> cv2 = CustomFun(Uniform(dd,0,2),lambda t: t[:,1]**2)
>>> sc = CubQMCNetG(g1,abs_tol=1e-6,check_cone=True,
...     control_variates = [cv1,cv2],
...     control_variate_means = [1,4/3])
```

(continues on next page)

(continued from previous page)

```
>>> sol,data = sc.integrate()
>>> sol
array([5.33333333])
>>> exactsol = 16/3
>>> abs(sol-exactsol)<1e-6
array([ True])
>>> dnb2 = DigitalNetB2(3,seed=7)
>>> f = BoxIntegral(dnb2, s=[-1,1])
>>> abs_tol = 1e-3
>>> sc = CubQMCNetG(f, abs_tol=abs_tol)
>>> solution,data = sc.integrate()
>>> solution
array([1.18944142, 0.96064165])
>>> sol3neg1 = -pi/4-1/2*log(2)+log(5+3*sqrt(3))
>>> sol31 = sqrt(3)/4+1/2*log(2+sqrt(3))-pi/24
>>> true_value = array([sol3neg1,sol31])
>>> (abs(true_value-solution)<abs_tol).all()
True
>>> f2 = BoxIntegral(dnb2,s=[3,4])
>>> sc = CubQMCNetG(f2,control_variates=f,control_variate_means=true_value,update_
↪beta=True)
>>> solution,data = sc.integrate()
>>> solution
array([1.10168119, 1.26661293])
>>> data
LDTransformData (AccumulateData Object)
    solution        [1.102 1.267]
    comb_bound_low  [1.099 1.262]
    comb_bound_high [1.104 1.271]
    comb_flags      [ True  True]
    n_total         2^(10)
    n               [1024. 1024.]
    time_integrate  ...
CubQMCNetG (StoppingCriterion Object)
    abs_tol         0.010
    rel_tol         0
    n_init          2^(10)
    n_max           2^(35)
    cv              BoxIntegral (Integrand Object)
                       s               [-1  1]
    cv_mu           [1.19  0.961]
    update_beta     1
BoxIntegral (Integrand Object)
    s               [3 4]
Uniform (TrueMeasure Object)
    lower_bound     0
    upper_bound     1
DigitalNetB2 (DiscreteDistribution Object)
    d               3
    dvec            [0 1 2]
    randomize       LMS_DS
    graycode        0
```

(continues on next page)

(continued from previous page)

```
    entropy         7
    spawn_key       ()
>>> cf = CustomFun(
...     true_measure = Uniform(DigitalNetB2(6,seed=7)),
...     g = lambda x,compute_flags=None: (2*arange(1,7)*x).reshape(-1,2,3),
...     dimension_indv = (2,3))
>>> sol,data = CubQMCNetG(cf,abs_tol=1e-6).integrate()
>>> data
LDTransformData (AccumulateData Object)
    solution        [[1. 2. 3.]
                    [4. 5. 6.]]
    comb_bound_low  [[1. 2. 3.]
                    [4. 5. 6.]]
    comb_bound_high [[1. 2. 3.]
                    [4. 5. 6.]]
    comb_flags      [[ True  True  True]
                    [ True  True  True]]
    n_total         2^(13)
    n               [[2048. 1024. 1024.]
                    [8192. 4096. 2048.]]
    time_integrate  ...
CubQMCNetG (StoppingCriterion Object)
    abs_tol         1.00e-06
    rel_tol         0
    n_init          2^(10)
    n_max           2^(35)
CustomFun (Integrand Object)
Uniform (TrueMeasure Object)
    lower_bound     0
    upper_bound     1
DigitalNetB2 (DiscreteDistribution Object)
    d               6
    dvec            [0 1 2 3 4 5]
    randomize       LMS_DS
    graycode        0
    entropy         7
    spawn_key       ()
```

Original Implementation:

https://github.com/GailGithub/GAIL_Dev/blob/master/Algorithms/IntegrationExpectation/cubSobol_g.m

References

[1] Fred J. Hickernell and Lluis Antoni Jimenez Rugama, Reliable adaptive cubature using digital sequences, 2014. Submitted for publication: arXiv:1410.8615.

[2] Sou-Cheng T. Choi, Yuhan Ding, Fred J. Hickernell, Lan Jiang, Lluis Antoni Jimenez Rugama, Da Li, Jagadeeswaran Rathinavel, Xin Tong, Kan Zhang, Yizhi Zhang, and Xuan Zhou, GAIL: Guaranteed Automatic Integration Library (Version 2.3) [MATLAB Software], 2019. Available from http://gailgithub.github.io/GAIL_Dev/

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 below.

__init__(*integrand*, *abs_tol=0.01*, *rel_tol=0.0*, *n_init=1024.0*, *n_max=34359738368.0*, *fudge=<function CubQMCNetG.<lambda>>*, *check_cone=False*, *control_variates=[]*, *control_variate_means=[]*, *update_beta=False*, *error_fun=<function CubQMCNetG.<lambda>>*)

Parameters

- **integrand** (Integrand) – an instance of Integrand
- **abs_tol** (*ndarray*) – absolute error tolerance
- **rel_tol** (*ndarray*) – relative error tolerance
- **n_init** (*int*) – initial number of samples
- **n_max** (*int*) – maximum number of samples
- **fudge** (*function*) – positive function multiplying the finite sum of Fast Fourier coefficients specified in the cone of functions
- **check_cone** (*boolean*) – check if the function falls in the cone
- **control_variates** (*list*) – list of integrand objects to be used as control variates. Control variates are currently only compatible with single level problems. The same discrete distribution instance must be used for the integrand and each of the control variates.
- **control_variate_means** (*list*) – list of means for each control variate
- **update_beta** (*bool*) – update control variate beta coefficients at each iteration
- **error_fun** – function taking in the approximate solution vector, absolute tolerance, and relative tolerance which returns the approximate error. Default indicates integration until either absolute OR relative tolerance is satisfied.

class qmcpy.stopping_criterion.cub_qmc_net_g.**CubQMCSobolG**(*integrand*, *abs_tol=0.01*, *rel_tol=0.0*, *n_init=1024.0*, *n_max=34359738368.0*, *fudge=<function CubQMCNetG.<lambda>>*, *check_cone=False*, *control_variates=[]*, *control_variate_means=[]*, *update_beta=False*, *error_fun=<function CubQMCNetG.<lambda>>*)

4.4.3 Guaranteed Lattice Cubature (QMC)

class qmcpy.stopping_criterion.cub_qmc_lattice_g.**CubQMCLatticeG**(*integrand*, *abs_tol=0.01*, *rel_tol=0.0*, *n_init=1024.0*, *n_max=34359738368.0*, *fudge=<function CubQMCLatticeG.<lambda>>*, *check_cone=False*, *ptransform='Baker'*, *error_fun=<function CubQMCLatticeG.<lambda>>*)

Stopping Criterion 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.

```
>>> k = Keister(Lattice(2,seed=7))
>>> sc = CubQMCLatticeG(k,abs_tol=.05)
>>> solution,data = sc.integrate()
>>> data
LDTransformData (AccumulateData Object)
    solution        1.810
    comb_bound_low  1.806
    comb_bound_high 1.815
    comb_flags      1
    n_total         2^(10)
    n               2^(10)
    time_integrate  ...
CubQMCLatticeG (StoppingCriterion Object)
    abs_tol         0.050
    rel_tol         0
    n_init          2^(10)
    n_max           2^(35)
Keister (Integrand Object)
Gaussian (TrueMeasure Object)
    mean            0
    covariance      2^(-1)
    decomp_type     PCA
Lattice (DiscreteDistribution Object)
    d               2^(1)
    dvec            [0 1]
    randomize       1
    order           natural
    gen_vec         [     1 182667]
    entropy         7
    spawn_key       ()
>>> f = BoxIntegral(Lattice(3,seed=7), s=[-1,1])
>>> abs_tol = 1e-3
>>> sc = CubQMCLatticeG(f, abs_tol=abs_tol)
>>> solution,data = sc.integrate()
>>> solution
array([1.18954582, 0.96056304])
>>> sol3neg1 = -pi/4-1/2*log(2)+log(5+3*sqrt(3))
>>> sol31 = sqrt(3)/4+1/2*log(2+sqrt(3))-pi/24
```

(continues on next page)

(continued from previous page)

```
>>> true_value = array([sol3neg1,sol31])
>>> (abs(true_value-solution)<abs_tol).all()
True
>>> cf = CustomFun(
...     true_measure = Uniform(Lattice(6,seed=7)),
...     g = lambda x,compute_flags=None: (2*arange(1,7)*x).reshape(-1,2,3),
...     dimension_indv = (2,3))
>>> sol,data = CubQMCLatticeG(cf,abs_tol=1e-6).integrate()
>>> data
LDTransformData (AccumulateData Object)
    solution        [[1. 2. 3.]
                    [4. 5. 6.]]
    comb_bound_low  [[1. 2. 3.]
                    [4. 5. 6.]]
    comb_bound_high [[1. 2. 3.]
                    [4. 5. 6.]]
    comb_flags      [[ True  True  True]
                    [ True  True  True]]
    n_total         2^(15)
    n               [[ 8192. 16384. 16384.]
                    [16384. 32768. 32768.]]
    time_integrate  ...
CubQMCLatticeG (StoppingCriterion Object)
    abs_tol         1.00e-06
    rel_tol         0
    n_init          2^(10)
    n_max           2^(35)
CustomFun (Integrand Object)
Uniform (TrueMeasure Object)
    lower_bound     0
    upper_bound     1
Lattice (DiscreteDistribution Object)
    d               6
    dvec            [0 1 2 3 4 5]
    randomize       1
    order           natural
    gen_vec         [     1 182667 469891 498753 110745 446247]
    entropy         7
    spawn_key       ()
```

Original Implementation:

https://github.com/GailGithub/GAIL_Dev/blob/master/Algorithms/IntegrationExpectation/cubLattice_g.m

References

[1] Lluis Antoni Jimenez Rugama and Fred J. Hickernell, "Adaptive multidimensional integration based on rank-1 lattices," Monte Carlo and Quasi-Monte Carlo Methods: MCQMC, Leuven, Belgium, April 2014 (R. Cools and D. Nuyens, eds.), Springer Proceedings in Mathematics and Statistics, vol. 163, Springer-Verlag, Berlin, 2016, arXiv:1411.1966, pp. 407-422.

[2] Sou-Cheng T. Choi, Yuhan Ding, Fred J. Hickernell, Lan Jiang, Lluis Antoni Jimenez Rugama, Da Li, Jagadeeswaran Rathinavel, Xin Tong, Kan Zhang, Yizhi Zhang, and Xuan Zhou, GAIL: Guaranteed Automatic Integration Library (Version 2.3) [MATLAB Software], 2019. Available from http://gailgithub.github.io/GAIL_Dev/

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 below.

__init__(*integrand*, *abs_tol=0.01*, *rel_tol=0.0*, *n_init=1024.0*, *n_max=34359738368.0*, *fudge=<function CubQMCLatticeG.<lambda>>*, *check_cone=False*, *ptransform='Baker'*, *error_fun=<function CubQMCLatticeG.<lambda>>*)

Parameters

- **integrand** (Integrand) – an instance of Integrand
- **abs_tol** (*ndarray*) – absolute error tolerance
- **rel_tol** (*ndarray*) – relative error tolerance
- **n_init** (*int*) – initial number of samples
- **n_max** (*int*) – maximum number of samples
- **fudge** (*function*) – positive function multiplying the finite sum of Fast Fourier coefficients specified in the cone of functions
- **check_cone** (*boolean*) – check if the function falls in the cone
- **error_fun** – function taking in the approximate solution vector, absolute tolerance, and relative tolerance which returns the approximate error. Default indicates integration until either absolute OR relative tolerance is satisfied.

4.4.4 Bayesian Lattice Cubature (QMC)

class qmcpy.stopping_criterion.cub_qmc_bayes_lattice_g.**CubBayesLatticeG**(*integrand*, *abs_tol=0.01*, *rel_tol=0*, *n_init=256*, *n_max=4194304*, *order=2*, *alpha=0.01*, *ptransform='C1sin'*, *error_fun=<function CubBayesLatticeG.<lambda>>*)

Stopping criterion for Bayesian Cubature using rank-1 Lattice sequence with guaranteed accuracy over a d-dimensional region to integrate within a specified generalized error tolerance with guarantees under Bayesian assumptions.

```
>>> k = Keister(Lattice(2, order='linear', seed=123456789))
>>> sc = CubBayesLatticeG(k,abs_tol=.05)
>>> solution,data = sc.integrate()
>>> data
LDTransformBayesData (AccumulateData Object)
    solution        1.808
    comb_bound_low  1.808
    comb_bound_high 1.809
    comb_flags      1
    n_total         2^(8)
    n               2^(8)
    time_integrate  ...
CubBayesLatticeG (StoppingCriterion Object)
    abs_tol         0.050
    rel_tol         0
    n_init          2^(8)
    n_max           2^(22)
    order           2^(1)
Keister (Integrand Object)
Gaussian (TrueMeasure Object)
    mean            0
    covariance      2^(-1)
    decomp_type     PCA
Lattice (DiscreteDistribution Object)
    d               2^(1)
    dvec            [0 1]
    randomize       1
    order           linear
    gen_vec         [     1 182667]
    entropy         123456789
    spawn_key       ()
```

Adapted from GAIL cubBayesLattice_g.

Guarantees:

This algorithm attempts to calculate the integral of function f over the hyperbox $[0,1]^d$ to a prescribed error tolerance tolfun $:= max(\text{abstol}, \text{reltol} * |I|)$ with a 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:

$$Pr(|Q - I| <= \text{tolfun}) = 99\%.$$

This Bayesian cubature algorithm guarantees for integrands that are considered to be an instance of a Gaussian process that falls in the middle of samples space spanned. Where The sample space is spanned by the covariance kernel parametrized by the scale and shape parameter inferred from the sampled values of the integrand. For more details on how the covariance kernels are defined and the parameters are obtained, please refer to the references below.

References

[1] Jagadeeswaran Rathinavel and Fred J. Hickernell, Fast automatic Bayesian cubature using lattice sampling. Stat Comput 29, 1215-1229 (2019). Available from Springer.

[2] Sou-Cheng T. Choi, Yuhan Ding, Fred J. Hickernell, Lan Jiang, Lluis Antoni Jimenez Rugama, Da Li, Jagadeeswaran Rathinavel, Xin Tong, Kan Zhang, Yizhi Zhang, and Xuan Zhou, GAIL: Guaranteed Automatic Integration Library (Version 2.3) [MATLAB Software], 2019. Available from GAIL.

__init__(*integrand*, *abs_tol=0.01*, *rel_tol=0*, *n_init=256*, *n_max=4194304*, *order=2*, *alpha=0.01*, *ptransform='C1sin'*, *error_fun=<function CubBayesLatticeG.<lambda>>*)

Parameters

- **integrand** (Integrand) – an instance of Integrand
- **abs_tol** (*ndarray*) – absolute error tolerance
- **rel_tol** (*ndarray*) – relative error tolerance
- **n_init** (*int*) – initial number of samples
- **n_max** (*int*) – maximum number of samples
- **order** (*int*) – Bernoulli kernel's order. If zero, choose order automatically
- **alpha** (*float*) – p-value
- **ptransform** (*str*) – periodization transform applied to the integrand
- **error_fun** – function taking in the approximate solution vector, absolute tolerance, and relative tolerance which returns the approximate error. Default indicates integration until either absolute OR relative tolerance is satisfied.

class qmcpy.stopping_criterion.cub_qmc_bayes_lattice_g.**CubQMCBayesLatticeG**(*integrand*, *abs_tol=0.01*, *rel_tol=0*, *n_init=256*, *n_max=4194304*, *order=2*, *alpha=0.01*, *ptransform='C1sin'*, *error_fun=<function CubBayesLatticeG.<lambda>>*)

4.4.5 Bayesian Digital Net Cubature (QMC)

class qmcpy.stopping_criterion.cub_qmc_bayes_net_g.**CubBayesNetG**(*integrand*, *abs_tol=0.01*, *rel_tol=0*, *n_init=256*, *n_max=4194304*, *alpha=0.01*, *error_fun=<function CubBayesNetG.<lambda>>*)

Stopping criterion for Bayesian Cubature using digital net sequence with guaranteed accuracy over a d-dimensional region to integrate within a specified generalized error tolerance with guarantees under Bayesian assumptions.

```
>>> k = Keister(DigitalNetB2(2, seed=123456789))
>>> sc = CubBayesNetG(k,abs_tol=.05)
>>> solution,data = sc.integrate()
>>> data
LDTransformBayesData (AccumulateData Object)
    solution        1.812
    comb_bound_low  1.796
    comb_bound_high 1.827
    comb_flags      1
    n_total         2^(8)
    n               2^(8)
    time_integrate  ...
CubBayesNetG (StoppingCriterion Object)
    abs_tol         0.050
    rel_tol         0
    n_init          2^(8)
    n_max           2^(22)
Keister (Integrand Object)
Gaussian (TrueMeasure Object)
    mean            0
    covariance      2^(-1)
    decomp_type     PCA
DigitalNetB2 (DiscreteDistribution Object)
    d               2^(1)
    dvec            [0 1]
    randomize       LMS_DS
    graycode        0
    entropy         123456789
    spawn_key       ()
```

Adapted from GAIL cubBayesNet_g.

Guarantee:

This algorithm attempts to calculate the integral of function f over the hyperbox $[0,1]^d$ to a prescribed error tolerance tolfun $:= max(\text{abstol}, \text{reltol} * |I|)$ with a 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:

$$Pr(|Q - I| <= \text{tolfun}) = 99\%.$$

This Bayesian cubature algorithm guarantees for integrands that are considered to be an instance of a Gaussian process that falls in the middle of samples space spanned. Where The sample space is spanned by the covariance kernel parametrized by the scale and shape parameter inferred from the sampled values of the integrand. For more details on how the covariance kernels are defined and the parameters are obtained, please refer to the references below.

References

[1] Jagadeeswaran Rathinavel, Fast automatic Bayesian cubature using matching kernels and designs, PhD thesis, Illinois Institute of Technology, 2019.

[2] Sou-Cheng T. Choi, Yuhan Ding, Fred J. Hickernell, Lan Jiang, Lluis Antoni Jimenez Rugama, Da Li, Jagadeeswaran Rathinavel, Xin Tong, Kan Zhang, Yizhi Zhang, and Xuan Zhou, GAIL: Guaranteed Automatic Integration Library (Version 2.3) [MATLAB Software], 2019. Available from GAIL.

__init__(*integrand*, *abs_tol=0.01*, *rel_tol=0*, *n_init=256*, *n_max=4194304*, *alpha=0.01*, *error_fun=<function CubBayesNetG.<lambda>>*)

Parameters

- **integrand** (Integrand) – an instance of Integrand
- **abs_tol** (*ndarray*) – absolute error tolerance
- **rel_tol** (*ndarray*) – relative error tolerance
- **n_init** (*int*) – initial number of samples
- **n_max** (*int*) – maximum number of samples
- **alpha** (*float*) – signifcance level or p-value
- **error_fun** – function taking in the approximate solution vector, absolute tolerance, and relative tolerance which returns the approximate error. Default indicates integration until either absolute OR relative tolerance is satisfied.

class qmcpy.stopping_criterion.cub_qmc_bayes_net_g.**CubBayesSobolG**(*integrand*, *abs_tol=0.01*, *rel_tol=0*, *n_init=256*, *n_max=4194304*, *alpha=0.01*, *error_fun=<function CubBayesNetG.<lambda>>*)

class qmcpy.stopping_criterion.cub_qmc_bayes_net_g.**CubQMCBayesNetG**(*integrand*, *abs_tol=0.01*, *rel_tol=0*, *n_init=256*, *n_max=4194304*, *alpha=0.01*, *error_fun=<function CubBayesNetG.<lambda>>*)

class qmcpy.stopping_criterion.cub_qmc_bayes_net_g.**CubQMCBayesSobolG**(*integrand*, *abs_tol=0.01*, *rel_tol=0*, *n_init=256*, *n_max=4194304*, *alpha=0.01*, *error_fun=<function CubBayesNetG.<lambda>>*)

4.4.6 CLT QMC Cubature (with Replications)

class qmcpy.stopping_criterion.cub_qmc_clt.**CubQMCCLT**(*integrand*, *abs_tol=0.01*, *rel_tol=0.0*, *n_init=256.0*, *n_max=1073741824*, *inflate=1.2*, *alpha=0.01*, *replications=16.0*, *error_fun=<function CubQMCCLT.<lambda>>*)

Stopping criterion based on Central Limit Theorem for multiple replications.

```
>>> k = Keister(Lattice(seed=7))
>>> sc = CubQMCCLT(k,abs_tol=.05)
>>> solution,data = sc.integrate()
>>> solution
array([1.38030146])
>>> data
MeanVarDataRep (AccumulateData Object)
    solution        1.380
    comb_bound_low  1.380
    comb_bound_high 1.381
    comb_flags      1
    n_total         2^(12)
    n               2^(12)
    n_rep           2^(8)
    time_integrate  ...
CubQMCCLT (StoppingCriterion Object)
    inflate         1.200
    alpha           0.010
    abs_tol         0.050
    rel_tol         0
    n_init          2^(8)
    n_max           2^(30)
    replications    2^(4)
Keister (Integrand Object)
Gaussian (TrueMeasure Object)
    mean            0
    covariance      2^(-1)
    decomp_type     PCA
Lattice (DiscreteDistribution Object)
    d               1
    dvec            0
    randomize       1
    order           natural
    gen_vec         1
    entropy         7
    spawn_key       ()
>>> f = BoxIntegral(Lattice(3,seed=7), s=[-1,1])
>>> abs_tol = 1e-3
>>> sc = CubQMCCLT(f, abs_tol=abs_tol)
>>> solution,data = sc.integrate()
>>> solution
array([1.19023153, 0.96068581])
>>> data
MeanVarDataRep (AccumulateData Object)
```

(continues on next page)

(continued from previous page)

```
    solution        [1.19  0.961]
    comb_bound_low  [1.19 0.96]
    comb_bound_high [1.191 0.961]
    comb_flags      [ True  True]
    n_total         2^(21)
    n               [2097152.    8192.]
    n_rep           [131072.    512.]
    time_integrate  ...
CubQMCCLT (StoppingCriterion Object)
    inflate         1.200
    alpha           0.010
    abs_tol         0.001
    rel_tol         0
    n_init          2^(8)
    n_max           2^(30)
    replications    2^(4)
BoxIntegral (Integrand Object)
    s               [-1  1]
Uniform (TrueMeasure Object)
    lower_bound     0
    upper_bound     1
Lattice (DiscreteDistribution Object)
    d               3
    dvec            [0 1 2]
    randomize       1
    order           natural
    gen_vec         [     1 182667 469891]
    entropy         7
    spawn_key       ()
>>> sol3neg1 = -pi/4-1/2*log(2)+log(5+3*sqrt(3))
>>> sol31 = sqrt(3)/4+1/2*log(2+sqrt(3))-pi/24
>>> true_value = array([sol3neg1,sol31])
>>> (abs(true_value-solution)<abs_tol).all()
True
>>> cf = CustomFun(
...     true_measure = Uniform(DigitalNetB2(6,seed=7)),
...     g = lambda x,compute_flags=None: (2*arange(1,7)*x).reshape(-1,2,3),
...     dimension_indv = (2,3))
>>> sol,data = CubQMCCLT(cf,abs_tol=1e-4).integrate()
>>> data
MeanVarDataRep (AccumulateData Object)
    solution        [[1. 2. 3.]
                     [4. 5. 6.]]
    comb_bound_low  [[1. 2. 3.]
                     [4. 5. 6.]]
    comb_bound_high [[1. 2. 3.]
                     [4. 5. 6.]]
    comb_flags      [[ True  True  True]
                     [ True  True  True]]
    n_total         2^(14)
    n               [[ 4096.  4096.  4096.]
                     [16384.  4096.  4096.]]
```

(continues on next page)

(continued from previous page)

```
    n_rep           [[ 256.  256.  256.]
                    [1024.  256.  256.]]
    time_integrate  ...
CubQMCCLT (StoppingCriterion Object)
    inflate         1.200
    alpha           0.010
    abs_tol         1.00e-04
    rel_tol         0
    n_init          2^(8)
    n_max           2^(30)
    replications    2^(4)
CustomFun (Integrand Object)
Uniform (TrueMeasure Object)
    lower_bound     0
    upper_bound     1
DigitalNetB2 (DiscreteDistribution Object)
    d               6
    dvec            [0 1 2 3 4 5]
    randomize       LMS_DS
    graycode        0
    entropy         7
    spawn_key       ()
```

__init__(*integrand*, *abs_tol=0.01*, *rel_tol=0.0*, *n_init=256.0*, *n_max=1073741824*, *inflate=1.2*, *alpha=0.01*, *replications=16.0*, *error_fun=<function CubQMCCLT.<lambda>>*)

Parameters

- **integrand** (Integrand) – an instance of Integrand
- **inflate** (*float*) – inflation factor when estimating variance
- **alpha** (*ndarray*) – significance level for confidence interval
- **abs_tol** (*ndarray*) – absolute error tolerance
- **rel_tol** (*ndarray*) – relative error tolerance
- **n_max** (*int*) – maximum number of samples
- **replications** (*int*) – number of replications
- **error_fun** – function taking in the approximate solution vector, absolute tolerance, and relative tolerance which returns the approximate error. Default indicates integration until either absolute OR relative tolerance is satisfied.

integrate()

See abstract method.

set_tolerance(*abs_tol=None*, *rel_tol=None*)

See abstract method.

Parameters

- **abs_tol** (*float*) – absolute tolerance. Reset if supplied, ignored if not.
- **rel_tol** (*float*) – relative tolerance. Reset if supplied, ignored if not.

class qmcpy.stopping_criterion.cub_qmc_clt.**CubQMCRep**(*integrand*, *abs_tol=0.01*, *rel_tol=0.0*, *n_init=256.0*, *n_max=1073741824*, *inflate=1.2*, *alpha=0.01*, *replications=16.0*, *error_fun=<function CubQMCCLT.<lambda>>*)

4.4.7 Guaranteed MC Cubature

class qmcpy.stopping_criterion.cub_mc_g.**CubMCG**(*integrand*, *abs_tol=0.01*, *rel_tol=0.0*, *n_init=1024.0*, *n_max=10000000000.0*, *inflate=1.2*, *alpha=0.01*, *control_variates=[]*, *control_variate_means=[]*)

Stopping criterion with guaranteed accuracy.

```
>>> k = Keister(IIDStdUniform(2,seed=7))
>>> sc = CubMCG(k,abs_tol=.05)
>>> solution,data = sc.integrate()
>>> data
MeanVarData (AccumulateData Object)
    solution        1.807
    error_bound     0.050
    n_total         15256
    n               14232
    levels          1
    time_integrate  ...
CubMCG (StoppingCriterion Object)
    abs_tol         0.050
    rel_tol         0
    n_init          2^(10)
    n_max           10000000000
    inflate         1.200
    alpha           0.010
Keister (Integrand Object)
Gaussian (TrueMeasure Object)
    mean            0
    covariance      2^(-1)
    decomp_type     PCA
IIDStdUniform (DiscreteDistribution Object)
    d               2^(1)
    entropy         7
    spawn_key       ()
>>> dd = IIDStdUniform(1,seed=7)
>>> k = Keister(dd)
>>> cv1 = CustomFun(Uniform(dd),lambda x: sin(pi*x).sum(1))
>>> cv1mean = 2/pi
>>> cv2 = CustomFun(Uniform(dd),lambda x: (-3*(x-.5)**2+1).sum(1))
>>> cv2mean = 3/4
>>> sc1 = CubMCG(k,abs_tol=.05,control_variates=[cv1,cv2],control_variate_
 ↪means=[cv1mean,cv2mean])
>>> sol,data = sc1.integrate()
>>> sol
1.384...
```

Original Implementation:

https://github.com/GailGithub/GAIL_Dev/blob/master/Algorithms/IntegrationExpectation/meanMC_g.m

References

[1] Fred J. Hickernell, Lan Jiang, Yuewei Liu, and Art B. Owen, "Guaranteed conservative fixed width confidence intervals via Monte Carlo sampling," Monte Carlo and Quasi-Monte Carlo Methods 2012 (J. Dick, F. Y. Kuo, G. W. Peters, and I. H. Sloan, eds.), pp. 105-128, Springer-Verlag, Berlin, 2014. DOI: 10.1007/978-3-642-41095-6_5

[2] Sou-Cheng T. Choi, Yuhan Ding, Fred J. Hickernell, Lan Jiang, Lluis Antoni Jimenez Rugama, Da Li, Jagadeeswaran Rathinavel, Xin Tong, Kan Zhang, Yizhi Zhang, and Xuan Zhou, GAIL: Guaranteed Automatic Integration Library (Version 2.3) [MATLAB Software], 2019. Available from http://gailgithub.github.io/GAIL_Dev/

Guarantee:

This algorithm attempts to calculate the mean, mu, of a random variable to a prescribed error tolerance, _tol_fun:= 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: ¶($|mu - tmu|$
$letol\_fun$)
$ge1 -$
$alpha$.

__init__(*integrand*, *abs_tol=0.01*, *rel_tol=0.0*, *n_init=1024.0*, *n_max=10000000000.0*, *inflate=1.2*, *alpha=0.01*, *control_variates=[]*, *control_variate_means=[]*)

Parameters

- **integrand** (Integrand) – an instance of Integrand
- **inflate** – inflation factor when estimating variance
- **alpha** – significance level for confidence interval
- **abs_tol** – absolute error tolerance
- **rel_tol** – relative error tolerance
- **n_init** – initial number of samples
- **n_max** – maximum number of samples
- **control_variates** (*list*) – list of integrand objects to be used as control variates. Control variates are currently only compatible with single level problems. The same discrete distribution instance must be used for the integrand and each of the control variates.
- **control_variate_means** (*list*) – list of means for each control variate

integrate()

See abstract method.

set_tolerance(*abs_tol=None*, *rel_tol=None*)

See abstract method.

Parameters

- **abs_tol** (*float*) – absolute tolerance. Reset if supplied, ignored if not.
- **rel_tol** (*float*) – relative tolerance. Reset if supplied, ignored if not.

4.4.8 CLT MC Cubature

class qmcpy.stopping_criterion.cub_mc_clt.**CubMCCLT**(*integrand*, *abs_tol=0.01*, *rel_tol=0.0*, *n_init=1024.0*, *n_max=10000000000.0*, *inflate=1.2*, *alpha=0.01*, *control_variates=[]*, *control_variate_means=[]*, *error_fun=<function CubMCCLT.<lambda>>*)

Stopping criterion based on the Central Limit Theorem.

```
>>> ao = AsianOption(IIDStdUniform(seed=7))
>>> sc = CubMCCLT(ao,abs_tol=.05)
>>> solution,data = sc.integrate()
>>> data
MeanVarData (AccumulateData Object)
    solution        1.519
    error_bound     0.046
    n_total         96028
    n               95004
    levels          1
    time_integrate  ...
CubMCCLT (StoppingCriterion Object)
    abs_tol         0.050
    rel_tol         0
    n_init          2^(10)
    n_max           10000000000
    inflate         1.200
    alpha           0.010
AsianOption (Integrand Object)
    volatility      2^(-1)
    call_put        call
    start_price     30
    strike_price    35
    interest_rate   0
    mean_type       arithmetic
    dim_frac        0
BrownianMotion (TrueMeasure Object)
    time_vec        1
    drift           0
    mean            0
    covariance      1
    decomp_type     PCA
IIDStdUniform (DiscreteDistribution Object)
    d               1
    entropy         7
    spawn_key       ()
>>> ao = AsianOption(IIDStdUniform(seed=7),multilevel_dims=[2,4,8])
>>> sc = CubMCCLT(ao,abs_tol=.05)
>>> solution,data = sc.integrate()
>>> dd = IIDStdUniform(1,seed=7)
>>> k = Keister(dd)
>>> cv1 = CustomFun(Uniform(dd),lambda x: sin(pi*x).sum(1))
>>> cv1mean = 2/pi
>>> cv2 = CustomFun(Uniform(dd),lambda x: (-3*(x-.5)**2+1).sum(1))
```

(continues on next page)

(continued from previous page)

```
>>> cv2mean = 3/4
>>> sc1 = CubMCCLT(k,abs_tol=.05,control_variates=[cv1,cv2],control_variate_
↪means=[cv1mean,cv2mean])
>>> sol,data = sc1.integrate()
>>> data
MeanVarData (AccumulateData Object)
    solution        1.381
    error_bound     0.010
    n_total         3072
    n               2^(11)
    levels          1
    time_integrate  ...
CubMCCLT (StoppingCriterion Object)
    abs_tol         0.050
    rel_tol         0
    n_init          2^(10)
    n_max           10000000000
    inflate         1.200
    alpha           0.010
Keister (Integrand Object)
Gaussian (TrueMeasure Object)
    mean            0
    covariance      2^(-1)
    decomp_type     PCA
IIDStdUniform (DiscreteDistribution Object)
    d               1
    entropy         7
    spawn_key       ()
```

__init__(*integrand*, *abs_tol=0.01*, *rel_tol=0.0*, *n_init=1024.0*, *n_max=10000000000.0*, *inflate=1.2*, *alpha=0.01*, *control_variates=[]*, *control_variate_means=[]*, *error_fun=<function CubMCCLT.<lambda>>*)

Parameters

- **integrand** (Integrand) – an instance of Integrand
- **inflate** (*float*) – inflation factor when estimating variance
- **alpha** (*float*) – significance level for confidence interval
- **abs_tol** (*ndarray*) – absolute error tolerance
- **rel_tol** (*ndarray*) – relative error tolerance
- **n_max** (*int*) – maximum number of samples
- **control_variates** (*list*) – list of integrand objects to be used as control variates. Control variates are currently only compatible with single level problems. The same discrete distribution instance must be used for the integrand and each of the control variates.
- **control_variate_means** (*list*) – list of means for each control variate

integrate()

See abstract method.

set_tolerance(*abs_tol=None*, *rel_tol=None*)

See abstract method.

Parameters

- **abs_tol** (*float*) – absolute tolerance. Reset if supplied, ignored if not.
- **rel_tol** (*float*) – relative tolerance. Reset if supplied, ignored if not.

4.4.9 Continuation Multilevel QMC Cubature

class qmcpy.stopping_criterion.cub_qmc_ml_cont.**CubQMCMLCont**(*integrand*, *abs_tol=0.05*, *alpha=0.01*, *rmse_tol=None*, *n_init=256.0*, *n_max=10000000000.0*, *replications=32.0*, *levels_min=2*, *levels_max=10*, *n_tols=10*, *tol_mult=1.6681005372000588*, *theta_init=0.5*)

Stopping criterion based on continuation multi-level quasi-Monte Carlo.

```
>>> mlco = MLCallOptions(Lattice(seed=7))
>>> sc = CubQMCMLCont(mlco,abs_tol=.05)
>>> solution,data = sc.integrate()
>>> data
MLQMCData (AccumulateData Object)
    solution        10.421
    n_total         98304
    n_level         [2048.  256.  256.  256.  256.]
    levels          5
    mean_level      [10.054  0.183  0.102  0.054  0.028]
    var_level       [2.027e-04 5.129e-05 2.656e-05 1.064e-05 3.466e-06]
    bias_estimate   0.016
    time_integrate  ...
CubQMCMLCont (StoppingCriterion Object)
    rmse_tol        0.019
    n_init          2^(8)
    n_max           10000000000
    replications    2^(5)
    levels_min      2^(1)
    levels_max      10
    n_tols          10
    tol_mult        1.668
    theta_init      2^(-1)
    theta           2^(-3)
MLCallOptions (Integrand Object)
    option          european
    sigma           0.200
    k               100
    r               0.050
    t               1
    b               85
    level           0
Gaussian (TrueMeasure Object)
```

(continues on next page)

(continued from previous page)

```
    mean            0
    covariance      1
    decomp_type     PCA
Lattice (DiscreteDistribution Object)
    d               1
    dvec            0
    randomize       1
    order           natural
    gen_vec         1
    entropy         7
    spawn_key       ()
```

References

[1] https://github.com/PieterjanRobbe/MultilevelEstimators.jl

__init__(*integrand*, *abs_tol=0.05*, *alpha=0.01*, *rmse_tol=None*, *n_init=256.0*, *n_max=10000000000.0*, *replications=32.0*, *levels_min=2*, *levels_max=10*, *n_tols=10*, *tol_mult=1.6681005372000588*, *theta_init=0.5*)

Parameters

- **integrand** (Integrand) – integrand with multi-level g method
- **abs_tol** (*float*) – absolute tolerance
- **alpha** (*float*) – uncertainty level. If rmse_tol not supplied, then rmse_tol = abs_tol/norm.ppf(1-alpha/2)
- **rmse_tol** (*float*) – root mean squared error If supplied (not None), then absolute tolerance and alpha are ignored in favor of the rmse tolerance
- **n_max** (*int*) – maximum number of samples
- **replications** (*int*) – number of replications on each level
- **levels_min** (*int*) – minimum level of refinement >= 2
- **levels_max** (*int*) – maximum level of refinement >= Lmin
- **n_tols** (*int*) – number of coarser tolerances to run
- **tol_mult** (*float*) – coarser tolerance multiplication factor
- **theta_init** (*float*) – initial error splitting constant

integrate()

ABSTRACT METHOD to determine the number of samples needed to satisfy the tolerance.

Returns

tuple containing:

- solution (float): approximation to the integral
- data (AccumulateData): an AccumulateData object

Return type
tuple

set_tolerance(*abs_tol=None*, *alpha=0.01*, *rmse_tol=None*)

See abstract method.

Parameters

- **integrand** (Integrand) – integrand with multi-level g method
- **abs_tol** (*float*) – absolute tolerance. Reset if supplied, ignored if not.
- **alpha** (*float*) – uncertainty level. If rmse_tol not supplied, then rmse_tol = abs_tol/norm.ppf(1-alpha/2)
- **rel_tol** (*float*) – relative tolerance. Reset if supplied, ignored if not. Takes priority over absolute tolerance and alpha if supplied.

4.4.10 Multilevel QMC Cubature

class qmcpy.stopping_criterion.cub_qmc_ml.**CubQMCML**(*integrand*, *abs_tol=0.05*, *alpha=0.01*, *rmse_tol=None*, *n_init=256.0*, *n_max=10000000000.0*, *replications=32.0*, *levels_min=2*, *levels_max=10*)

Stopping criterion based on multi-level quasi-Monte Carlo.

```
>>> mlco = MLCallOptions(Lattice(seed=7))
>>> sc = CubQMCML(mlco,abs_tol=.05)
>>> solution,data = sc.integrate()
>>> data
MLQMCData (AccumulateData Object)
    solution        10.434
    n_total         172032
    n_level         [4096.  256.  256.  256.  256.  256.]
    levels          6
    mean_level      [10.053  0.183  0.102  0.054  0.028  0.014]
    var_level       [5.699e-05 5.129e-05 2.656e-05 1.064e-05 3.466e-06 1.113e-06]
    bias_estimate   0.007
    time_integrate  ...
CubQMCML (StoppingCriterion Object)
    rmse_tol        0.019
    n_init          2^(8)
    n_max           10000000000
    replications    2^(5)
MLCallOptions (Integrand Object)
    option          european
    sigma           0.200
    k               100
    r               0.050
    t               1
    b               85
    level           0
Gaussian (TrueMeasure Object)
    mean            0
    covariance      1
    decomp_type     PCA
Lattice (DiscreteDistribution Object)
```

(continues on next page)

(continued from previous page)

```
    d               1
    dvec            0
    randomize       1
    order           natural
    gen_vec         1
    entropy         7
    spawn_key       ()
```

References

[1] M.B. Giles and B.J. Waterhouse. 'Multilevel quasi-Monte Carlo path simulation'. pp.165-181 in Advanced Financial Modelling, in Radon Series on Computational and Applied Mathematics, de Gruyter, 2009. http://people.maths.ox.ac.uk/~gilesm/files/radon.pdf

__init__(*integrand*, *abs_tol=0.05*, *alpha=0.01*, *rmse_tol=None*, *n_init=256.0*, *n_max=10000000000.0*, *replications=32.0*, *levels_min=2*, *levels_max=10*)

Parameters

- **integrand** (Integrand) – integrand with multi-level g method
- **abs_tol** (*float*) – absolute tolerance
- **alpha** (*float*) – uncertainty level. If rmse_tol not supplied, then rmse_tol = abs_tol/norm.ppf(1-alpha/2)
- **rmse_tol** (*float*) – root mean squared error If supplied (not None), then absolute tolerance and alpha are ignored in favor of the rmse tolerance
- **n_max** (*int*) – maximum number of samples
- **replications** (*int*) – number of replications on each level
- **levels_min** (*int*) – minimum level of refinement >= 2
- **levels_max** (*int*) – maximum level of refinement >= Lmin

integrate()

See abstract method.

set_tolerance(*abs_tol=None*, *alpha=0.01*, *rmse_tol=None*)

See abstract method.

Parameters

- **integrand** (Integrand) – integrand with multi-level g method
- **abs_tol** (*float*) – absolute tolerance. Reset if supplied, ignored if not.
- **alpha** (*float*) – uncertainty level. If rmse_tol not supplied, then rmse_tol = abs_tol/norm.ppf(1-alpha/2)
- **rel_tol** (*float*) – relative tolerance. Reset if supplied, ignored if not. Takes priority over absolute tolerance and alpha if supplied.

4.4.11 Continuation Multilevel MC Cubature

class qmcpy.stopping_criterion.cub_mc_ml_cont.**CubMCMLCont**(*integrand*, *abs_tol=0.05*, *alpha=0.01*, *rmse_tol=None*, *n_init=256.0*, *n_max=10000000000.0*, *levels_min=2*, *levels_max=10*, *n_tols=10*, *tol_mult=1.6681005372000588*, *theta_init=0.5*)

Stopping criterion based on continuation multi-level monte carlo.

```
>>> mlco = MLCallOptions(IIDStdUniform(seed=7))
>>> sc = CubMCMLCont(mlco,abs_tol=.05)
>>> solution,data = sc.integrate()
>>> data
MLMCData (AccumulateData Object)
    solution        10.400
    n_total         1193331
    levels          2^(2)
    n_level         [1133772.   22940.    8676.    2850.]
    mean_level      [10.059  0.186  0.105  0.05 ]
    var_level       [1.959e+02 1.603e-01 4.567e-02 1.013e-02]
    cost_per_sample [1. 2. 4. 8.]
    alpha           0.942
    beta            1.992
    gamma           1.000
    time_integrate  ...
CubMCMLCont (StoppingCriterion Object)
    rmse_tol        0.019
    n_init          2^(8)
    levels_min      2^(1)
    levels_max      10
    n_tols          10
    tol_mult        1.668
    theta_init      2^(-1)
    theta           2^(-1)
MLCallOptions (Integrand Object)
    option          european
    sigma           0.200
    k               100
    r               0.050
    t               1
    b               85
    level           0
Gaussian (TrueMeasure Object)
    mean            0
    covariance      1
    decomp_type     PCA
IIDStdUniform (DiscreteDistribution Object)
    d               1
    entropy         7
    spawn_key       ()
```

References

[1] https://github.com/PieterjanRobbe/MultilevelEstimators.jl

__init__(*integrand*, *abs_tol=0.05*, *alpha=0.01*, *rmse_tol=None*, *n_init=256.0*, *n_max=10000000000.0*, *levels_min=2*, *levels_max=10*, *n_tols=10*, *tol_mult=1.6681005372000588*, *theta_init=0.5*)

Parameters

- **integrand** (*Integrand*) – integrand with multi-level g method
- **abs_tol** (*float*) – absolute tolerance. Reset if supplied, ignored if not.
- **alpha** (*float*) – uncertainty level. If rmse_tol not supplied, then rmse_tol = abs_tol/norm.ppf(1-alpha/2)
- **rel_tol** (*float*) – relative tolerance. Reset if supplied, ignored if not. Takes priority over absolute tolerance and alpha if supplied.
- **n_init** (*int*) – initial number of samples
- **n_max** (*int*) – maximum number of samples
- **levels_min** (*int*) – minimum level of refinement >= 2
- **levels_max** (*int*) – maximum level of refinement >= Lmin
- **n_tols** (*int*) – number of coarser tolerances to run
- **tol_mult** (*float*) – coarser tolerance multiplication factor
- **theta_init** (*float*) – initial error splitting constant

integrate()

ABSTRACT METHOD to determine the number of samples needed to satisfy the tolerance.

Returns

tuple containing:

- solution (float): approximation to the integral
- data (AccumulateData): an AccumulateData object

Return type
tuple

set_tolerance(*abs_tol=None*, *alpha=0.01*, *rmse_tol=None*)

See abstract method.

Parameters

- **abs_tol** (*float*) – absolute tolerance. Reset if supplied, ignored if not.
- **alpha** (*float*) – uncertainty level. If rmse_tol not supplied, then rmse_tol = abs_tol/norm.ppf(1-alpha/2)
- **rel_tol** (*float*) – relative tolerance. Reset if supplied, ignored if not. Takes priority over absolute tolerance and alpha if supplied.

4.4.12 Multilevel MC Cubature

class qmcpy.stopping_criterion.cub_mc_ml.**CubMCML**(*integrand*, *abs_tol=0.05*, *alpha=0.01*, *rmse_tol=None*, *n_init=256.0*, *n_max=10000000000.0*, *levels_min=2*, *levels_max=10*, *alpha0=- 1.0*, *beta0=- 1.0*, *gamma0=- 1.0*)

Stopping criterion based on multi-level monte carlo.

```
>>> mlco = MLCallOptions(IIDStdUniform(seed=7))
>>> sc = CubMCML(mlco,abs_tol=.05)
>>> solution,data = sc.integrate()
>>> data
MLMCData (AccumulateData Object)
    solution        10.450
    n_total         1213658
    levels          7
    n_level         [1.173e+06 2.369e+04 1.174e+04 3.314e+03 1.144e+03 4.380e+02 1.
↪690e+02]
    mean_level      [1.006e+01 1.856e-01 1.053e-01 5.127e-02 2.699e-02 1.558e-02 7.
↪068e-03]
    var_level       [1.958e+02 1.596e-01 4.603e-02 1.057e-02 2.978e-03 8.701e-04 2.
↪552e-04]
    cost_per_sample [ 1.  2.  4.  8. 16. 32. 64.]
    alpha           0.936
    beta            1.870
    gamma           1.000
    time_integrate  ...
CubMCML (StoppingCriterion Object)
    rmse_tol        0.019
    n_init          2^(8)
    levels_min      2^(1)
    levels_max      10
    theta           2^(-1)
MLCallOptions (Integrand Object)
    option          european
    sigma           0.200
    k               100
    r               0.050
    t               1
    b               85
    level           0
Gaussian (TrueMeasure Object)
    mean            0
    covariance      1
    decomp_type     PCA
IIDStdUniform (DiscreteDistribution Object)
    d               1
    entropy         7
    spawn_key       ()
```

Original Implementation:

http://people.maths.ox.ac.uk/~gilesm/mlmc/#MATLAB

References

[1] M.B. Giles. 'Multi-level Monte Carlo path simulation'. Operations Research, 56(3):607-617, 2008. http://people.maths.ox.ac.uk/~gilesm/files/OPRE_2008.pdf.

__init__(*integrand*, *abs_tol=0.05*, *alpha=0.01*, *rmse_tol=None*, *n_init=256.0*, *n_max=10000000000.0*, *levels_min=2*, *levels_max=10*, *alpha0=- 1.0*, *beta0=- 1.0*, *gamma0=- 1.0*)

Parameters

- **integrand** (Integrand) – integrand with multi-level g method
- **abs_tol** (*float*) – absolute tolerance. Reset if supplied, ignored if not.
- **alpha** (*float*) – uncertainty level. If rmse_tol not supplied, then rmse_tol = abs_tol/norm.ppf(1-alpha/2)
- **rel_tol** (*float*) – relative tolerance. Reset if supplied, ignored if not. Takes priority over absolute tolerance and alpha if supplied.
- **n_init** (*int*) – initial number of samples
- **n_max** (*int*) – maximum number of samples
- **levels_min** (*int*) – minimum level of refinement >= 2
- **levels_max** (*int*) – maximum level of refinement >= Lmin
- **alpha0** (*float*) – weak error is O(2^{-alpha0*level})
- **beta0** (*float*) – variance is O(2^{-bet0a*level})
- **gamma0** (*float*) – sample cost is O(2^{gamma0*level})

Note: if alpha, beta, gamma are not positive, then they will be estimated

---

integrate()

See abstract method.

set_tolerance(*abs_tol=None*, *alpha=0.01*, *rmse_tol=None*)

See abstract method.

Parameters

- **abs_tol** (*float*) – absolute tolerance. Reset if supplied, ignored if not.
- **alpha** (*float*) – uncertainty level. If rmse_tol not supplied, then rmse_tol = abs_tol/norm.ppf(1-alpha/2)
- **rel_tol** (*float*) – relative tolerance. Reset if supplied, ignored if not. Takes priority over absolute tolerance and alpha if supplied.

4.4.13 Probability of Failure with Guassian Processes

class qmcpy.stopping_criterion.pf_gp_ci.**PFGPCI**(*integrand*, *failure_threshold*, *failure_above_threshold*, *abs_tol=0.005*, *alpha=0.01*, *n_init=64*, *init_samples=None*, *batch_sampler=<qmcpy.stopping_criterion.pf_gp_ci.PFSampleErrorDen object>*, *n_batch=4*, *n_max=1000*, *n_approx=1048576*, *gpytorch_prior_mean=ZeroMean()*, *gpytorch_prior_cov=ScaleKernel((base_kernel): MaternKernel((raw_lengthscale_constraint): Positive()) (raw_outputscale_constraint): Positive())*, *gpytorch_likelihood=GaussianLikelihood((noise_covar): HomoskedasticNoise((raw_noise_constraint): Interval(1.000E-12, 1.000E-08)))*, *gpytorch_marginal_log_likelihood_func=<function PFGPCI.<lambda>>*, *torch_optimizer_func=<function PFGPCI.<lambda>>*, *gpytorch_train_iter=100*, *gpytorch_use_gpu=False*, *verbose=False*, *n_ref_approx=4194304*, *seed_ref_approx=None*)

Probability of failure estimation using adaptive Gaussian Processes (GP) construction and resulting credible intervals.

```
>>> pfgpci = PFGPCI(
...     integrand = Ishigami(DigitalNetB2(3,seed=17)),
...     failure_threshold = 0,
...     failure_above_threshold = False,
...     abs_tol = 2.5e-2,
...     alpha = 1e-1,
...     n_init = 64,
...     init_samples = None,
...     batch_sampler = PFSampleErrorDensityAR(verbose=False),
...     n_batch = 16,
...     n_max = 128,
...     n_approx = 2**18,
...     gpytorch_prior_mean = gpytorch.means.ZeroMean(),
...     gpytorch_prior_cov = gpytorch.kernels.ScaleKernel(
...         gpytorch.kernels.MaternKernel(nu=2.5,lengthscale_constraint = gpytorch.
↪constraints.Interval(.5,1)),
...         outputscale_constraint = gpytorch.constraints.Interval(1e-8,.5)),
...     gpytorch_likelihood = gpytorch.likelihoods.GaussianLikelihood(noise_
↪constraint = gpytorch.constraints.Interval(1e-12,1e-8)),
...     gpytorch_marginal_log_likelihood_func = lambda likelihood,gpyt_model:␣
↪gpytorch.mlls.ExactMarginalLogLikelihood(likelihood,gpyt_model),
...     torch_optimizer_func = lambda gpyt_model: torch.optim.Adam(gpyt_model.
↪parameters(),lr=0.1),
...     gpytorch_train_iter = 100,
...     gpytorch_use_gpu = False,
...     verbose = False,
...     n_ref_approx = 2**22,
...     seed_ref_approx = 11)
```

(continues on next page)

(continued from previous page)

```
>>> solution,data = pfgpci.integrate(seed=7,refit=True)
>>> data
PFGPCIData (AccumulateData Object)
    solution        0.161
    error_bound     0.025
    bound_low       0.136
    bound_high      0.186
    n_total         112
    time_integrate  ...
PFGPCI (StoppingCriterion Object)
Ishigami (Integrand Object)
Uniform (TrueMeasure Object)
    lower_bound     -3.142
    upper_bound     3.142
DigitalNetB2 (DiscreteDistribution Object)
    d               3
    dvec            [0 1 2]
    randomize       LMS_DS
    graycode        0
    entropy         17
    spawn_key       ()
>>> df = data.get_results_dict()
```

__init__(*integrand*, *failure_threshold*, *failure_above_threshold*, *abs_tol=0.005*, *alpha=0.01*, *n_init=64*, *init_samples=None*, *batch_sampler=<qmcpy.stopping_criterion.pf_gp_ci.PFSampleErrorDensityAR object>*, *n_batch=4*, *n_max=1000*, *n_approx=1048576*, *gpytorch_prior_mean=ZeroMean()*, *gpytorch_prior_cov=ScaleKernel((base_kernel): MaternKernel((raw_lengthscale_constraint): Positive()) (raw_outputscale_constraint): Positive())*, *gpytorch_likelihood=GaussianLikelihood((noise_covar): HomoskedasticNoise((raw_noise_constraint): Interval(1.000E-12, 1.000E-08)))*, *gpytorch_marginal_log_likelihood_func=<function PFGPCI.<lambda>>*, *torch_optimizer_func=<function PFGPCI.<lambda>>*, *gpytorch_train_iter=100*, *gpytorch_use_gpu=False*, *verbose=False*, *n_ref_approx=4194304*, *seed_ref_approx=None*)

Parameters

- **integrand** (*Integrand*) – The simulation whose probability of failure is estimated
- **failure_threshold** (*float*) – Thresholds for failure.
- **failure_above_threshold** (*bool*) – Set to True if failure occurs when the simulation exceeds failure_threshold and False otherwise
- **abs_tol** (*float*) – The desired maximum distance from the estimate to either end of the confidence interval
- **alpha** (*float*) – The credible interval is constructed to hold with probability at least 1 - alpha
- **n_init** (*float*) – Initial number of samples from integrand.discrete_distrib from which to build the first surrogate GP
- **init_samples** (*float*) – If the simulation has already been run, pass in (x,y) where x are past samples from the discrete distribution and y are corresponding simulation evaluations.
- **batch_sampler** (*Suggester or DiscreteDistsribution*) – A suggestion scheme for future samples.

- **n_batch** (*int*) – The number of samples per batch to draw from batch_sampler.
- **n_max** (*int*) – Budget of simulations.
- **n_approx** (*int*) – Number of points from integrand.discrete_distrib used to approximate estimate and credible interval bounds
- **gpytorch_prior_mean** (*gpytorch.means*) – prior mean function of the GP
- **gpytorch_prior_cov** (*gpytorch.kernels*) – Prior covariance kernel of the GP
- **gpytorch_likelihood** (*gpytorch.likelihoods*) – GP likelihood, require one of gpytorch.likelihoods.{GaussianLikelihood, GaussianLikelihoodWithMissingObs, FixedNoiseGaussianLikelihood}
- **gpytorch_marginal_log_likelihood_func** (*callable*) – Function taking in the likelihood and gpytorch model and returning a marginal log likelihood from gpytorch.mlls
- **torch_optimizer_func** (*callable*) – Function taking in the gpytorch model and returning an optimizer from torch.optim
- **gpytorch_train_iter** (*int*) – Triaining iterations for the GP in gpytorch
- **gpytorch_use_gpu** (*bool*) – If True, have gpytorch use a GPU for fitting and trining the GP
- **verbose** (*int*) – If verbose > 0, print information throught the call to integrate()
- **n_ref_approx** (*int*) – If n_ref_approx > 0, use n_ref_approx points to get a reference QMC approximation of the true solution. Caution: If n_ref_approx > 0, it should be a large int e.g. 2**22, in which case it is only helpful for cheap to evaluate simulations
- **seed_ref_approx** (*int*) – Seed for the reference aproximation. Only applies when n_ref_approx>0

integrate(*seed=None*, *refit=False*)

ABSTRACT METHOD to determine the number of samples needed to satisfy the tolerance.

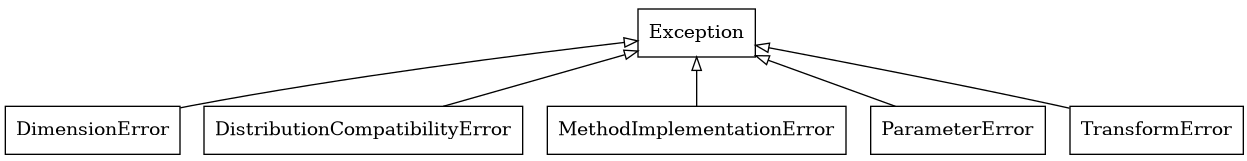
Returns

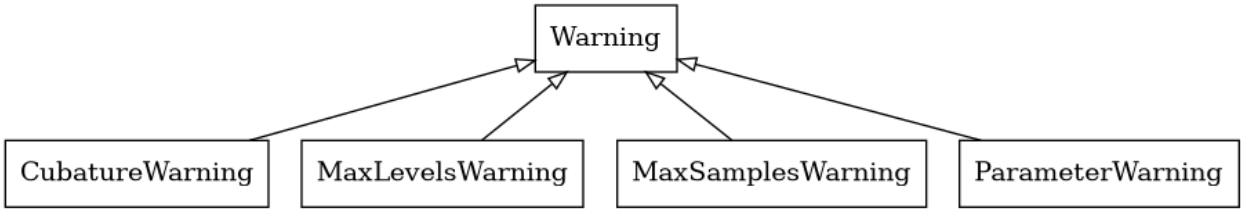
tuple containing:

- solution (float): approximation to the integral
- data (AccumulateData): an AccumulateData object

Return type
tuple

4.5 Utilities





qmcpy.util.latnetbuilder_linker.**latnetbuilder_linker**(*lnb_dir='./'*, *out_dir='./'*, *fout_prefix='lnb4qmcpy'*)

Parameters

- **lnb_dir** (*str*) – relative path to directory where *outputMachine.txt* is stored e.g. 'my_lnb/poly_lat/'
- **out_dir** (*str*) – relative path to directory where output should be stored e.g. 'my_lnb/poly_lat_qmcpy/'
- **fout_prefix** (*str*) – start of output file name. e.g. 'my_poly_lat_vec'

Returns

path to file which can be passed into QMCPy's Lattice or Sobol' in order to use
the linked latnetbuilder generating vector/matrix e.g. 'my_poly_lat_vec.10.16.npy'

Return type
str

Adapted from latnetbuilder parser:
https://github.com/umontreal-simul/latnetbuilder/blob/master/python-wrapper/latnetbuilder/parse_output.py#L74

CHAPTER

FIVE

DEMOS

5.1 A QMCPy Quick Start

In this tutorial, we introduce QMCPy [1] by an example. QMCPy can be installed with **pip install qmcpy** or cloned from the QMCSoftware GitHub repository.

Consider the problem of integrating the Keister function [2] with respect to a d-dimensional Gaussian measure:

$$f(\boldsymbol{x}) = \pi^{d/2} \cos(||\boldsymbol{x}||), \qquad \boldsymbol{x} \in \mathbb{R}^d, \qquad \boldsymbol{X} \sim \mathcal{N}(\boldsymbol{0}_d, \mathsf{I}_d/2),$$
$$\mu = \mathbb{E}[f(\boldsymbol{X})] := \int_{\mathbb{R}^d} f(\boldsymbol{x})\, \pi^{-d/2} \exp(-||\boldsymbol{x}||^2)\, \mathrm{d}\boldsymbol{x}$$
$$= \int_{[0,1]^d} \pi^{d/2} \cos\left(\sqrt{\frac{1}{2}\sum_{j=1}^{d} \Phi^{-1}(x_j)}\right) \mathrm{d}\boldsymbol{x},$$

where $||\boldsymbol{x}||$ is the Euclidean norm, I_d is the d-dimensional identity matrix, and Φ denotes the standard normal cumulative distribution function. When $d = 2$, $\mu \approx 1.80819$ and we can visualize the Keister function and realizations of the sampling points depending on the tolerance values, ε, in the following figure:

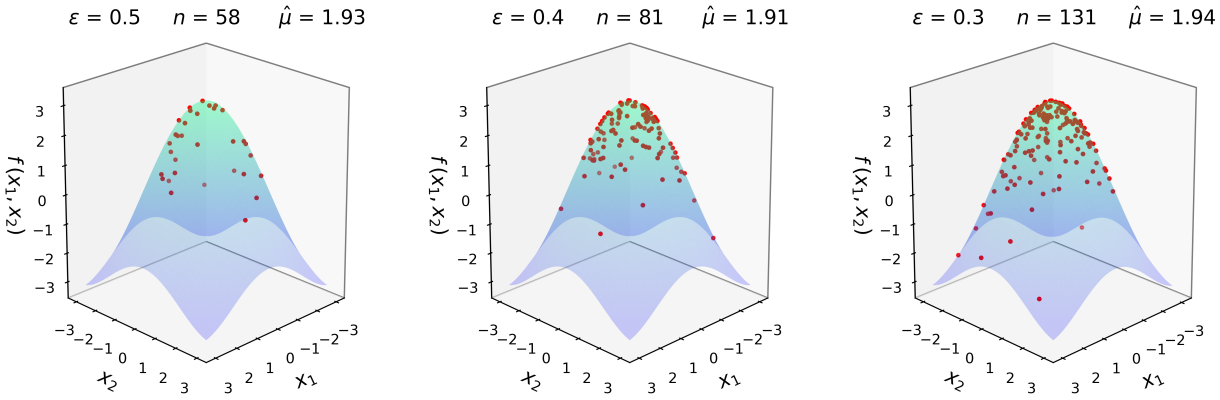


Fig. 1: Keister Function

The Keister function is implemented below with help from NumPy [3] in the following code snippet:

```
import numpy as np
def keister(x):
    """
```

(continues on next page)

(continued from previous page)

```
    x: nxd numpy ndarray
       n samples
       d dimensions

    returns n-vector of the Kesiter function
    evaluated at the n input samples
    """
    d = x.shape[1]
    norm_x = np.sqrt((x**2).sum(1))
    k = np.pi**(d/2) * np.cos(norm_x)
    return k # size n vector
```

In addition to our Keister integrand and Gaussian true measure, we must select a discrete distribution, and a stopping criterion [4]. The stopping criterion determines the number of points at which to evaluate the integrand in order for the mean approximation to be accurate within a user-specified error tolerance, ε. The discrete distribution determines the sites at which the integrand is evaluated.

For this Keister example, we select the lattice sequence as the discrete distribution and corresponding cubature-based stopping criterion [5]. The discrete distribution, true measure, integrand, and stopping criterion are then constructed within the QMCPy framework below.

```
import qmcpy
d = 2
discrete_distrib = qmcpy.Lattice(dimension = d)
true_measure = qmcpy.Gaussian(discrete_distrib, mean = 0, covariance = 1/2)
integrand = qmcpy.CustomFun(true_measure,keister)
stopping_criterion = qmcpy.CubQMCLatticeG(integrand = integrand, abs_tol = 1e-3)
```

Calling *integrate* on the *stopping_criterion* instance returns the numerical solution and a data object. Printing the data object will provide a neat summary of the integration problem. For details of the output fields, refer to the online, searchable QMCPy Documentation at https://qmcpy.readthedocs.io/.

```
solution, data = stopping_criterion.integrate()
print(data)
```

```
LDTransformData (AccumulateData Object)
    solution        1.808
    comb_bound_low  1.808
    comb_bound_high 1.809
    comb_flags      1
    n_total         2^(13)
    n               2^(13)
    time_integrate  0.017
CubQMCLatticeG (StoppingCriterion Object)
    abs_tol         0.001
    rel_tol         0
    n_init          2^(10)
    n_max           2^(35)
CustomFun (Integrand Object)
Gaussian (TrueMeasure Object)
    mean            0
    covariance      2^(-1)
    decomp_type     PCA
```

(continues on next page)

(continued from previous page)

```
Lattice (DiscreteDistribution Object)
    d               2^(1)
    dvec            [0 1]
    randomize       1
    order           natural
    entropy         273562359450377681412227949180408652150
    spawn_key       ()
```

This guide is not meant to be exhaustive but rather a quick introduction to the QMCPy framework and syntax. In an upcoming blog, we will take a closer look at low-discrepancy sequences such as the lattice sequence from the above example.

5.1.1 References

1. Choi, S.-C. T., Hickernell, F., McCourt, M., Rathinavel J., & Sorokin, A. QMCPy: A quasi-Monte Carlo Python Library. https://qmcsoftware.github.io/QMCSoftware/. 2020.
2. Keister, B. D. Multidimensional Quadrature Algorithms. Computers in Physics 10, 119–122 (1996).
3. Oliphant, T., Guide to NumPy https://ecs.wgtn.ac.nz/foswiki/pub/Support/ManualPagesAndDocumentation/numpybook.pdf (Trelgol Publishing USA, 2006).
4. Hickernell, F., Choi, S.-C. T., Jiang, L. & Jimenez Rugama, L. A. in WileyStatsRef-Statistics Reference Online (eds Davidian, M.et al.) (John Wiley & Sons Ltd., 2018).
5. Jimenez Rugama, L. A. & Hickernell, F. Adaptive Multidimensional Inte-gration Based on Rank-1 Lattices in Monte Carlo and Quasi-Monte Carlo Methods: MCQMC, Leuven, Belgium, April 2014 (eds Cools, R. & Nuyens, D.) 163.arXiv:1411.1966 (Springer-Verlag, Berlin, 2016), 407–422.

5.2 Welcome to QMCPy

5.2.1 Importing QMCPy

Here we show three different ways to import QMCPy in a Python environment. First, we can import the package `qmcpy` under the alias `qp`.

```
import qmcpy as qp
print(qp.name, qp.__version__)
```

```
qmcpy 1.3
```

Alternatively, we can import individual objects from 'qmcpy' as shown below.

```
from qmcpy.integrand import *
from qmcpy.true_measure import *
from qmcpy.discrete_distribution import *
from qmcpy.stopping_criterion import *
```

Lastly, we can import all objects from the package using an asterisk.

```
from qmcpy import *
```

5.2.2 Important Notes

IID vs LDS

Low discrepancy (LD) sequences such as lattice and Sobol' are not independent like IID (independent identically distributed) points.

The code below generates 4 Sobol samples of 2 dimensions.

```
distribution = Lattice(dimension=2, randomize=True, seed=7)
distribution.gen_samples(n_min=0,n_max=4)
```

```
array([[0.04386058, 0.58727432],
       [0.54386058, 0.08727432],
       [0.29386058, 0.33727432],
       [0.79386058, 0.83727432]])
```

Multi-Dimensional Inputs

Suppose we want to create an integrand in QMCPy for evaluating the following integral:

$$\int_{[0,1]^d} \|x\|_2^{\|x\|_2^{1/2}} dx,$$

where $[0,1]^d$ is the unit hypercube in $\mathbb{R}^d$. The integrand is defined everywhere except at $x = 0$ and hence the definite integral is also defined.

The key in defining a Python function of an integrand in the QMCPy framework is that not only it should be able to take one point $x \in \mathbb{R}^d$ and return a real value, but also that it would be able to take a set of n sampling points as rows in a Numpy array of size $n \times d$ and return an array with n values evaluated at each sampling point. The following examples illustrate this point.

```
from numpy.linalg import norm as norm
from numpy import sqrt, array
```

Our first attempt maybe to create the integrand as a Python function as follows:

```
def f(x): return norm(x) ** sqrt(norm(x))
```

It looks reasonable except that maybe the Numpy function norm is executed twice. It's okay for now. Let us quickly test if the function behaves as expected at a point value:

```
x = 0.01
f(x)
```

```
0.6309573444801932
```

What about an array that represents $n = 3$ sampling points in a two-dimensional domain, i.e., $d = 2$?

```
x = array([[1., 0.],
           [0., 0.01],
           [0.04, 0.04]])
f(x)
```

```
1.001650000560437
```

Now, the function should have returned $n = 3$ real values that corresponding to each of the sampling points. Let's debug our Python function.

```
norm(x)
```

```
1.0016486409914407
```

Numpy's `norm(x)` is obviously a matrix norm, but we want it to be vector 2-norm that acts on each row of `x`. To that end, let's add an axis argument to the function:

```
norm(x, axis = 1)
```

```
array([1.        , 0.01      , 0.05656854])
```

Now it's working! Let's make sure that the `sqrt` function is acting on each element of the vector norm results:

```
sqrt(norm(x, axis = 1))
```

```
array([1.        , 0.1       , 0.23784142])
```

It is. Putting everything together, we have:

```
norm(x, axis = 1) ** sqrt(norm(x, axis = 1))
```

```
array([1.        , 0.63095734, 0.50502242])
```

We have got our proper function definition now.

```
def myfunc(x):
    x_norms = norm(x, axis = 1)
    return x_norms ** sqrt(x_norms)
```

We can now create an `integrand` instance with our `QuickConstruct` class in QMCPy and then invoke QMCPy's `integrate` function:

```
dim = 1
abs_tol = .01
integrand = CustomFun(
    true_measure = Uniform(IIDStdUniform(dimension=dim, seed=7)),
    g=myfunc)
solution,data = CubMCCLT(integrand,abs_tol=abs_tol,rel_tol=0).integrate()
print(data)
```

```
MeanVarData (AccumulateData Object)
    solution        0.659
    error_bound     0.010
    n_total         4400
    n               3376
    levels          1
    time_integrate  0.001
```

(continues on next page)

(continued from previous page)

```
CubMCCLT (StoppingCriterion Object)
    abs_tol         0.010
    rel_tol         0
    n_init          2^(10)
    n_max           10000000000
    inflate         1.200
    alpha           0.010
CustomFun (Integrand Object)
Uniform (TrueMeasure Object)
    lower_bound     0
    upper_bound     1
IIDStdUniform (DiscreteDistribution Object)
    d               1
    entropy         7
    spawn_key       ()
```

For our integral, we know the true value. Let's check if QMCPy's solution is accurate enough:

```
true_sol = 0.658582  # In WolframAlpha: Integral[x**Sqrt[x], {x,0,1}]
abs_tol = data.stopping_crit.abs_tol
qmcpy_error = abs(true_sol - solution)
if qmcpy_error > abs_tol: raise Exception("Error not within bounds")
```

It's good. Shall we test the function with $d = 2$ by simply changing the input parameter value of dimension for QuickConstruct?

```
dim = 2
integrand = CustomFun(
    true_measure = Uniform(IIDStdUniform(dimension=dim, seed=7)),
    g = myfunc)
solution2,data2 = CubMCCLT(integrand,abs_tol=abs_tol,rel_tol=0).integrate()
print(data2)
```

```
MeanVarData (AccumulateData Object)
    solution        0.831
    error_bound     0.010
    n_total         6659
    n               5635
    levels          1
    time_integrate  0.002
CubMCCLT (StoppingCriterion Object)
    abs_tol         0.010
    rel_tol         0
    n_init          2^(10)
    n_max           10000000000
    inflate         1.200
    alpha           0.010
CustomFun (Integrand Object)
Uniform (TrueMeasure Object)
    lower_bound     0
    upper_bound     1
IIDStdUniform (DiscreteDistribution Object)
```

(continues on next page)

(continued from previous page)

```
    d               2^(1)
    entropy         7
    spawn_key       ()
```

Once again, we could test for accuracy of QMCPy with respect to the true value:

```
true_sol2 = 0.827606  # In WolframAlpha:␣
↪Integral[Sqrt[x**2+y**2])**Sqrt[Sqrt[x**2+y**2]], {x,0,1}, {y,0,1}]
abs_tol2 = data2.stopping_crit.abs_tol
qmcpy_error2 = abs(true_sol2 - solution2)
if qmcpy_error2 > abs_tol2: raise Exception("Error not within bounds")
```

5.3 Integration Examples using QMCPy package

In this demo, we show how to use `qmcpy` for performing numerical multiple integration of two built-in integrands, namely, the Keister function and the Asian call option payoff. To start, we import the `qmcpy` module and the function `arrange()` from `numpy` for generating evenly spaced discrete vectors in the examples.

```
from qmcpy import *
from numpy import arange
```

5.3.1 Keister Example

We recall briefly the mathematical definitions of the Keister function, the Gaussian measure, and the Sobol distribution:

- Keister integrand: $y_j = \pi^{d/2}\cos(||x_j||_2)$
- Gaussian true measure: $\mathcal{N}(0, \frac{1}{2})$
- Sobol discrete distribution: $x_j \overset{LD}{\sim} \mathcal{U}(0,1)$

```
integrand = Keister(Sobol(dimension=3,seed=7))
solution,data = CubQMCSobolG(integrand,abs_tol=.05).integrate()
print(data)
```

```
LDTransformData (AccumulateData Object)
    solution        2.170
    comb_bound_low  2.159
    comb_bound_high 2.181
    comb_flags      1
    n_total         2^(10)
    n               2^(10)
    time_integrate  0.002
CubQMCSobolG (StoppingCriterion Object)
    abs_tol         0.050
    rel_tol         0
    n_init          2^(10)
    n_max           2^(35)
Keister (Integrand Object)
Gaussian (TrueMeasure Object)
```

(continues on next page)

(continued from previous page)

```
    mean            0
    covariance      2^(-1)
    decomp_type     PCA
Sobol (DiscreteDistribution Object)
    d               3
    dvec            [0 1 2]
    randomize       LMS_DS
    graycode        0
    entropy         7
    spawn_key       ()
```

5.3.2 Arithmetic-Mean Asian Put Option: Single Level

In this example, we want to estimate the payoff of an European Asian put option that matures at time T. The key mathematical entities are defined as follows:

- Stock price at time $t_j := jT/d$ for $j = 1, \dots, d$ is a function of its initial price $S(0)$, interest rate r, and volatility σ: $S(t_j) = S(0)e^{\left(r-\frac{\sigma^2}{2}\right)t_j+\sigma\mathcal{B}(t_j)}$
- Discounted put option payoff is defined as the difference of a fixed strike price K and the arithmetic average of the underlying stock prices at d discrete time intervals in $[0,T]$: $\max\left(K-\frac{1}{d}\sum_{j=1}^{d} S(t_j), 0\right)e^{-rT}$
- Brownian motion true measure: $\mathcal{B}(t_j) = \mathcal{B}(t_{j-1}) + Z_j\sqrt{t_j - t_{j-1}}$ for $Z_j \sim \mathcal{N}(0,1)$
- Lattice discrete distribution: $x_j \overset{LD}{\sim} \mathcal{U}(0,1)$

```
integrand = AsianOption(
    sampler = IIDStdUniform(dimension=16, seed=7),
    volatility = 0.5,
    start_price = 30,
    strike_price = 25,
    interest_rate = 0.01,
    mean_type = 'arithmetic')
solution,data = CubMCCLT(integrand, abs_tol=.025).integrate()
print(data)
```

```
MeanVarData (AccumulateData Object)
    solution        6.257
    error_bound     0.025
    n_total         889904
    n               888880
    levels          1
    time_integrate  1.303
CubMCCLT (StoppingCriterion Object)
    abs_tol         0.025
    rel_tol         0
    n_init          2^(10)
    n_max           10000000000
    inflate         1.200
    alpha           0.010
AsianOption (Integrand Object)
```

(continues on next page)

(continued from previous page)

```
    volatility      2^(-1)
    call_put        call
    start_price     30
    strike_price    25
    interest_rate   0.010
    mean_type       arithmetic
    dim_frac        0
BrownianMotion (TrueMeasure Object)
    time_vec        [0.062 0.125 0.188 ... 0.875 0.938 1.   ]
    drift           0
    mean            [0. 0. 0. ... 0. 0. 0.]
    covariance      [[0.062 0.062 0.062 ... 0.062 0.062 0.062]
                    [0.062 0.125 0.125 ... 0.125 0.125 0.125]
                    [0.062 0.125 0.188 ... 0.188 0.188 0.188]
                    ...
                    [0.062 0.125 0.188 ... 0.875 0.875 0.875]
                    [0.062 0.125 0.188 ... 0.875 0.938 0.938]
                    [0.062 0.125 0.188 ... 0.875 0.938 1.   ]]
    decomp_type     PCA
IIDStdUniform (DiscreteDistribution Object)
    d               2^(4)
    entropy         7
    spawn_key       ()
```

5.3.3 Arithmetic-Mean Asian Put Option: Multi-Level

This example is similar to the last one except that we use a multi-level method for estimation of the option price. The main idea can be summarized as follows:

$Y_0 = 0$

$Y_1 =$ Asian option monitored at $t = [\frac{1}{4}, \frac{1}{2}, \frac{3}{4}, 1]$

$Y_2 =$ Asian option monitored at $t = [\frac{1}{16}, \frac{1}{8}, ..., 1]$

$Y_3 =$ Asian option monitored at $t = [\frac{1}{64}, \frac{1}{32}, ..., 1]$

$Z_1 = \mathbb{E}[Y_1 - Y_0] + \mathbb{E}[Y_2 - Y_1] + \mathbb{E}[Y_3 - Y_2] = \mathbb{E}[Y_3]$

```
integrand = AsianOption(
        sampler = IIDStdUniform(seed=7),
        volatility = 0.5,
        start_price = 30,
        strike_price = 25,
        interest_rate = 0.01,
        mean_type = 'arithmetic',
        multilevel_dims = [4,8,16])
solution,data = CubMCCLT(integrand, abs_tol=.025).integrate()
print(data)
```

```
MeanVarData (AccumulateData Object)
    solution        6.264
    error_bound     0.025
```

(continues on next page)

(continued from previous page)

```
    n_total         1938085
    n               [1875751.   31235.   28027.]
    levels          3
    time_integrate  0.879
CubMCCLT (StoppingCriterion Object)
    abs_tol         0.025
    rel_tol         0
    n_init          2^(10)
    n_max           10000000000
    inflate         1.200
    alpha           0.010
AsianOption (Integrand Object)
    volatility      2^(-1)
    call_put        call
    start_price     30
    strike_price    25
    interest_rate   0.010
    mean_type       arithmetic
    multilevel_dims [ 4  8 16]
BrownianMotion (TrueMeasure Object)
    time_vec        1
    drift           0
    mean            0
    covariance      1
    decomp_type     PCA
IIDStdUniform (DiscreteDistribution Object)
    d               1
    entropy         7
    spawn_key       ()
```

5.3.4 Keister Example using Bayesian Cubature

This examples repeats the Keister using cubBayesLatticeG and cubBayesNetG stopping criterion:

- Keister integrand: $y_j = \pi^{d/2}\cos(||x_j||_2)$
- Gaussian true measure: $\mathcal{N}(0, \frac{1}{2})$
- Lattice discrete distribution: $x_j \overset{LD}{\sim} \mathcal{U}(0,1)$

```
dimension=3
abs_tol=.001
integrand = Keister(Lattice(dimension=dimension, order='linear'))
solution,data = CubBayesLatticeG(integrand,abs_tol=abs_tol).integrate()
print(data)
```

```
LDTransformBayesData (AccumulateData Object)
    solution        2.168
    comb_bound_low  2.167
    comb_bound_high 2.169
    comb_flags      1
    n_total         2^(12)
```

(continues on next page)

(continued from previous page)

```
    n               2^(12)
    time_integrate  0.044
CubBayesLatticeG (StoppingCriterion Object)
    abs_tol         0.001
    rel_tol         0
    n_init          2^(8)
    n_max           2^(22)
    order           2^(1)
Keister (Integrand Object)
Gaussian (TrueMeasure Object)
    mean            0
    covariance      2^(-1)
    decomp_type     PCA
Lattice (DiscreteDistribution Object)
    d               3
    dvec            [0 1 2]
    randomize       1
    order           linear
    entropy         375332914484089177125958786096311032 2
    spawn_key       ()
```

```
dimension=3
abs_tol=.001
integrand = Keister(Sobol(dimension=dimension, graycode=False))
solution,data = CubBayesNetG(integrand,abs_tol=abs_tol).integrate()
print(data)
```

```
LDTransformBayesData (AccumulateData Object)
    solution        2.168
    comb_bound_low  2.167
    comb_bound_high 2.169
    comb_flags      1
    n_total         2^(13)
    n               2^(13)
    time_integrate  0.051
CubBayesNetG (StoppingCriterion Object)
    abs_tol         0.001
    rel_tol         0
    n_init          2^(8)
    n_max           2^(22)
Keister (Integrand Object)
Gaussian (TrueMeasure Object)
    mean            0
    covariance      2^(-1)
    decomp_type     PCA
Sobol (DiscreteDistribution Object)
    d               3
    dvec            [0 1 2]
    randomize       LMS_DS
    graycode        0
    entropy         221722953892730177222557457450863582068
    spawn_key       ()
```

5.4 QMCPy for Lebesgue Integration

This notebook will give examples of how to use QMCPy for integration problems that not are defined in terms of a standard measure. i.e. Uniform or Gaussian.

```
from qmcpy import *
from numpy import *
```

5.4.1 Sample Problem 1

$y = \int_{[0,2]} x^2 dx$, Lebesgue Measure

$= 2 \int_{[0,2]} \frac{x^2}{2} dx$, Uniform Measure

```
abs_tol = .01
dim = 1
a = 0
b = 2
true_value = 8./3
```

```
# Lebesgue Measure
integrand = CustomFun(
    true_measure = Lebesgue(Uniform(Halton(dim, seed=7),lower_bound=a, upper_bound=b)),
    g = lambda x: (x**2).sum(1))
solution,data = CubQMCCLT(integrand, abs_tol=abs_tol).integrate()
print('y = %.3f'%solution)
error = abs((solution-true_value))
if error>abs_tol:
    raise Exception("Not within error tolerance")
```

```
y = 2.667
```

```
# Uniform Measure
integrand = CustomFun(
    true_measure = Uniform(IIDStdUniform(dim, seed=7), lower_bound=a, upper_bound=b),
    g = lambda x: (2*(x**2)).sum(1))
solution,data = CubMCCLT(integrand, abs_tol=abs_tol).integrate()
print('y = %.3f'%solution)
error = abs((solution-true_value))
if error>abs_tol:
    raise Exception("Not within error tolerance")
```

```
y = 2.666
```

5.4.2 Sample Problem 2

$y = \int_{[a,b]^d} ||x||_2^2 dx$, Lebesgue Measure

$= \Pi_{i=1}^d (b_i - a_i) \int_{[a,b]^d} ||x||_2^2 \, [\Pi_{i=1}^d (b_i - a_i)]^{-1} dx$, Uniform Measure

```
abs_tol = .001
dim = 2
a = array([1.,2.])
b = array([2.,4.])
true_value = ((a[0]**3-b[0]**3)*(a[1]-b[1])+(a[0]-b[0])*(a[1]**3-b[1]**3))/3
print('Answer = %.5f'%true_value)
```

```
Answer = 23.33333
```

```
# Lebesgue Measure
integrand = CustomFun(
    true_measure = Lebesgue(Uniform(DigitalNetB2(dim, seed=7), lower_bound=a, upper_
↪bound=b)),
    g = lambda x: (x**2).sum(1))
solution,data = CubQMCCLT(integrand, abs_tol=abs_tol).integrate()
print('y = %.5f'%solution)
error = abs((solution-true_value))
if error>abs_tol:
    raise Exception("Not within error tolerance")
```

```
y = 23.33329
```

```
# Uniform Measure
integrand = CustomFun(
    true_measure = Uniform(DigitalNetB2(dim, seed=17), lower_bound=a, upper_bound=b),
    g = lambda x: (b-a).prod()*(x**2).sum(1))
solution,data = CubQMCCLT(integrand, abs_tol=abs_tol).integrate()
print('y = %.5f'%solution)
error = abs((solution-true_value))
if error>abs_tol:
    raise Exception("Not within error tolerance")
```

```
y = 23.33308
```

5.4.3 Sample Problem 3

Integral that cannot be done in terms of any standard mathematical functions

$$y = \int_{[a,b]} \frac{\sin x}{\log x} dx, \text{ Lebesgue Measure}$$

Mathematica Code: `Integrate[Sin[x]/Log[x], {x,a,b}]`

```
abs_tol = .0001
dim = 1
```

(continues on next page)

(continued from previous page)

```
a = 3
b = 5
true_value = -0.87961
```

```
# Lebesgue Measure
integrand = CustomFun(
    true_measure = Lebesgue(Uniform(Lattice(dim, randomize=True, seed=7),a,b)),
    g = lambda x: (sin(x)/log(x)).sum(1))
solution,data = CubQMCLatticeG(integrand, abs_tol=abs_tol).integrate()
print('y = %.3f'%solution)
error = abs((solution-true_value))
if error>abs_tol:
    raise Exception("Not within error tolerance")
```

```
y = -0.880
```

5.4.4 Sample Problem 4

Integral over $\mathbb{R}^d$

$$y = \int_{\mathbb{R}^2} e^{-||x||_2^2} dx$$

```
abs_tol = .1
dim = 2
true_value = pi
```

```
integrand = CustomFun(
    true_measure = Lebesgue(Gaussian(Lattice(dim,seed=7))),
    g = lambda x: exp(-x**2).prod(1))
solution,data = CubQMCLatticeG(integrand,abs_tol=abs_tol).integrate()
print('y = %.3f'%solution)
error = abs((solution-true_value))
if error>abs_tol:
    raise Exception("Not within error tolerance")
```

```
y = 3.142
```

5.5 Scatter Plots of Samples

```
from qmcpy import *
from copy import deepcopy
from numpy import ceil, linspace, meshgrid, zeros, array, arange, random
from mpl_toolkits.mplot3d.axes3d import Axes3D

import matplotlib
%matplotlib inline
```

(continues on next page)

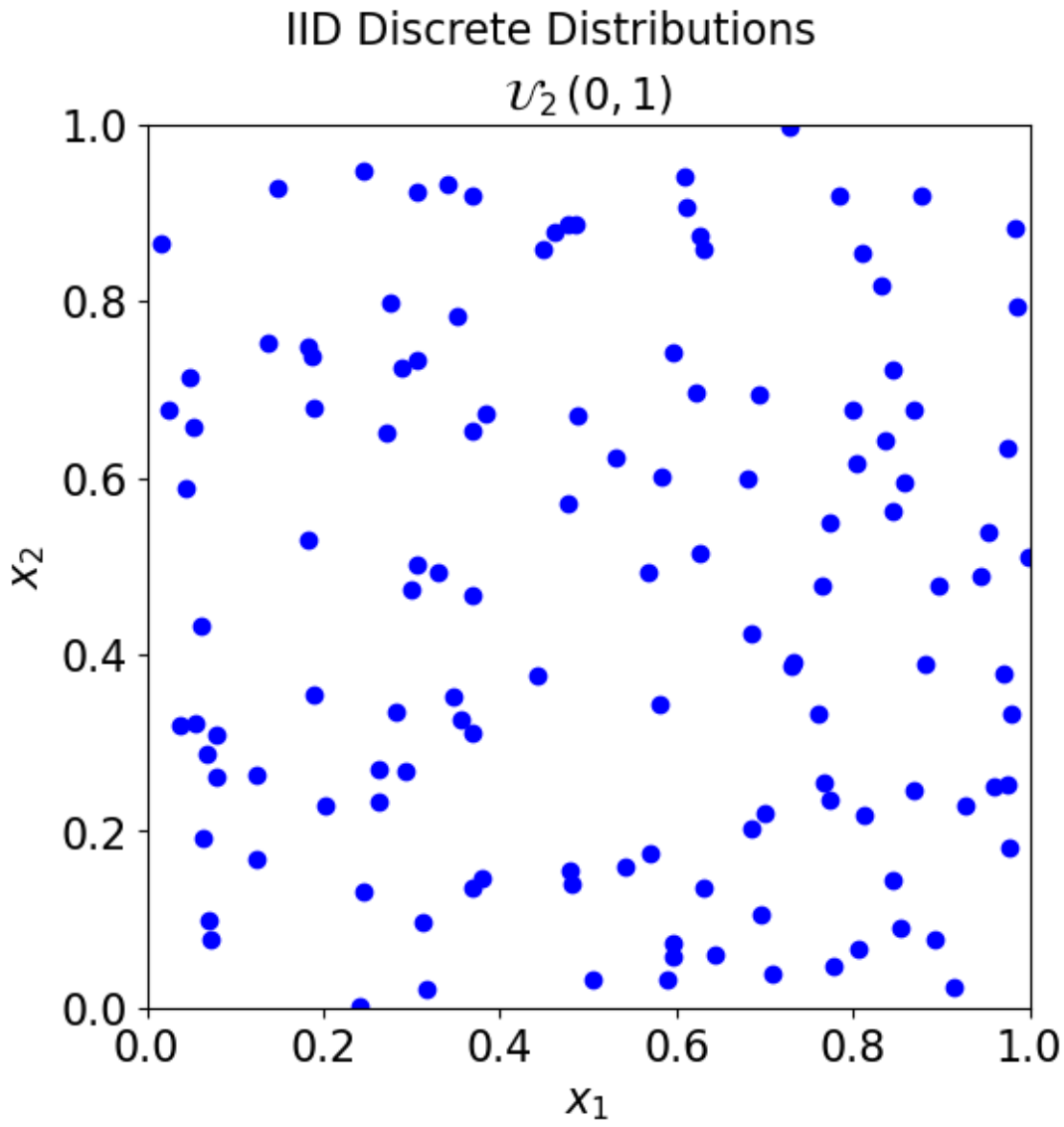
(continued from previous page)

```
import matplotlib.pyplot as plt

plt.rc('font', size=16)          # controls default text sizes
plt.rc('axes', titlesize=16)     # fontsize of the axes title
plt.rc('axes', labelsize=16)    # fontsize of the x and y labels
plt.rc('xtick', labelsize=16)    # fontsize of the tick labels
plt.rc('ytick', labelsize=16)    # fontsize of the tick labels
plt.rc('legend', fontsize=16)    # legend fontsize
plt.rc('figure', titlesize=16)  # fontsize of the figure title
```

```
n = 128
```

5.5.1 IID Samples

```
random.seed(7)
discrete_distribs = [
    IIDStdUniform(dimension=2, seed=7),
    #IIDStdGaussian(dimension=2, seed=7),
    #CustomIIDDistribution(lambda n: random.exponential(scale=2./3,size=(n,2)))
]
dd_names = ["$\\mathcal{U}_2\\,(0,1)$", "$\\mathcal{N}_2\\,(0,1)$", "Exp(1.5)"]
colors = ["b", "r", "g"]
lims = [[0, 1], [-2.5, 2.5],[0,4]]
fig, ax = plt.subplots(nrows=1, ncols=1, figsize=(16, 6))
for i, (dd_obj, color, lim, dd_name) in enumerate(zip(discrete_distribs, colors, lims,␣
↪dd_names)):
    samples = dd_obj.gen_samples(n)
    ax.scatter(samples[:, 0], samples[:, 1], color=color)
    ax.set_xlabel("$x_1$")
    ax.set_ylabel("$x_2$")
    ax.set_xlim(lim)
    ax.set_ylim(lim)
    ax.set_aspect("equal")
    ax.set_title(dd_name)
fig.suptitle("IID Discrete Distributions");
```



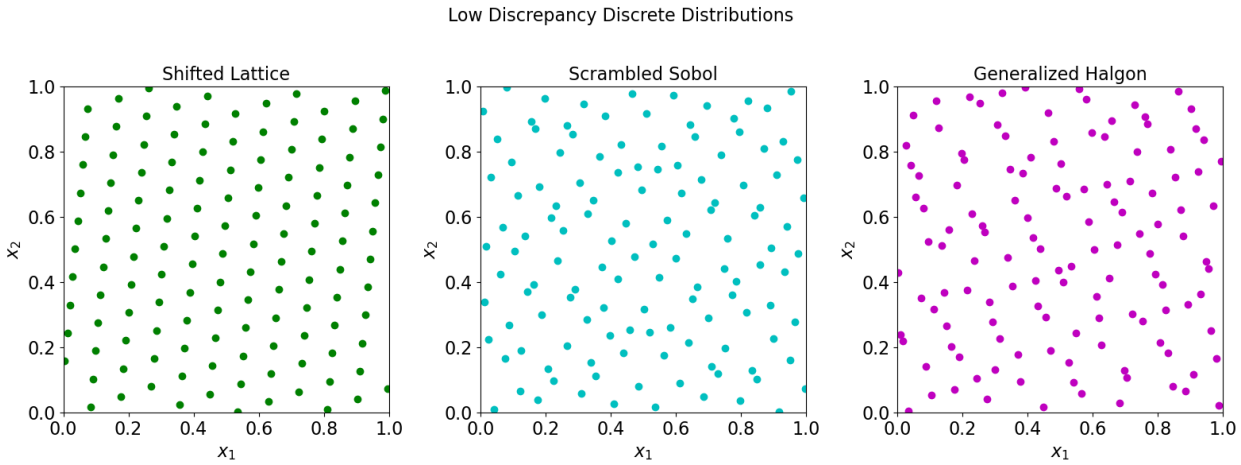
5.5.2 LD Samples

```
discrete_distribs = [
    Lattice(dimension=2, randomize=True, seed=7),
    Sobol(dimension=2, randomize=True, seed=7),
    Halton(dimension=2,generalize=True,seed=7)]
dd_names = ["Shifted Lattice", "Scrambled Sobol", "Generalized Halgon", "Randomized␣
↪Korobov"]
colors = ["g", "c", "m", "r"]
fig, ax = plt.subplots(nrows=1, ncols=3, figsize=(16, 6))
for i, (dd_obj, color, dd_name) in \
        enumerate(zip(discrete_distribs, colors, dd_names)):
    samples = dd_obj.gen_samples(n)
    ax[i].scatter(samples[:, 0], samples[:, 1], color=color)
    ax[i].set_xlabel("$x_1$")
```

(continues on next page)

(continued from previous page)

```
    ax[i].set_ylabel("$x_2$")
    ax[i].set_xlim([0, 1])
    ax[i].set_ylim([0, 1])
    ax[i].set_aspect("equal")
    ax[i].set_title(dd_name)
fig.suptitle("Low Discrepancy Discrete Distributions")
plt.tight_layout();
```



5.5.3 Transform to the True Distribution

Transform samples from a few discrete distributions to mimic various measures

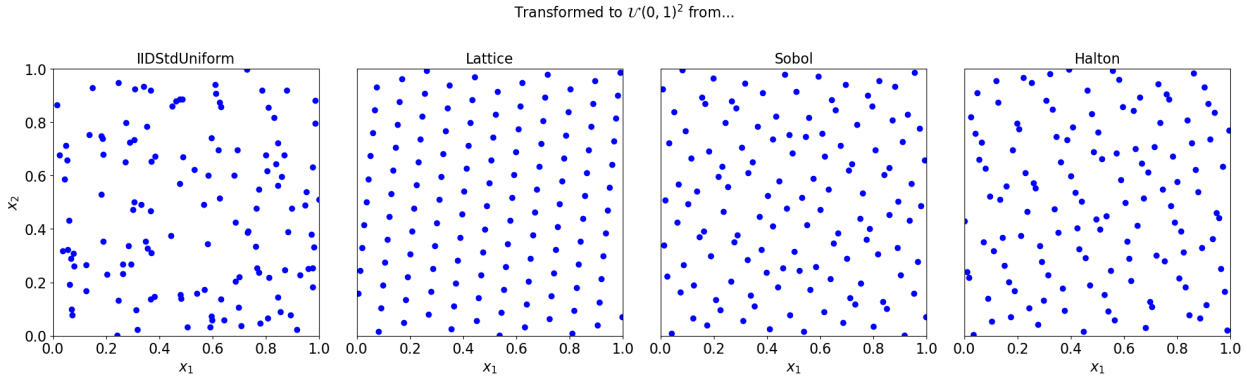
```
def plot_tm_tranformed(tm_name, color, lim, measure, **kwargs):
    fig, ax = plt.subplots(nrows=1, ncols=4, figsize=(20, 6))
    i = 0
    # IID Distributions
    iid_distribs = [
        IIDStdUniform(dimension=2, seed=7),
        #IIDStdGaussian(dimension=2, seed=7)
    ]
    iid_names = [
        "IID $\\mathcal{U}\\,(0,1)^2$",
        "IID $\\mathcal{N}\\,(0,1)^2$"]
    for distrib, distrib_name in zip(iid_distribs, iid_names):
        measure_obj = measure(distrib, **kwargs)
        samples = measure_obj.gen_samples(n)
        ax[i].scatter(samples[:, 0], samples[:, 1], color=color)
        i += 1
    # Quasi Random Distributions
    qrng_distribs = [
        Lattice(dimension=2, randomize=True, seed=7),
        Sobol(dimension=2, randomize=True, seed=7),
        Halton(dimension=2, randomize=True, seed=7)]
    qrng_names = ["Shifted Lattice",
                  "Scrambled Sobol",
```

(continues on next page)

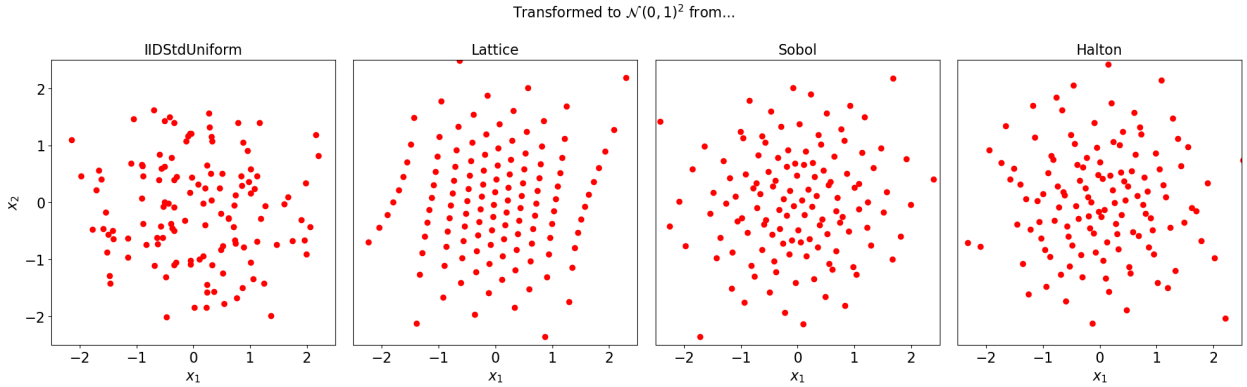
(continued from previous page)

```
                "Randomized Halton"]
    for distrib, distrib_name in zip(qrng_distribs, qrng_names):
        measure_obj = measure(distrib, **kwargs)
        samples = measure_obj.gen_samples(n_min=0,n_max=n)
        ax[i].scatter(samples[:, 0], samples[:, 1], color=color)
        i += 1
    # Plot Metas
    for i,distrib in enumerate(iid_distribs+qrng_distribs):
        ax[i].set_xlabel("$x_1$")
        if i==0:
            ax[i].set_ylabel("$x_2$")
        else:
            ax[i].set_yticks([])
        ax[i].set_xlim(lim)
        ax[i].set_ylim(lim)
        ax[i].set_aspect("equal")
        ax[i].set_title(type(distrib).__name__)
    fig.suptitle("Transformed to %s from..." % tm_name)
    plt.tight_layout()
    prefix = type(measure_obj).__name__;
```

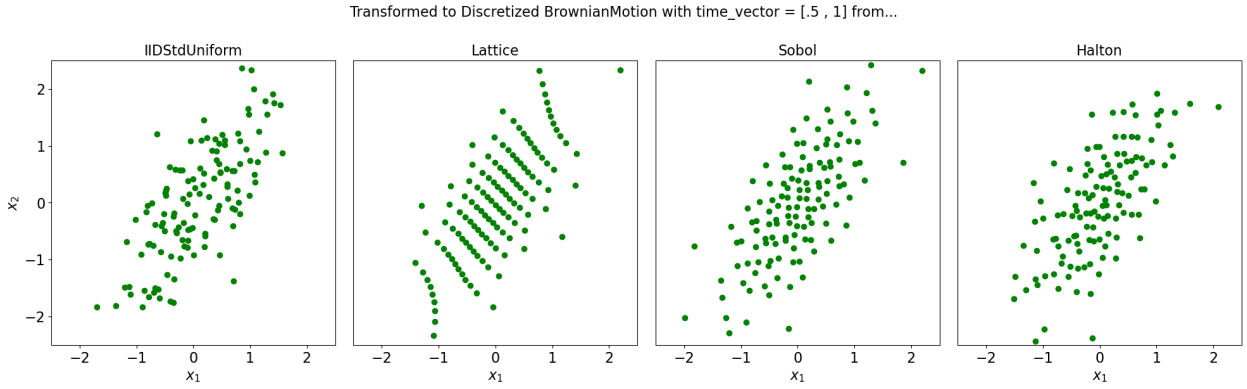
```
plot_tm_tranformed("$\\mathcal{U}\\,(0,1)^2$","b",[0, 1],Uniform)
```




```
plot_tm_tranformed("$\\mathcal{N}\\,(0,1)^2$","r",[-2.5, 2.5],Gaussian)
```



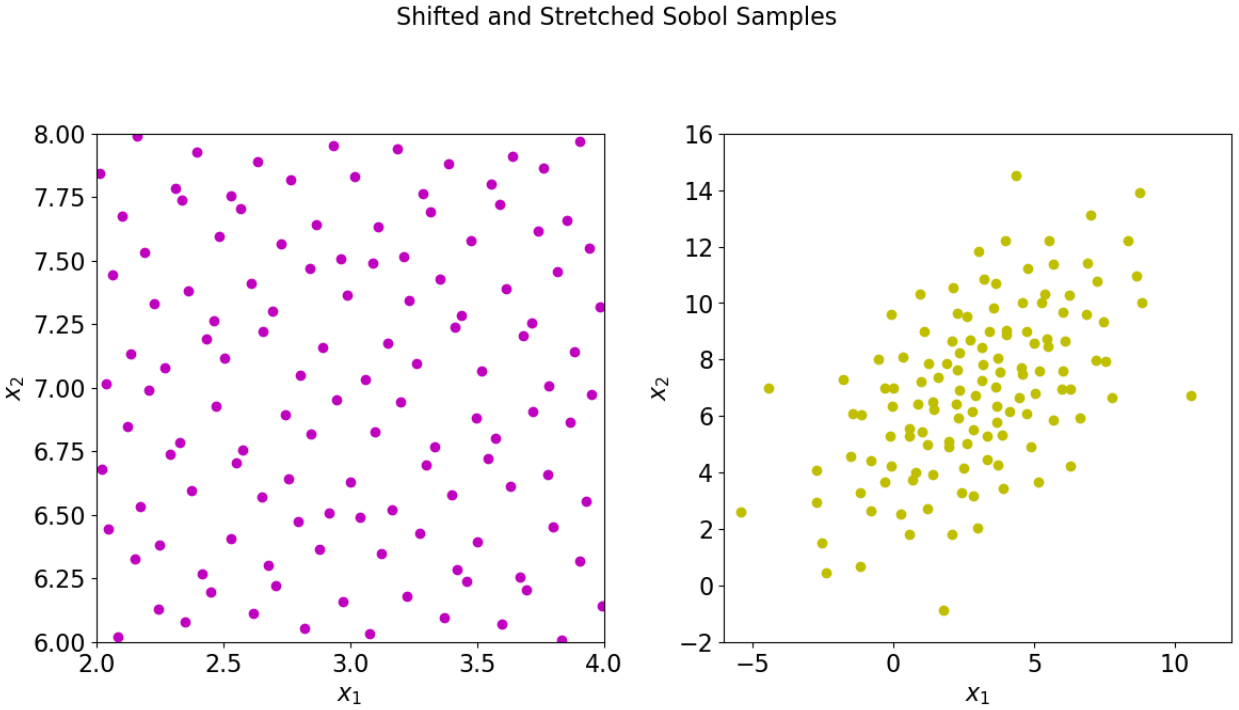
```
plot_tm_tranformed("Discretized BrownianMotion with time_vector = [.5 , 1]",
                   "g",[-2.5, 2.5],BrownianMotion)
```



5.5.4 Shift and Stretch the True Distribution

Transform Sobol sequences to mimic non-standard Uniform and Gaussian measures

```
fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(12, 7))
u1_a, u1_b = 2, 4
u2_a, u2_b = 6, 8
g1_mu, g1_var = 3, 9
g2_mu, g2_var = 7, 9
g_cov = 5
distribution = Sobol(dimension=2, randomize=True, seed=7)
uniform_measure = Uniform(distribution,lower_bound=[u1_a, u2_a],upper_bound=[u1_b, u2_b])
gaussian_measure = Gaussian(distribution,mean=[g1_mu, g2_mu],covariance=[[g1_var, g_cov],
↪[g_cov,g2_var]])
# Generate Samples and Create Scatter Plots
for i, (measure, color) in enumerate(zip([uniform_measure, gaussian_measure], ["m", "y
↪"])):
    samples = measure.gen_samples(n)
    ax[i].scatter(samples[:, 0], samples[:, 1], color=color)
# Plot Metas
for i in range(2):
    ax[i].set_xlabel("$x_1$")
    ax[i].set_ylabel("$x_2$")
    ax[i].set_aspect("equal")
ax[0].set_xlim([u1_a, u1_b])
ax[0].set_ylim([u2_a, u2_b])
spread_g1 = ceil(3 * g1_var**.5)
spread_g2 = ceil(3 * g2_var**.5)
ax[1].set_xlim([g1_mu - spread_g1, g1_mu + spread_g1])
ax[1].set_ylim([g2_mu - spread_g2, g2_mu + spread_g2])
fig.suptitle("Shifted and Stretched Sobol Samples")
plt.tight_layout();
```



5.5.5 Plots samples on a 2D Keister function

```
abs_tol = .3
integrand = Keister(IIDStdUniform(dimension=2, seed=7))
solution,data = CubMCCLT(integrand,abs_tol=abs_tol,rel_tol=0,n_init=16, n_max=1e10).
↪integrate()
print(data)
```

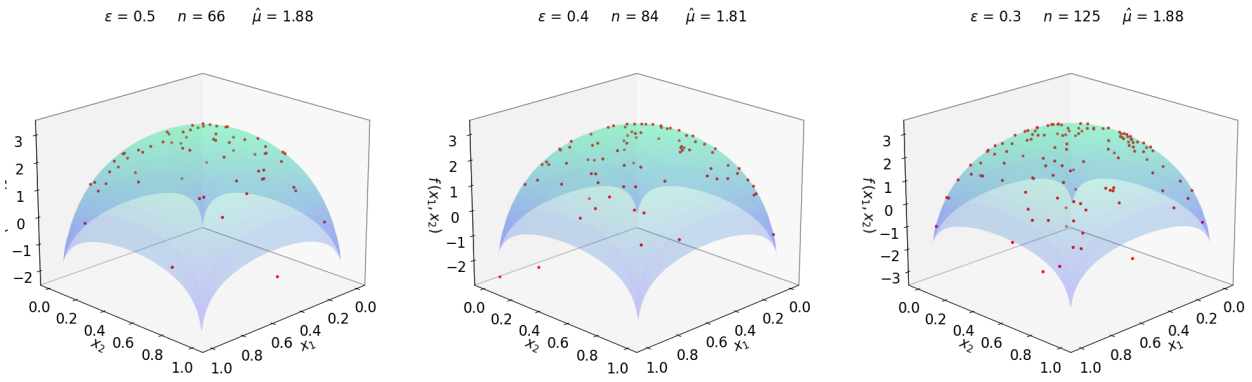
```
MeanVarData (AccumulateData Object)
    solution        1.562
    error_bound     0.541
    n_total         76
    n               60
    levels          1
    time_integrate  9.80e-04
CubMCCLT (StoppingCriterion Object)
    abs_tol         0.300
    rel_tol         0
    n_init          2^(4)
    n_max           10000000000
    inflate         1.200
    alpha           0.010
Keister (Integrand Object)
Gaussian (TrueMeasure Object)
    mean            0
    covariance      2^(-1)
    decomp_type     PCA
IIDStdUniform (DiscreteDistribution Object)
```

(continues on next page)

(continued from previous page)

```
    d               2^(1)
    entropy         7
    spawn_key       ()
```

```
# Constants based on running the above CLT Example
eps_list = [.5, .4, .3]
n_list = [66, 84, 125]
mu_hat_list = [1.8757, 1.8057, 1.8829]
# Function Points
nx, ny = (99, 99)
points_fun = zeros((nx * ny, 3))
x = arange(.01, 1, .01)
y = arange(.01, 1, .01)
x_2d, y_2d = meshgrid(x, y)
points_fun[:, 0] = x_2d.flatten()
points_fun[:, 1] = y_2d.flatten()
points_fun[:, 2] = integrand.f(points_fun[:, :2]).squeeze()
x_surf = points_fun[:, 0].reshape((nx, ny))
y_surf = points_fun[:, 1].reshape((nx, ny))
z_surf = points_fun[:, 2].reshape((nx, ny))
# 3D Plot
fig = plt.figure(figsize=(25, 8))
ax1 = fig.add_subplot(131, projection="3d")
ax2 = fig.add_subplot(132, projection="3d")
ax3 = fig.add_subplot(133, projection="3d")
for idx, ax in enumerate([ax1, ax2, ax3]):
    n = n_list[idx]
    epsilon = eps_list[idx]
    mu = mu_hat_list[idx]
    # Surface
    ax.plot_surface(x_surf, y_surf, z_surf, cmap="winter", alpha=.2)
    # Scatters
    points = zeros((n, 3))
    points[:, :2] = integrand.discrete_distrib.gen_samples(n)
    points[:, 2] = integrand.f(points[:, :2]).squeeze()
    ax.scatter(points[:, 0], points[:, 1], points[:, 2], color="r", s=5)
    ax.scatter(points[:, 0], points[:, 1], points[:, 2], color="r", s=5)
    ax.set_title("\t$\\epsilon$ = %-7.1f $n$ = %-7d $\\hat{\\mu}$ = %-7.2f "
                 % (epsilon, n, mu), fontdict={"fontsize": 16})
    # axis metas
    n *= 2
    ax.grid(False)
    ax.xaxis.pane.set_edgecolor("black")
    ax.yaxis.pane.set_edgecolor("black")
    ax.set_xlabel("$x_1$", fontdict={"fontsize": 16})
    ax.set_ylabel("$x_2$", fontdict={"fontsize": 16})
    ax.set_zlabel("$f\\:(x_1,x_2)$", fontdict={"fontsize": 16})
    ax.view_init(20, 45);
```



5.6 A Monte Carlo vs Quasi-Monte Carlo Comparison

Monte Carlo algorithms work on independent identically distributed (IID) points while Quasi-Monte Carlo algorithms work on low discrepancy (LD) sequences. LD generators, such as those for the lattice and Sobol' sequences, provide samples whose space filling properties can be exploited by Quasi-Monte Carlo algorithms.

```
import pandas as pd
pd.options.display.float_format = '{:.2e}'.format
from matplotlib import pyplot as plt
import matplotlib
%matplotlib inline

plt.rc('font', size=16)          # controls default text sizes
plt.rc('axes', titlesize=16)     # fontsize of the axes title
plt.rc('axes', labelsize=16)    # fontsize of the x and y labels
plt.rc('xtick', labelsize=16)    # fontsize of the tick labels
plt.rc('ytick', labelsize=16)    # fontsize of the tick labels
plt.rc('legend', fontsize=16)    # legend fontsize
plt.rc('figure', titlesize=16)  # fontsize of the figure title
```

5.6.1 Vary Absolute Tolerance

Testing Parameters - relative tolerance = 0 - Results averaged over 3 trials

Keister Integrand - $y_i = \pi^{d/2}\cos(||\boldsymbol{x}_i||_2)$ - $d = 3$

Gaussian True Measure - $\mathcal{N}(\boldsymbol{0}, \boldsymbol{I}/2)$

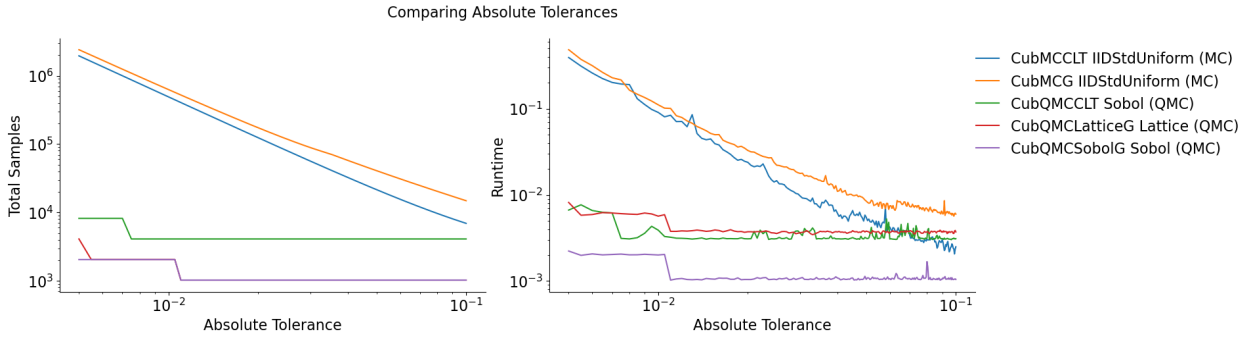
```
df = pd.read_csv('../workouts/mc_vs_qmc/out/vary_abs_tol.csv')
df['Problem'] = df['Stopping Criterion'] + ' ' + df['Distribution'] + ' (' + df['MC/QMC
↪'] + ')'
df = df.drop(['Stopping Criterion','Distribution','MC/QMC'],axis=1)
problems = ['CubMCCLT IIDStdUniform (MC)',
            'CubMCG IIDStdUniform (MC)',
            'CubQMCCLT Sobol (QMC)',
            'CubQMCLatticeG Lattice (QMC)',
            'CubQMCSobolG Sobol (QMC)']
df = df[df['Problem'].isin(problems)]
```

(continues on next page)

(continued from previous page)

```
df['abs_tol'] = df['abs_tol'].round(4)
df_grouped = df.groupby(['Problem'])
df_abs_tols = df_grouped['abs_tol'].apply(list).reset_index(name='abs_tol')
df_samples = df_grouped['n_samples'].apply(list).reset_index(name='n')
df_times = df.groupby(['Problem'])['time'].apply(list).reset_index(name='time')
df[df['abs_tol'].isin([.01,.05,.1])].set_index('Problem')
```

```
fig,ax = plt.subplots(nrows=1, ncols=2, figsize=(18, 5))
for problem in problems:
    abs_tols = df_abs_tols[df_abs_tols['Problem']==problem]['abs_tol'].tolist()[0]
    samples = df_samples[df_samples['Problem']==problem]['n'].tolist()[0]
    times = df_times[df_times['Problem']==problem]['time'].tolist()[0]
    ax[0].plot(abs_tols,samples,label=problem)
    ax[1].plot(abs_tols,times,label=problem)
for ax_i in ax:
    ax_i.set_xscale('log', base=10)
    ax_i.set_yscale('log', base=10)
    ax_i.spines['right'].set_visible(False)
    ax_i.spines['top'].set_visible(False)
    ax_i.set_xlabel('Absolute Tolerance')
ax[0].legend(loc='upper right', frameon=False,ncol=1,bbox_to_anchor=(2.8,1))
ax[0].set_ylabel('Total Samples')
ax[1].set_ylabel('Runtime')
fig.suptitle('Comparing Absolute Tolerances')
plt.subplots_adjust(wspace=.15, hspace=0);
```



Quasi-Monte Carlo takes less time and fewer samples to achieve the same accuracy as regular Monte Carlo The number of points for Monte Carlo algorithms is $\mathcal{O}(1/\epsilon^2)$ while quasi-Monte Carlo algorithms can be as efficient as $\mathcal{O}(1/\epsilon)$

5.6.2 Vary Dimension

Testing Parameters - absolute tolerance = 0 - relative tolerance = .01 - Results averaged over 3 trials

Keister Integrand - $y_i = \pi^{d/2}\cos(||\boldsymbol{x}_i||_2)$

Gaussian True Measure - $\mathcal{N}(\boldsymbol{0}, \boldsymbol{I}/2)$

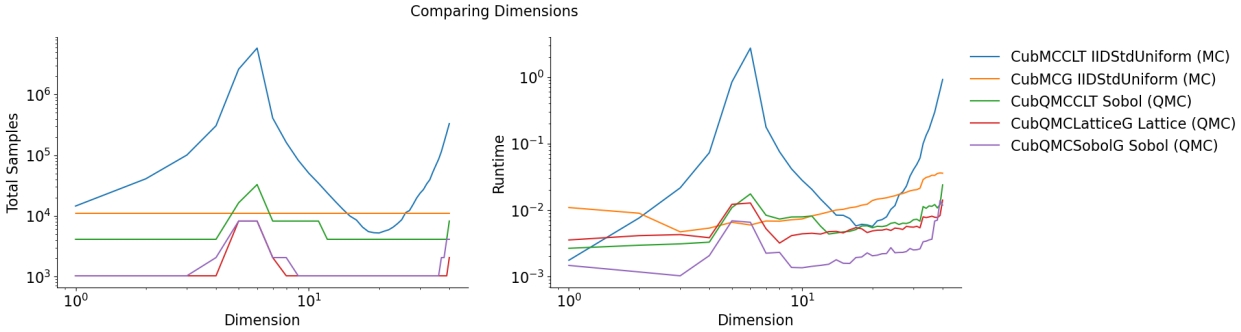
```
df = pd.read_csv('../workouts/mc_vs_qmc/out/vary_dimension.csv')
df['Problem'] = df['Stopping Criterion'] + ' ' + df['Distribution'] + ' (' + df['MC/QMC
↪'] + ')'
df = df.drop(['Stopping Criterion','Distribution','MC/QMC'],axis=1)
```

(continues on next page)

(continued from previous page)

```
problems = ['CubMCCLT IIDStdUniform (MC)',
            'CubMCG IIDStdUniform (MC)',
            'CubQMCCLT Sobol (QMC)',
            'CubQMCLatticeG Lattice (QMC)',
            'CubQMCSobolG Sobol (QMC)']
df = df[df['Problem'].isin(problems)]
df_grouped = df.groupby(['Problem'])
df_dims = df_grouped['dimension'].apply(list).reset_index(name='dimension')
df_samples = df_grouped['n_samples'].apply(list).reset_index(name='n')
df_times = df.groupby(['Problem'])['time'].apply(list).reset_index(name='time')
df[df['dimension'].isin([10,30])].set_index('Problem')
```

```
fig,ax = plt.subplots(nrows=1, ncols=2, figsize=(18, 5))
for problem in problems:
    dimension = df_dims[df_dims['Problem']==problem]['dimension'].tolist()[0]
    samples = df_samples[df_samples['Problem']==problem]['n'].tolist()[0]
    times = df_times[df_times['Problem']==problem]['time'].tolist()[0]
    ax[0].plot(dimension,samples,label=problem)
    ax[1].plot(dimension,times,label=problem)
for ax_i in ax:
    ax_i.set_xscale('log', base=10)
    ax_i.set_yscale('log', base=10)
    ax_i.spines['right'].set_visible(False)
    ax_i.spines['top'].set_visible(False)
    ax_i.set_xlabel('Dimension')
ax[0].legend(loc='upper right', frameon=False,ncol=1,bbox_to_anchor=(2.9,1))
ax[0].set_ylabel('Total Samples')
ax[1].set_ylabel('Runtime')
fig.suptitle('Comparing Dimensions');
```



5.7 Quasi-Random Sequence Generator Comparison

```
from qmcpy import *

import pandas as pd
pd.options.display.float_format = '{:.2e}'.format

from numpy import *
```

(continues on next page)

(continued from previous page)

```
set_printoptions(threshold=2**10)
set_printoptions(precision=3)

from matplotlib import pyplot as plt
import matplotlib
%matplotlib inline

SMALL_SIZE = 10
MEDIUM_SIZE = 12
BIGGER_SIZE = 14

plt.rc('font', size=BIGGER_SIZE)          # controls default text sizes
plt.rc('axes', titlesize=BIGGER_SIZE)     # fontsize of the axes title
plt.rc('axes', labelsize=BIGGER_SIZE)    # fontsize of the x and y labels
plt.rc('xtick', labelsize=MEDIUM_SIZE)    # fontsize of the tick labels
plt.rc('ytick', labelsize=MEDIUM_SIZE)    # fontsize of the tick labels
plt.rc('legend', fontsize=BIGGER_SIZE)    # legend fontsize
plt.rc('figure', titlesize=BIGGER_SIZE)  # fontsize of the figure title
```

5.7.1 General Usage

```
# Unshifted Samples
lattice = Lattice(dimension=2, randomize=False, seed=7)
unshifted_samples = lattice.gen_samples(n_min=2,n_max=8)
print('Shape: %s'%str(unshifted_samples.shape))
print('Samples:\n'+str(unshifted_samples))
```

```
Shape: (6, 2)
Samples:
[[0.25  0.75 ]
 [0.75  0.25 ]
 [0.125 0.375]
 [0.625 0.875]
 [0.375 0.125]
 [0.875 0.625]]
```

```
# Shifted Samples
lattice = Lattice(dimension=2, randomize=True, seed=7)
shifted_samples = lattice.gen_samples(n_min=4, n_max=8)
print('Shape: %s'%str(shifted_samples.shape))
print('Samples:\n'+str(shifted_samples))
```

```
Shape: (4, 2)
Samples:
[[0.169 0.962]
 [0.669 0.462]
 [0.419 0.712]
 [0.919 0.212]]
```

5.7.2 QMCPy Generator Times Comparison

Compare the speed of low-discrepancy-sequence generators from Python (QMCPy), MATLAB, and R. The following blocks visualize speed comparisons when generating one-dimensional unshifted/unscrambled sequences. Note that the generators are reinitialized before every trial.

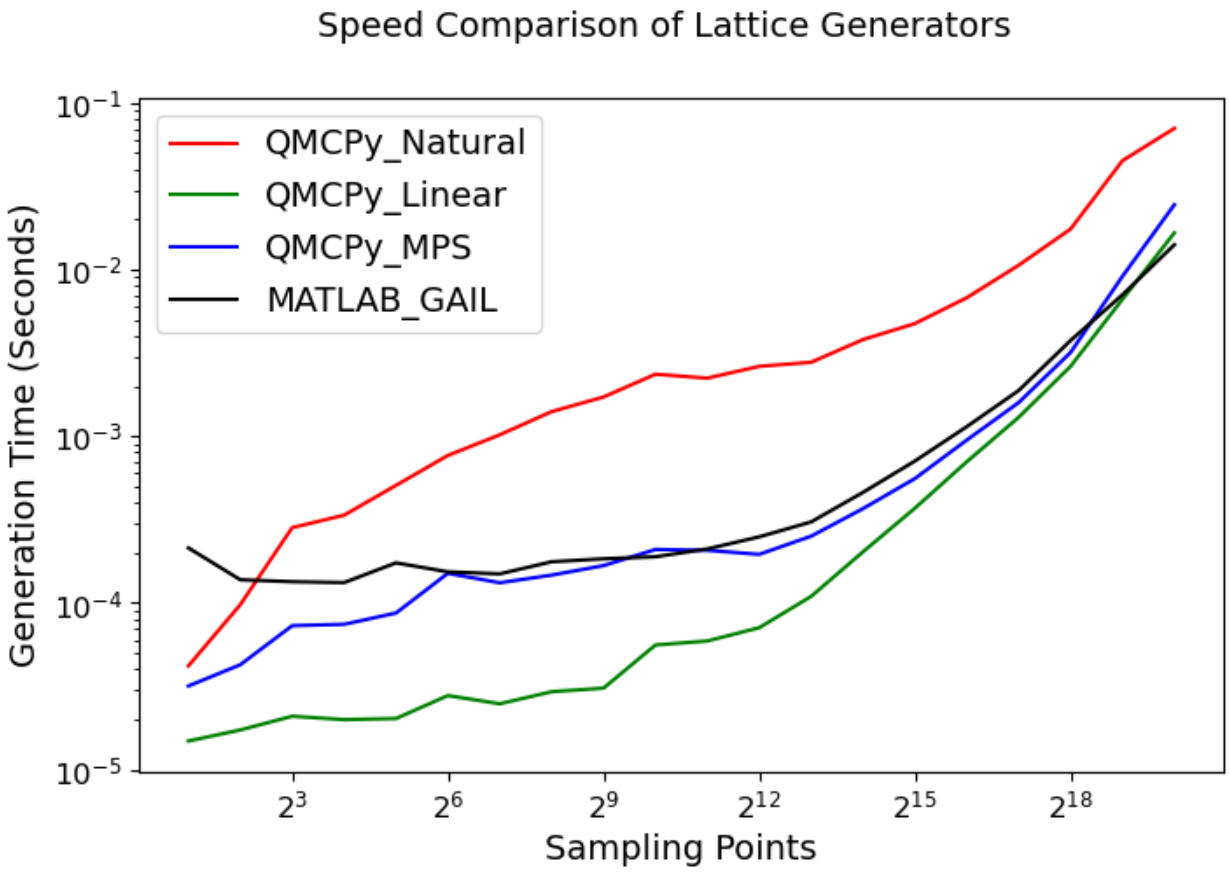
```
# Load AccumulateData
df_py = pd.read_csv('../workouts/lds_sequences/out/python_sequences.csv')
df_py.columns = ['n',
                 'py_n','py_l','py_mps',
                 'py_h_QRNG','py_h_Owen',
                 'py_s_QMCPy','py_s_SciPy']
df_m = pd.read_csv('../workouts/lds_sequences/out/matlab_sequences.csv', header=None)
df_m.columns = ['n', 'm_l', 'm_s','m_h']
df_r = pd.read_csv('../workouts/lds_sequences/out/r_sequences.csv')
df_r.columns = ['n','r_s','r_h','r_k']
df_r.reset_index(drop=True, inplace=True)
```

```
def plt_lds_comp(df,name,colors):
    fig,ax = plt.subplots(nrows=1, ncols=1, figsize=(8,5))
    labels = df.columns[1:]
    n = df['N']
    for label,color in zip(labels,colors):
        ax.plot(n, df[label], label=label, color=color)
        ax.set_xscale('log',base=2)
        ax.set_yscale('log',base=10)
    ax.legend(loc='upper left')
    ax.set_xlabel('Sampling Points')
    ax.set_ylabel('Generation Time (Seconds)')
    # Metas and Export
    fig.suptitle('Speed Comparison of %s Generators'%name)
```

Lattice

```
df_l = pd.concat([df_py['n'], df_py['py_n'], df_py['py_l'], df_py['py_mps'], df_m['m_l
↪']], axis=1)
df_l.columns = ['N','QMCPy_Natural','QMCPy_Linear','QMCPy_MPS','MATLAB_GAIL']
df_l.set_index('N')
```

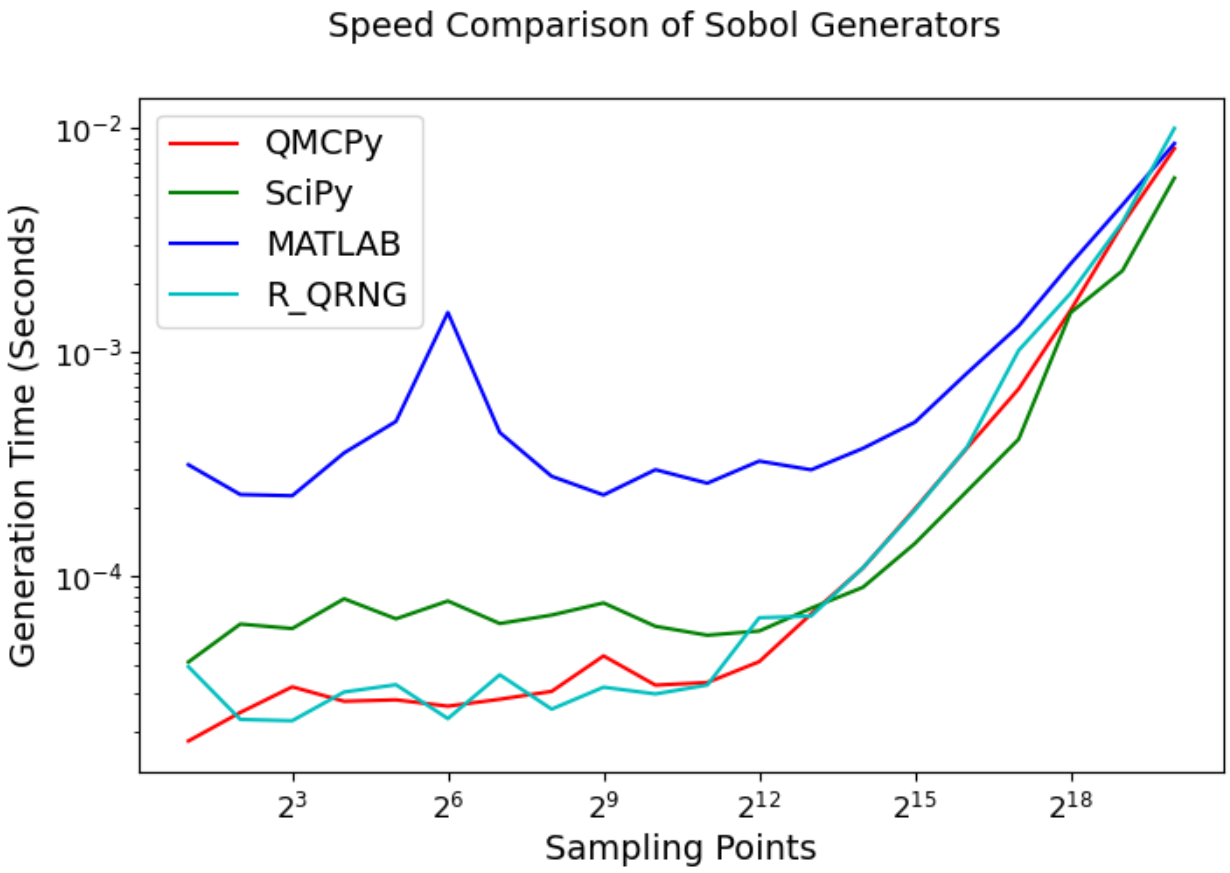
```
plt_lds_comp(df_l,'Lattice',colors=['r','g','b','k'])
```



Sobol

```
df_s = pd.concat([df_py['n'], df_py['py_s_QMCPy'], df_py['py_s_SciPy'], df_m['m_s'], df_
↪r['r_s']], axis=1)
df_s.columns = ['N','QMCPy','SciPy','MATLAB','R_QRNG']
df_s.set_index('N')
```

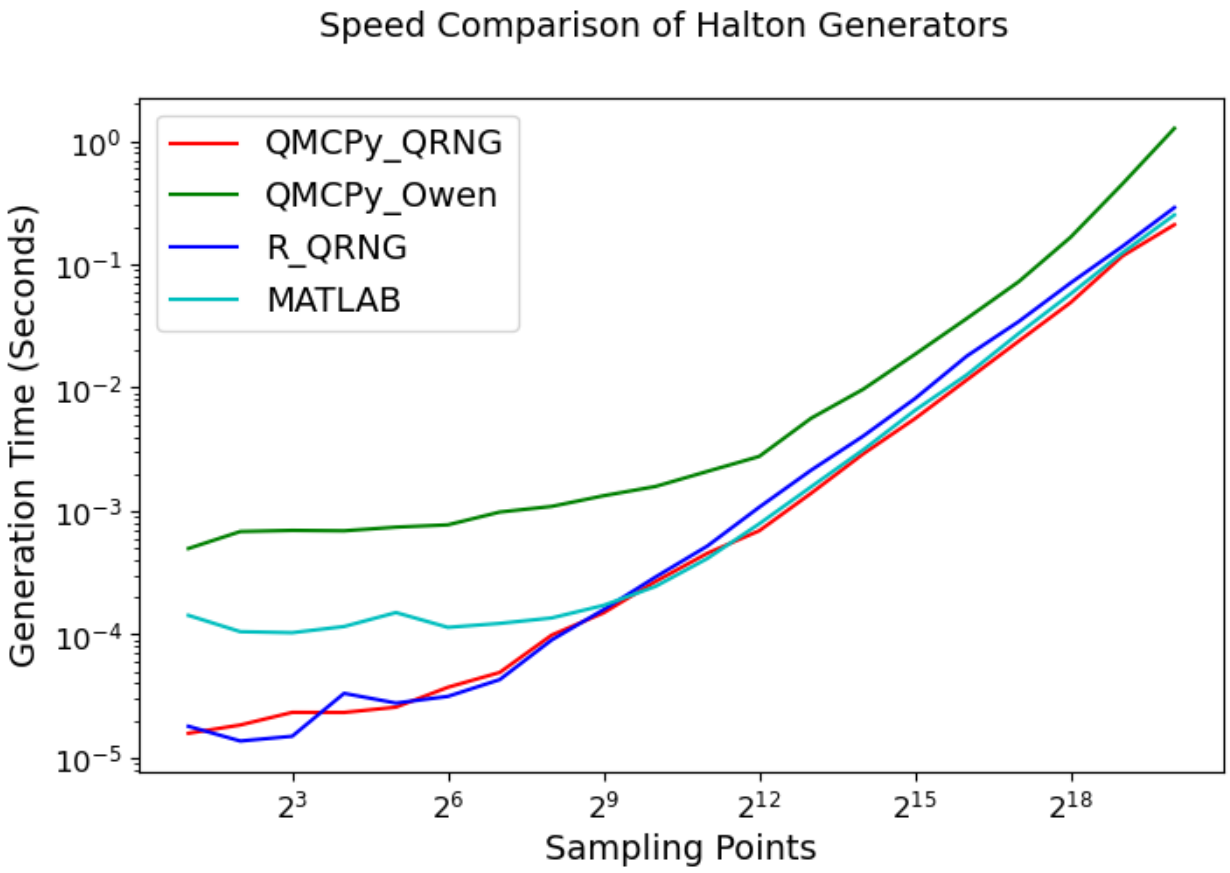
```
plt_lds_comp(df_s,'Sobol',['r','g','b','c','m']) # GC = GrayCode, N=Natural
```



Halton (Generalized)

```
df_h = pd.concat([df_py['n'], df_py['py_h_QRNG'],df_py['py_h_Owen'], df_r['r_h'],df_m['m_
↪h']], axis=1)
df_h.columns = ['N','QMCPy_QRNG','QMCPy_Owen','R_QRNG','MATLAB']
df_h.set_index('N')
```

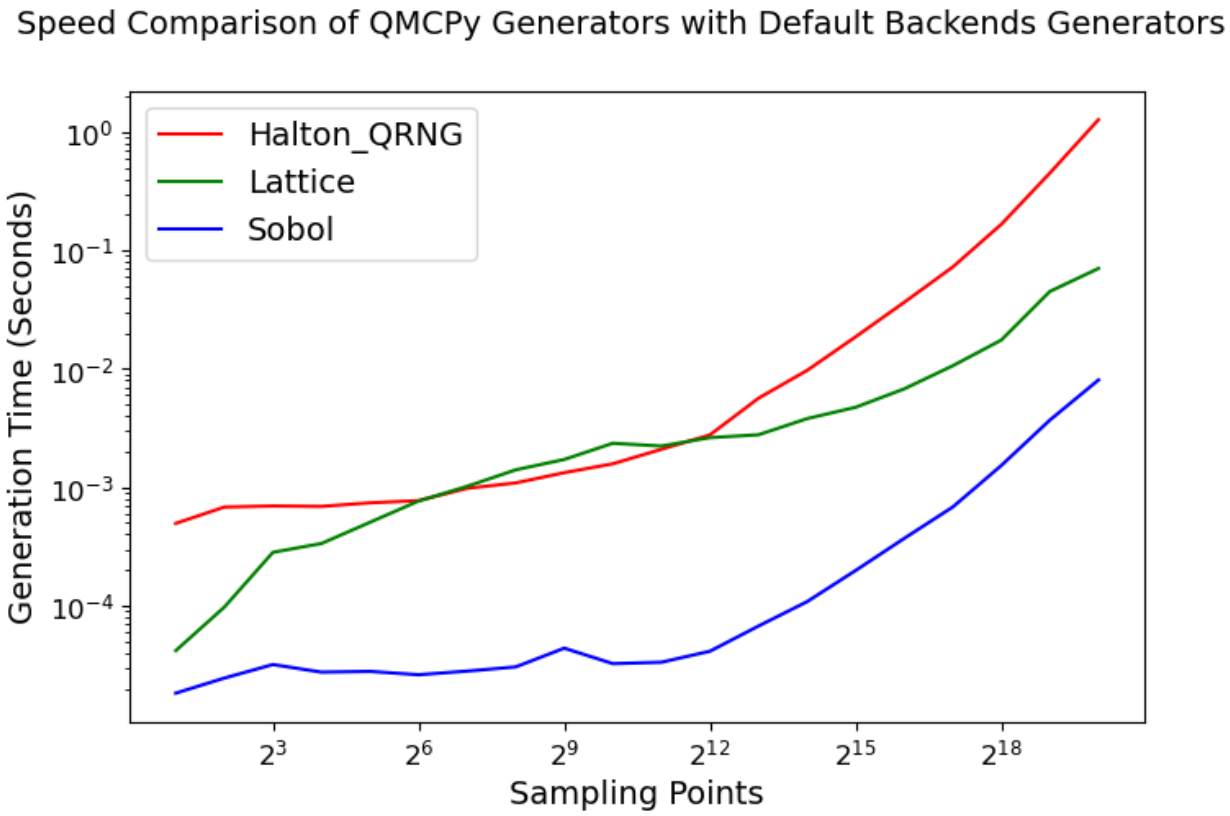
```
plt_lds_comp(df_h,'Halton',colors=['r','g','b','c'])
```



QMCPy Default Generators

```
df_qmcpy = pd.concat([df_py['n'], df_py['py_h_Owen'], df_py['py_n'], df_py['py_s_QMCPy
↪']], axis=1)
df_qmcpy.columns = ['N','Halton_QRNG', 'Lattice', 'Sobol']
df_qmcpy.set_index('N')
```

```
plt_lds_comp(df_qmcpy,'QMCPy Generators with Default Backends',colors=['r','g','b','c'])
```



5.8 Importance Sampling Examples

```
from qmcpy import *
from numpy import *
import pandas as pd
pd.options.display.float_format = '{:.2e}'.format
from matplotlib import pyplot as plt
import matplotlib
%matplotlib inline
```

5.8.1 Game Example

Consider a game where $X_1, X_2 \overset{\text{IID}}{\sim} \mathcal{U}(0,1)$ are drawn with a payoff of

$$Y = \text{payoff}(X_1, X_2) = \begin{cases} \$10, & 1.7 \le X_1 + X_2 \le 2, \\ 0, & 0 \le X_1 + X_2 < 1.7, \end{cases}$$

What is the expected payoff of this game?

```
payoff = lambda x: 10*(x.sum(1)>1.7)
abs_tol = 1e-3
```

Vanilla Monte Carlo

With ordinary Monte Carlo we do the following:

$$\mu = \mathbb{E}(Y) = \int_{[0,1]^2} \text{payoff}(x_1, x_2)\, \mathrm{d}x_1 \mathrm{d}x_2$$

```
d = 2
integral = CustomFun(
    true_measure = Uniform(Lattice(d)),
    g = payoff)
solution1,data1 = CubQMCLatticeG(integral, abs_tol).integrate()
data1
```

```
LDTransformData (AccumulateData Object)
    solution        0.450
    comb_bound_low  0.449
    comb_bound_high 0.450
    comb_flags      1
    n_total         2^(16)
    n               2^(16)
    time_integrate  0.043
CubQMCLatticeG (StoppingCriterion Object)
    abs_tol         0.001
    rel_tol         0
    n_init          2^(10)
    n_max           2^(35)
CustomFun (Integrand Object)
Uniform (TrueMeasure Object)
    lower_bound     0
    upper_bound     1
Lattice (DiscreteDistribution Object)
    d               2^(1)
    dvec            [0 1]
    randomize       1
    order           natural
    entropy         260768472417330885254602359583380448047
    spawn_key       ()
```

Monte Carlo with Importance Sampling

We may add the importance sampling to increase the number of samples with positive payoffs. Let

$$\boldsymbol{Z} = (X_1^{1/(p+1)}, X_2^{1/(p+1)}), \qquad \boldsymbol{X} \sim \mathcal{U}(0,1)^2$$

This means that Z_1 and Z_2 are IID with common CDF $F(z) = z^{p+1}$ and common PDF $\varrho(z) = (p+1)z^p$. Thus,

$$\begin{aligned}\mu = \mathbb{E}(Y) &= \int_{[0,1]^2} \frac{\text{payoff}(z_1, z_2)}{(p+1)^2 (z_1 z_2)^p}\, \varrho(z_1)\varrho(z_2)\, \mathrm{d}z_1 \mathrm{d}z_2 \\ &= \int_{[0,1]^2} \frac{\text{payoff}(x_1^{1/(p+1)}, x_2^{1/(p+1)})}{(p+1)^2 (x_1 x_2)^{p/(p+1)}}\, \mathrm{d}x_1 \mathrm{d}x_2\end{aligned}$$

```
p = 1
d = 2
integral = CustomFun(
    true_measure = Uniform(Lattice(d)),
    g = lambda x: payoff(x**(1/(p+1))) / ((p+1)**2 * (x.prod(1))**(p/(p+1))))
solution2,data2 = CubQMCLatticeG(integral, abs_tol).integrate()
data2
```

```
LDTransformData (AccumulateData Object)
    solution        0.451
    comb_bound_low  0.450
    comb_bound_high 0.451
    comb_flags      1
    n_total         2^(14)
    n               2^(14)
    time_integrate  0.019
CubQMCLatticeG (StoppingCriterion Object)
    abs_tol         0.001
    rel_tol         0
    n_init          2^(10)
    n_max           2^(35)
CustomFun (Integrand Object)
Uniform (TrueMeasure Object)
    lower_bound     0
    upper_bound     1
Lattice (DiscreteDistribution Object)
    d               2^(1)
    dvec            [0 1]
    randomize       1
    order           natural
    entropy         101794984569737015308869089738144162915
    spawn_key       ()
```

```
print('Imporance Sampling takes %.3f the time and %.3f the samples'%\
     (data2.time_integrate/data1.time_integrate,data2.n_total/data1.n_total))
```

```
Imporance Sampling takes 0.451 the time and 0.250 the samples
```

5.8.2 Asian Call Option Example

The stock price must raise significantly for the payoff to be positive. So we will give an upward drift to the Brownian motion that defines the stock price path. We can think of the option price as the multidimensional integral

$$\mu = \mathbb{E}[f(\boldsymbol{X})] = \int_{\mathbb{R}^d} f(\boldsymbol{x}) \frac{\exp\left(-\frac{1}{2}\boldsymbol{x}^T\Sigma^{-1}\boldsymbol{x}\right)}{\sqrt{(2\pi)^d \det(\Sigma)}} \,\mathrm{d}\boldsymbol{x}$$

where

$$\boldsymbol{X} \sim \mathcal{N}(\boldsymbol{0}, \Sigma), \qquad \Sigma = \big(\min(j,k)T/d\big)_{j,k=1}^{d}, \qquad d = 13,$$
$$f(\boldsymbol{x}) = \max\left(K - \frac{1}{d}\sum_{j=1}^{d} S(jT/d, \boldsymbol{x}), 0\right)\mathrm{e}^{-rT},$$
$$S(jT/d, \boldsymbol{x}) = S(0)\exp\big((r-\sigma^2/2)jT/d + \sigma x_j\big).$$

We will replace $\boldsymbol{X}$ by

$$\boldsymbol{Z} \sim \mathcal{N}(\boldsymbol{a}, \Sigma), \qquad \boldsymbol{a} = (aT/d)(1, \ldots, d)$$

where a positive a will create more positive payoffs. This corresponds to giving our Brownian motion a drift. To do this we re-write the integral as

$$\mu = \mathbb{E}[f_{\text{new}}(\boldsymbol{Z})] = \int_{\mathbb{R}^d} f_{\text{new}}(\boldsymbol{z}) \frac{\exp\big(-\frac{1}{2}(\boldsymbol{z}-\boldsymbol{a})^T\Sigma^{-1}(\boldsymbol{z}-\boldsymbol{a})\big)}{\sqrt{(2\pi)^d \det(\Sigma)}}\,\mathrm{d}\boldsymbol{z},$$
$$f_{\text{new}}(\boldsymbol{z}) = f(\boldsymbol{z})\frac{\exp\big(-\frac{1}{2}\boldsymbol{z}^T\Sigma^{-1}\boldsymbol{z}\big)}{\exp\big(-\frac{1}{2}(\boldsymbol{z}-\boldsymbol{a})^T\Sigma^{-1}(\boldsymbol{z}-\boldsymbol{a})\big)} = f(\boldsymbol{z})\exp\big((\boldsymbol{a}/2-\boldsymbol{z})^T\Sigma^{-1}\boldsymbol{a}\big)$$

Finally note that

$$\Sigma^{-1}\boldsymbol{a} = \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ a \end{pmatrix}, \qquad f_{\text{new}}(\boldsymbol{z}) = f(\boldsymbol{z})\exp\big((aT/2 - z_d)a\big)$$

This drift in the Brownian motion may be implemented by changing the `drift` input to the `BrownianMotion` object.

```
abs_tol = 1e-2
dimension = 32
```

Vanilla Monte Carlo

```
integrand = AsianOption(Sobol(dimension))
solution1,data1 = CubQMCSobolG(integrand, abs_tol).integrate()
data1
```

```
LDTransformData (AccumulateData Object)
    solution        1.788
    comb_bound_low  1.782
    comb_bound_high 1.794
    comb_flags      1
    n_total         2^(12)
    n               2^(12)
    time_integrate  0.014
CubQMCSobolG (StoppingCriterion Object)
    abs_tol         0.010
    rel_tol         0
    n_init          2^(10)
```

(continues on next page)

(continued from previous page)

```
    n_max           2^(35)
AsianOption (Integrand Object)
    volatility      2^(-1)
    call_put        call
    start_price     30
    strike_price    35
    interest_rate   0
    mean_type       arithmetic
    dim_frac        0
BrownianMotion (TrueMeasure Object)
    time_vec        [0.031 0.062 0.094 ... 0.938 0.969 1.   ]
    drift           0
    mean            [0. 0. 0. ... 0. 0. 0.]
    covariance      [[0.031 0.031 0.031 ... 0.031 0.031 0.031]
                    [0.031 0.062 0.062 ... 0.062 0.062 0.062]
                    [0.031 0.062 0.094 ... 0.094 0.094 0.094]
                    ...
                    [0.031 0.062 0.094 ... 0.938 0.938 0.938]
                    [0.031 0.062 0.094 ... 0.938 0.969 0.969]
                    [0.031 0.062 0.094 ... 0.938 0.969 1.   ]]
    decomp_type     PCA
Sobol (DiscreteDistribution Object)
    d               2^(5)
    dvec            [ 0  1  2 ... 29 30 31]
    randomize       LMS_DS
    graycode        0
    entropy         296048682239793520955192894339316320391
    spawn_key       ()
```

Monte Carlo with Importance Sampling

```
drift = 1
integrand = AsianOption(BrownianMotion(Sobol(dimension),drift=drift))
solution2,data2 = CubQMCSobolG(integrand, abs_tol).integrate()
data2
```

```
LDTransformData (AccumulateData Object)
    solution        1.783
    comb_bound_low  1.776
    comb_bound_high 1.790
    comb_flags      1
    n_total         2^(11)
    n               2^(11)
    time_integrate  0.014
CubQMCSobolG (StoppingCriterion Object)
    abs_tol         0.010
    rel_tol         0
    n_init          2^(10)
    n_max           2^(35)
AsianOption (Integrand Object)
```

(continues on next page)

(continued from previous page)

```
    volatility      2^(-1)
    call_put        call
    start_price     30
    strike_price    35
    interest_rate   0
    mean_type       arithmetic
    dim_frac        0
BrownianMotion (TrueMeasure Object)
    time_vec        [0.031 0.062 0.094 ... 0.938 0.969 1.   ]
    drift           0
    mean            [0. 0. 0. ... 0. 0. 0.]
    covariance      [[0.031 0.031 0.031 ... 0.031 0.031 0.031]
                    [0.031 0.062 0.062 ... 0.062 0.062 0.062]
                    [0.031 0.062 0.094 ... 0.094 0.094 0.094]
                    ...
                    [0.031 0.062 0.094 ... 0.938 0.938 0.938]
                    [0.031 0.062 0.094 ... 0.938 0.969 0.969]
                    [0.031 0.062 0.094 ... 0.938 0.969 1.   ]]
    decomp_type     PCA
    transform       BrownianMotion (TrueMeasure Object)
                       time_vec        [0.031 0.062 0.094 ... 0.938 0.969 1.   ]
                       drift           1
                       mean            [0.031 0.062 0.094 ... 0.938 0.969 1.   ]
                       covariance      [[0.031 0.031 0.031 ... 0.031 0.031 0.031]
                                       [0.031 0.062 0.062 ... 0.062 0.062 0.062]
                                       [0.031 0.062 0.094 ... 0.094 0.094 0.094]
                                       ...
                                       [0.031 0.062 0.094 ... 0.938 0.938 0.938]
                                       [0.031 0.062 0.094 ... 0.938 0.969 0.969]
                                       [0.031 0.062 0.094 ... 0.938 0.969 1.   ]]
                       decomp_type     PCA
Sobol (DiscreteDistribution Object)
    d               2^(5)
    dvec            [ 0  1  2 ... 29 30 31]
    randomize       LMS_DS
    graycode        0
    entropy         144971093458566807495711477416245436939
    spawn_key       ()
```

```
print('Imporance Sampling takes %.3f the time and %.3f the samples'%\
     (data2.time_integrate/data1.time_integrate,data2.n_total/data1.n_total))
```

```
Imporance Sampling takes 1.009 the time and 0.500 the samples
```

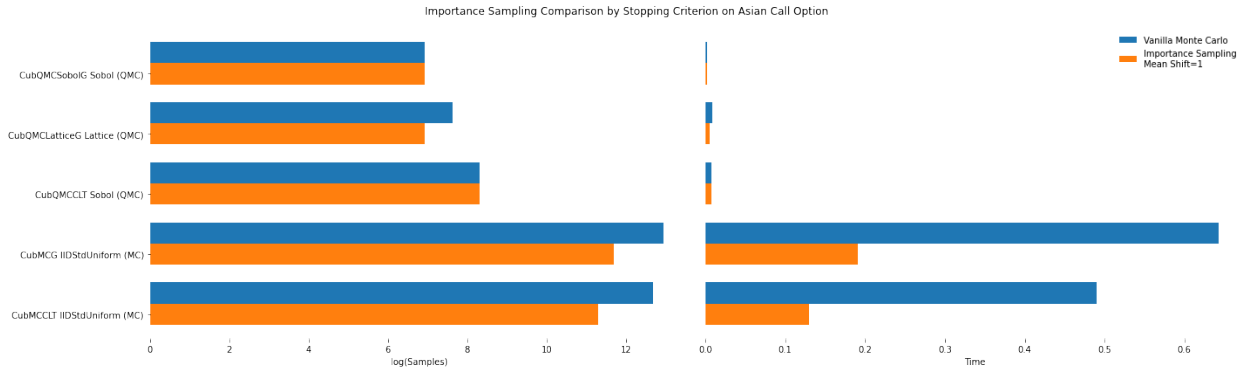
5.8.3 Importance Sampling MC vs QMC

Test Parameters

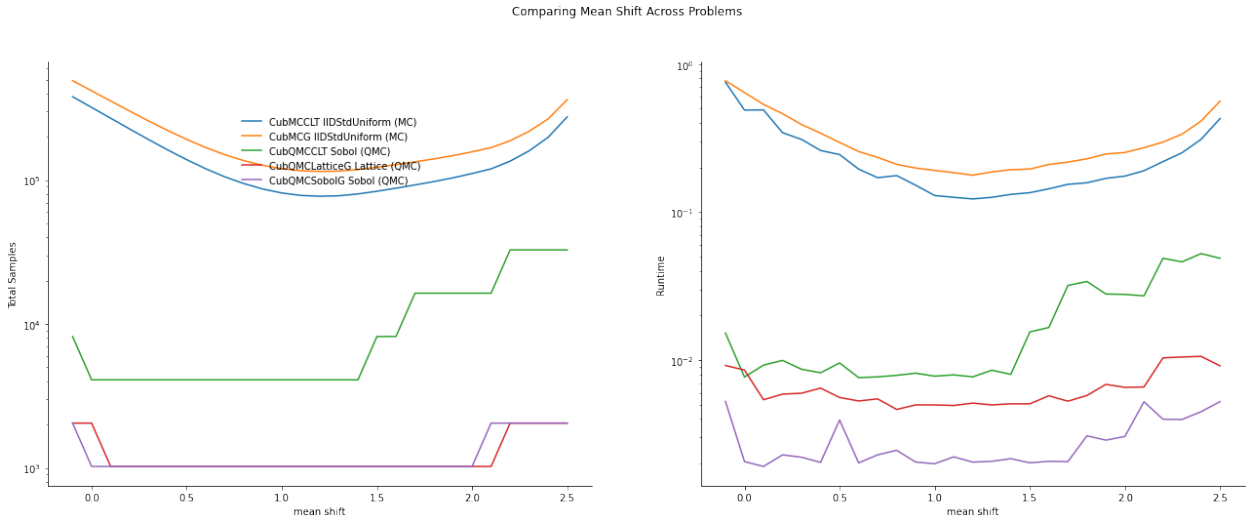
- dimension = 16
- abs_tol = .025
- trials = 3

```
df = pd.read_csv('../workouts/mc_vs_qmc/out/importance_sampling.csv')
df['Problem'] = df['Stopping Criterion'] + ' ' + df['Distribution'] + ' (' + df['MC/QMC
↪'] + ')'
df = df.drop(['Stopping Criterion','Distribution','MC/QMC'],axis=1)
problems = ['CubMCCLT IIDStdUniform (MC)',
            'CubMCG IIDStdUniform (MC)',
            'CubQMCCLT Sobol (QMC)',
            'CubQMCLatticeG Lattice (QMC)',
            'CubQMCSobolG Sobol (QMC)']
df = df[df['Problem'].isin(problems)]
mean_shifts = df.mean_shift.unique()
df_samples = df.groupby(['Problem'])['n_samples'].apply(list).reset_index(name='n')
df_times = df.groupby(['Problem'])['time'].apply(list).reset_index(name='time')
df.loc[(df.mean_shift==0) | (df.mean_shift==1)].set_index('Problem')
# Note: mean_shift==0 --> NOT using importance sampling
```

```
fig,ax = plt.subplots(nrows=1, ncols=2, figsize=(20, 6))
idx = arange(len(problems))
width = .35
ax[0].barh(idx+width,log(df.loc[df.mean_shift==0]['n_samples'].values),width)
ax[0].barh(idx,log(df.loc[df.mean_shift==1]['n_samples'].values),width)
ax[1].barh(idx+width,df.loc[df.mean_shift==0]['time'].values,width)
ax[1].barh(idx,df.loc[df.mean_shift==1]['time'].values,width)
fig.suptitle('Importance Sampling Comparison by Stopping Criterion on Asian Call Option')
xlabs = ['log(Samples)','Time']
for i in range(len(ax)):
    ax[i].set_xlabel(xlabs[i])
    ax[i].spines['top'].set_visible(False)
    ax[i].spines['bottom'].set_visible(False)
    ax[i].spines['right'].set_visible(False)
    ax[i].spines['left'].set_visible(False)
    ax[1].legend(['Vanilla Monte Carlo','Importance Sampling\nMean Shift=1'],loc='upper␣
↪right',frameon=False)
ax[1].get_yaxis().set_ticks([])
ax[0].set_yticks(idx)
ax[0].set_yticklabels(problems)
plt.tight_layout();
```




```
fig,ax = plt.subplots(nrows=1, ncols=2, figsize=(22, 8))
df_samples.apply(lambda row: ax[0].plot(mean_shifts,row.n,label=row['Problem']),axis=1)
df_times.apply(lambda row: ax[1].plot(mean_shifts,row.time,label=row['Problem']),axis=1)
ax[1].legend(frameon=False, loc=(-.85,.7),ncol=1)
ax[0].set_ylabel('Total Samples')
ax[0].set_yscale('log')
ax[1].set_yscale('log')
ax[1].set_ylabel('Runtime')
for i in range(len(ax)):
    ax[i].set_xlabel('mean shift')
    ax[i].spines['top'].set_visible(False)
    ax[i].spines['right'].set_visible(False)
fig.suptitle('Comparing Mean Shift Across Problems');
```



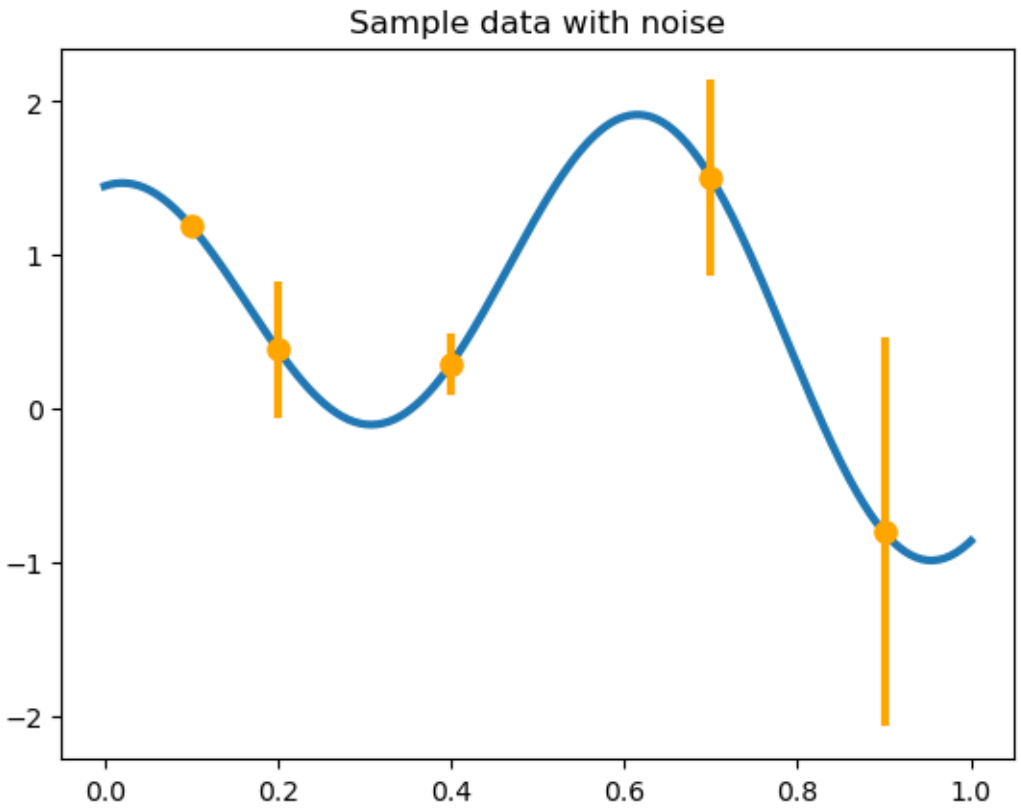
5.9 NEI (Noisy Expected Improvement) Demo

You can also look at the Botorch implementation, but that requires a lot more understanding of code which involves Pytorch. So we tried to put a simple example together here.

```
import numpy as np
import qmcpy as qp
from scipy.linalg import solve_triangular, cho_solve, cho_factor
from scipy.stats import norm
import matplotlib.pyplot as plt
%matplotlib inline

lw = 3
ms = 8
```

We make some fake data and consider the sequential decision-making problem of trying to optimize the function depicted below.

```
def yf(x):
    return np.cos(10 * x) * np.exp(.2 * x) + np.exp(-5 * (x - .4) ** 2)

xplt = np.linspace(0, 1, 300)
yplt = yf(xplt)

x = np.array([.1, .2, .4, .7, .9])
y = yf(x)
v = np.array([.001, .05, .01, .1, .4])

plt.plot(xplt, yplt, linewidth=lw)
plt.plot(x, y, 'o', markersize=ms, color='orange')
plt.errorbar(x, y, yerr=2 * np.sqrt(v), marker='', linestyle='', color='orange',␣
↪linewidth=3)
plt.title('Sample data with noise');
```



We can build a zero mean Gaussian process model to this data, observed under noise. Below are plots of the posterior distribution. We use the Gaussian (square exponential) kernel as our prior covariance belief.

This kernel has a shape parameter, the Gaussian process has a global variance, which are both chosen fixed for simplicity. The `fudge_factor` which is added here to prevent ill-conditioning for a large matrix.

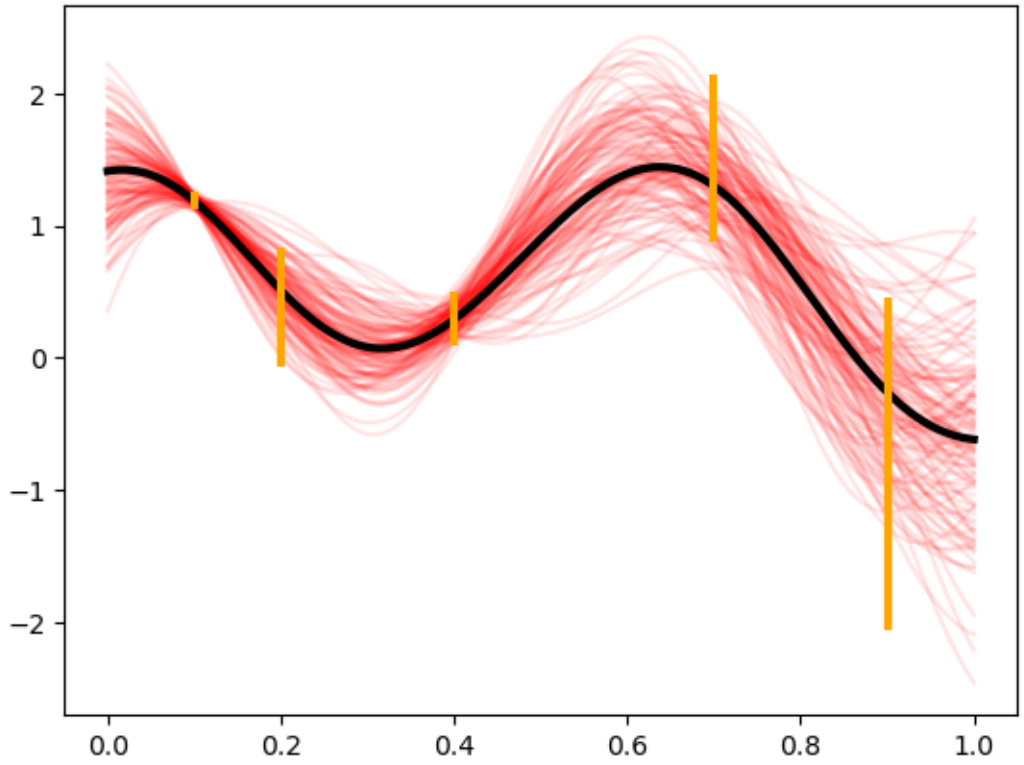
Notice the higher uncertainty in the posterior in locations where the observed noise is greater.

```
def gaussian_kernel(x, z, e, pv):
    return pv * np.exp(-e ** 2 * (x[:, None] - z[None, :]) ** 2)

shape_parameter = 4.1
process_variance = .9
fudge_factor = 1e-10

kernel_prior_data = gaussian_kernel(x, x, shape_parameter, process_variance)
kernel_cross_matrix = gaussian_kernel(xplt, x, shape_parameter, process_variance)
kernel_prior_plot = gaussian_kernel(xplt, xplt, shape_parameter, process_variance)

prior_cholesky = np.linalg.cholesky(kernel_prior_data + np.diag(v))
partial_cardinal_functions = solve_triangular(prior_cholesky, kernel_cross_matrix.T,␣
↪lower=True)
posterior_covariance = kernel_prior_plot - np.dot(partial_cardinal_functions.T, partial_
↪cardinal_functions)
posterior_cholesky = np.linalg.cholesky(posterior_covariance + fudge_factor * np.
↪eye(len(xplt)))

full_cardinal_functions = solve_triangular(prior_cholesky.T, partial_cardinal_functions,␣
↪lower=False)
```

(continues on next page)

(continued from previous page)

```
posterior_mean = np.dot(full_cardinal_functions.T, y)

num_posterior_draws = 123
normal_draws = np.random.normal(size=(num_posterior_draws, len(xplt)))
posterior_draws = posterior_mean[:, None] + np.dot(posterior_cholesky, normal_draws.T)

plt.plot(xplt, posterior_draws, alpha=.1, color='r')
plt.plot(xplt, posterior_mean, color='k', linewidth=lw)
plt.errorbar(x, y, yerr=2 * np.sqrt(v), marker='', linestyle='', color='orange',␣
↪linewidth=3);
```



First we take a look at the EI quantity by itself which, despite having a closed form, we will approximate using basic Monte Carlo below. The closed form is very preferable, but not applicable in all situations.

Expected improvement is just the expectation (under the posterior distribution) of the improvement beyond the current best value. If we were trying to maximize this function that we are studying then improvement would be defined as

$$I(x) = (Y_x|\mathcal{D} - y^*)_+,$$

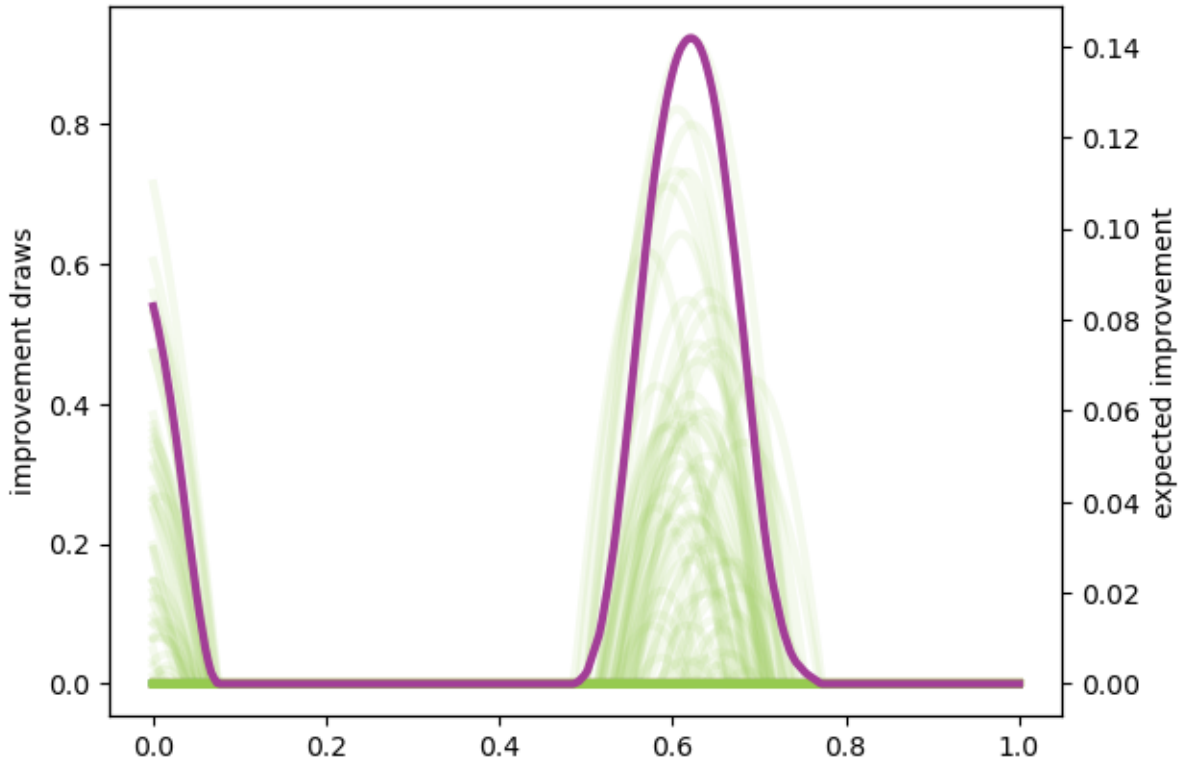
the positive part of the gap between the model $Y_x|\mathcal{D}$ and the current highest value $y^* = \max\{y_1, \ldots, y_N\}$. Since $Y_x|\mathcal{D}$ is a random variable (normally distributed because we have a Gaussian process model), we generally study the expected value of this, which is plotted below. Written as an integral, this would look like

$$\mathrm{EI}(x) = \int_{-\infty}^{\infty} (y - y^*)_+ \, p_{Y_x|\mathcal{D}}(y) \, \mathrm{d}y$$

NOTE: This quantity is written for maximization here, but most of the literature is concerned with minimization. We can rewrite this if needed, but the math is essentially the same.

This EI quantity is referred to as an *acquisition function*, a function which defines the utility associated with sampling at a given point. For each acquisition function, there is a balance between exploration and exploitation (as is the focus of most topics involving sequential decision-making under uncertainty).

```
improvement_draws = np.fmax(posterior_draws - max(y), 0)
plt.plot(xplt, improvement_draws, alpha=.1, color='#96CA4F', linewidth=lw)
plt.ylabel('improvement draws')
ax2 = plt.gca().twinx()
ax2.plot(xplt, np.mean(improvement_draws, axis=1), color='#A23D97', linewidth=lw)
ax2.set_ylabel('expected improvement');
```



The NEI quantity is then computed using multiple EI computations (each using a different posterior GP draw) computed without noise. In this computation below, we will use the closed form of EI, to speed up the computation – it is possible to execute the same strategy as above, though.

This computation is vectorized so as to compute for multiple x locations at the same time. The algorithm from the Facebook paper is written for only a single location. We are omitting the constraints aspect of their paper because the problem can be considered without that. To define the integral, though, we need some more definitions/notation.

First, we need to define $\text{EI}(x; \mathbf{y}, \mathcal{X}, \boldsymbol{\epsilon})$ to be the expected improvement at a location x, given the N values stored in the vector $\mathbf{y}$ having been evaluated with noise $\boldsymbol{\epsilon}$ at the points $\mathcal{X}$,

$$\mathbf{y} = \begin{pmatrix} y_1 \\ \vdots \\ y_N \end{pmatrix}, \qquad \mathcal{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}, \qquad \boldsymbol{\epsilon} = \begin{pmatrix} \epsilon_1 \\ \vdots \\ \epsilon_N \end{pmatrix}.$$

The noise is assumed to be $\epsilon_i \sim \mathcal{N}(0, \sigma^2)$ for some fixed σ^2. The noise need not actually be homoscedastic, but it is a standard assumption. We encapsulate this information in $\mathcal{D} = \{\mathbf{y}, \mathcal{X}, \boldsymbol{\epsilon}\}$. This is omitted from the earlier notation, because the data would be fixed.

The point of NEI though is to deal with **noisy** observed values (EI, itself, is notorious for not dealing with noisy data very well). It does this by considering a variety of posterior draws at the locations in $\mathcal{X}$. These have distribution

$$Y_{\mathcal{X}}|\mathcal{D} = Y_{\mathcal{X}}|\mathbf{y}, \mathcal{X}, \boldsymbol{\epsilon} \sim \mathcal{N}\left(\mathsf{K}(\mathsf{K}+\mathsf{E})^{-1}\mathbf{y}, \mathsf{K} - \mathsf{K}(\mathsf{K}+\mathsf{E})^{-1}\mathsf{K}\right),$$

where

$$\mathbf{k}(x) = \begin{pmatrix} K(x,x_1) \\ \vdots \\ K(x,x_N) \end{pmatrix}, \qquad \mathsf{K} = \begin{pmatrix} K(x_1,x_1) & \cdots & K(x_1,x_N) \\ & \vdots & \\ K(x_N,x_1) & \cdots & K(x_N,x_N) \end{pmatrix} = \begin{pmatrix} \mathbf{k}(x_1)^T \\ \vdots \\ \mathbf{k}(x_N)^T \end{pmatrix}, \qquad \mathsf{E} = \begin{pmatrix} \epsilon_1 & & \\ & \ddots & \\ & & \epsilon_N \end{pmatrix}$$

In practice, unless noise has actually been measured at each point, it would be common to simply plug in $\epsilon_1 = \ldots = \epsilon_N = \sigma^2$. The term `noisy_predictions_at_data` below is drawn from this distribution (though in a standard iid fashion, not a more awesome QMC fashion).

The EI integral, although approximated earlier using Monte Carlo, can actually be written in closed form. We do so below to also solidify our newer notation:

$$\mathrm{EI}(x; \mathbf{y}, \mathcal{X}, \boldsymbol{\epsilon}) = \int_{-\infty}^{\infty} (y - y^*)_+ \, p_{Y_x|\mathbf{y},\mathcal{X},\boldsymbol{\epsilon}}(y) \, \mathrm{d}y = s(z\Phi(z) + \phi(z))$$

where ϕ and Φ are the standard normal pdf and cdf, and

$$\mu = \mathbf{k}(x)^T(\mathsf{K}+\mathsf{E})^{-1}\mathbf{y}, \qquad s^2 = K(x,x) - \mathbf{k}(x)^T(\mathsf{K}+\mathsf{E})^{-1}\mathbf{k}(x), \qquad z = (\mu - y^*)/s.$$

It is very important to remember that these quantities are functions of $\mathbf{y}, \mathcal{X}, \boldsymbol{\epsilon}$ despite the absence of those quantities in the notation.

The goal of the NEI integral is to simulate many possible random realizations of what could actually be the truth at the locations $\mathcal{X}$ and then run a *noiseless* EI computation over each of those realizations. The average of these outcomes is the NEI quantity. This would look like:

$$\mathrm{NEI}(x) = \int_{\mathbf{f}\in\mathbb{R}^N} \mathrm{EI}(x; \mathbf{f}, \mathcal{X}, 0) \, p_{Y_{\mathcal{X}}|\mathbf{y},\mathcal{X},\boldsymbol{\epsilon}}(\mathbf{f}) \, \mathrm{d}\mathbf{f}$$

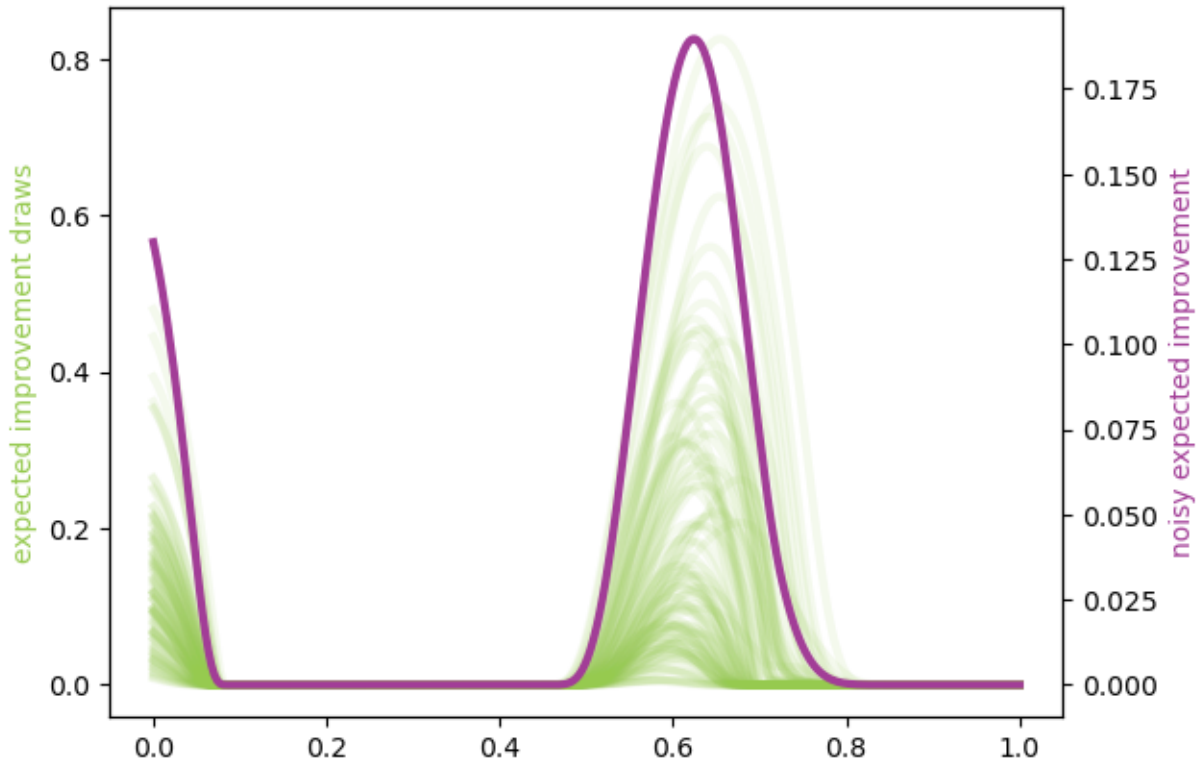
NOTE: There are ways to do this computation in a more vectorized fashion, so it would more likely be a loop involving chunks of MC elements at a time. Just so you know.

```
num_draws_at_data = 109
# These draws are done through QMC in the FB paper
normal_draws_at_data = np.random.normal(size=(num_draws_at_data, len(x)))

partial_cardinal_functions_at_data = solve_triangular(prior_cholesky, kernel_prior_data.
↪T, lower=True)
posterior_covariance_at_data = kernel_prior_data - np.dot(partial_cardinal_functions_at_
↪data.T, partial_cardinal_functions_at_data)
posterior_cholesky_at_data = np.linalg.cholesky(posterior_covariance_at_data + fudge_
↪factor * np.eye(len(x)))

noisy_predictions_at_data = y[:, None] + np.dot(posterior_cholesky_at_data, normal_draws_
↪at_data.T)

prior_cholesky_noiseless = np.linalg.cholesky(kernel_prior_data)
partial_cardinal_functions = solve_triangular(prior_cholesky_noiseless, kernel_cross_
↪matrix.T, lower=True)
full_cardinal_functions = solve_triangular(prior_cholesky.T, partial_cardinal_functions,␣
↪lower=False)
```

(continues on next page)

(continued from previous page)

```
pointwise_sd = np.sqrt(np.fmax(process_variance - np.sum(partial_cardinal_functions ** 2,
↪ axis=0), 1e-100))

all_noiseless_eis = []
for draw in noisy_predictions_at_data.T:
    posterior_mean = np.dot(full_cardinal_functions.T, draw)

    z = (posterior_mean - max(y)) / pointwise_sd
    ei = pointwise_sd * (z * norm.cdf(z) + norm.pdf(z))

    all_noiseless_eis.append(ei)

all_noiseless_eis = np.array(all_noiseless_eis)

plt.plot(xplt, all_noiseless_eis.T, alpha=.1, color='#96CA4F', linewidth=lw)
plt.ylabel('expected improvement draws', color='#96CA4F')
ax2 = plt.gca().twinx()
ax2.plot(xplt, np.mean(all_noiseless_eis, axis=0), color='#A23D97', linewidth=lw)
ax2.set_ylabel('noisy expected improvement', color='#A23D97');
```



5.9.1 Goal

What would be really great would be if we could compute integrals like the EI integral or the NEI integral using QMC. If there are opportunities to use the latest research to adaptively study tolerance and truncate, that would be absolutely amazing.

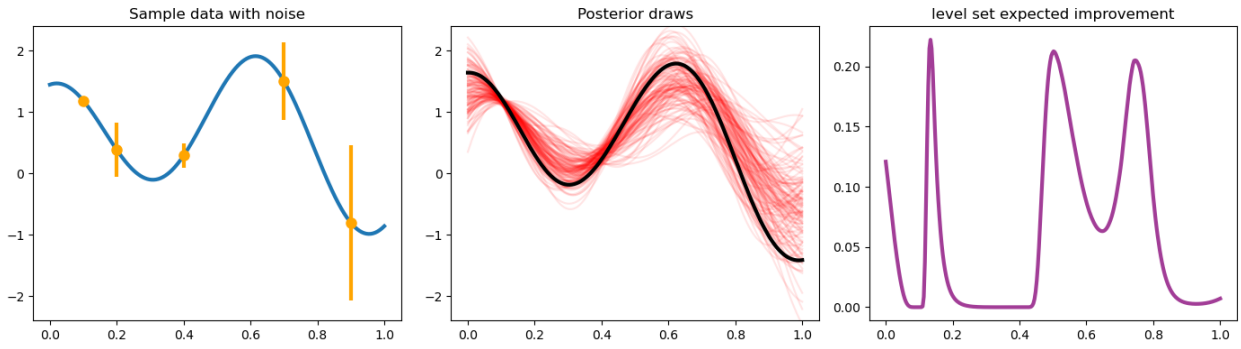
We put the NEI example up first because the FB crew has already done a great job showing how QMC can play a role. But, as you can see, NEI is more complicated than EI, and also not yet as popular in the community (though that may change).

Bonus stuff

Even the EI integral, which does have a closed form, might better be considered in a QMC fashion because of interesting use cases. We are going to reconsider the same problem from above, but here we am not looking to maximize the function – we want to find the "level set" associated with the value $y = 1$. Below you can see how the different outcome might look.

In this case, the quantity of relevance is not exactly an integral, but it is a function of this posterior mean and standard deviation, which might need to be estimated through an integral (rather than the closed form, which we do have for a GP situation).

```
fig, axes = plt.subplots(1, 3, figsize=(14, 4))

ax = axes[0]
ax.plot(xplt, yplt, linewidth=lw)
ax.plot(x, y, 'o', markersize=ms, color='orange')
ax.errorbar(x, y, yerr=2 * np.sqrt(v), marker='', linestyle='', color='orange',␣
↪linewidth=3)
ax.set_title('Sample data with noise')
ax.set_ylim(-2.4, 2.4)

ax = axes[1]
ax.plot(xplt, posterior_draws, alpha=.1, color='r')
ax.plot(xplt, posterior_mean, color='k', linewidth=lw)
ax.set_title('Posterior draws')
ax.set_ylim(-2.4, 2.4)

ax = axes[2]
posterior_mean_distance_from_1 = np.mean(np.abs(posterior_draws - 1), axis=1)
posterior_standard_deviation = np.std(posterior_draws, axis=1)
level_set_expected_improvement = norm.cdf(-posterior_mean_distance_from_1 / posterior_
↪standard_deviation)
ax.plot(xplt, level_set_expected_improvement, color='#A23D97', linewidth=lw)
ax.set_title('level set expected improvement')

plt.tight_layout();
```



5.9.2 Computation of the QEI quantity using qmcpy

NEI is an important quantity, but there are other quantities as well which could be considered relevant demonstrations of higher dimensional integrals.

One such quantity is a computation involving q "next points" to sample in a BO process; in the standard formulation this quantity might involve just $q = 1$, but $q > 1$ is also of interest for batched evaluation in parallel.

This quantity is defined as

$$\mathrm{EI}_q(x_1, \ldots, x_q; \mathbf{y}, \mathcal{X}, \boldsymbol{\epsilon}) = \int_{\mathbb{R}^q} \max_{1 \le i \le q} [(y_i - y^*)_+] \; p_{Y_{x_1,\ldots,x_q}|\mathbf{y},\mathcal{X},\boldsymbol{\epsilon}}(y_1, \ldots, y_q) \, \mathrm{d}y_1 \cdots \mathrm{d}y_q$$

The example we are considering here is with $q = 5$ but this quantity could be made larger. Each of these QEI computations (done in a vectorized fashion in production) would be needed in an optimization loop (likely powered by CMAES or some other high dimensional non-convex optimization tool). This optimization problem would take place in a qd dimensional space, which is one aspect which usually prevents q from being too large.

Note that some of this will look much more confusing in $d > 1$, but it is written here in a simplified version.

```
q = 5  # number of "next points" to be considered simultaneously
next_x = np.array([0.158,  0.416,  0.718,  0.935,  0.465])

def compute_qei(next_x, mc_strat, num_posterior_draws):
    q = len(next_x)

    kernel_prior_data = gaussian_kernel(x, x, shape_parameter, process_variance)
    kernel_cross_matrix = gaussian_kernel(next_x, x, shape_parameter, process_variance)
    kernel_prior_plot = gaussian_kernel(next_x, next_x, shape_parameter, process_
↪variance)
    prior_cholesky = np.linalg.cholesky(kernel_prior_data + np.diag(v))

    partial_cardinal_functions = solve_triangular(prior_cholesky, kernel_cross_matrix.T,␣
↪lower=True)
    posterior_covariance = kernel_prior_plot - np.dot(partial_cardinal_functions.T,␣
↪partial_cardinal_functions)
    posterior_cholesky = np.linalg.cholesky(posterior_covariance + fudge_factor * np.
↪eye(q))

    full_cardinal_functions = solve_triangular(prior_cholesky.T, partial_cardinal_
↪functions, lower=False)
    posterior_mean = np.dot(full_cardinal_functions.T, y)
```

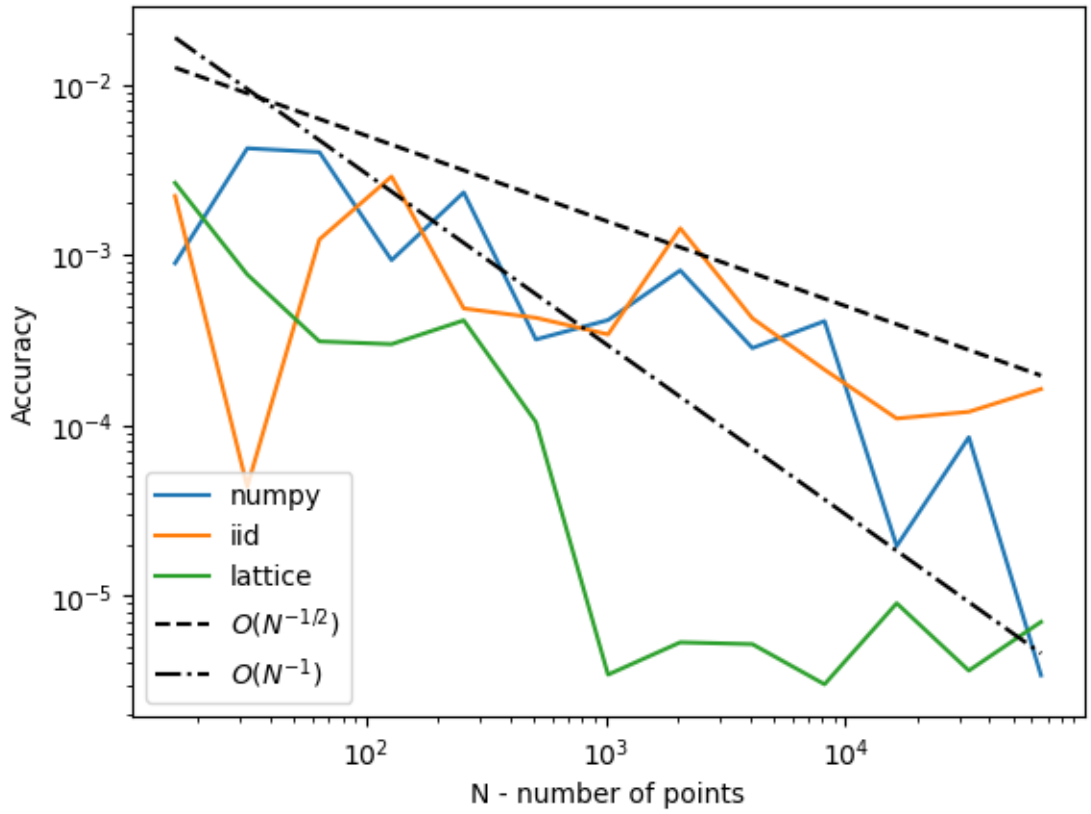
(continues on next page)

(continued from previous page)

```
    if mc_strat == 'numpy':
        normal_draws = np.random.normal(size=(num_posterior_draws, q))
    elif mc_strat == 'lattice':
        g =  qp.Gaussian(qp.Lattice(dimension=q, randomize=True))
        normal_draws = g.gen_samples(n=num_posterior_draws)
    else:
        g =  qp.Gaussian(qp.IIDStdUniform(dimension=q))
        normal_draws = g.gen_samples(n = num_posterior_draws)
    posterior_draws = posterior_mean[:, None] + np.dot(posterior_cholesky, normal_draws.
↪T)

    return np.mean(np.fmax(np.max(posterior_draws[:, :num_posterior_draws] - max(y),␣
↪axis=0), 0))
```

```
num_posterior_draws_to_test = 2 ** np.arange(4, 17)
trials = 10

vals = {}
for mc_strat in ('numpy', 'iid', 'lattice'):
    vals[mc_strat] = []

    for num_posterior_draws in num_posterior_draws_to_test:
        qei_estimate = 0.
        for trial in range(trials):
            qei_estimate += compute_qei(next_x, mc_strat, num_posterior_draws)
        avg_qei_estimate = qei_estimate/float(trials)
        vals[mc_strat].append(avg_qei_estimate)

    vals[mc_strat] = np.array(vals[mc_strat])
#reference_answer = compute_qei(next_x, 'lattice', 2 ** 7 * max(num_posterior_draws_to_
↪test))
reference_answer = compute_qei(next_x, 'lattice', 2 ** 20)
```

```
for name, results in vals.items():
    plt.loglog(num_posterior_draws_to_test, abs(results - reference_answer), label=name)
plt.loglog(num_posterior_draws_to_test, .05 * num_posterior_draws_to_test ** -.5, '--k',␣
↪label='$O(N^{-1/2})$')
plt.loglog(num_posterior_draws_to_test, .3 * num_posterior_draws_to_test ** -1.0, '-.k',␣
↪label='$O(N^{-1})$')
plt.xlabel('N - number of points')
plt.ylabel('Accuracy')
plt.legend(loc='lower left');
```



This is very similar to what the FB paper talked about, and we think exactly the kind of thing we should be emphasizing in our discussions in a potential blog post which talks about BO applications of QMC.

Such a blog post is something that we would be happy to write up, by the way.

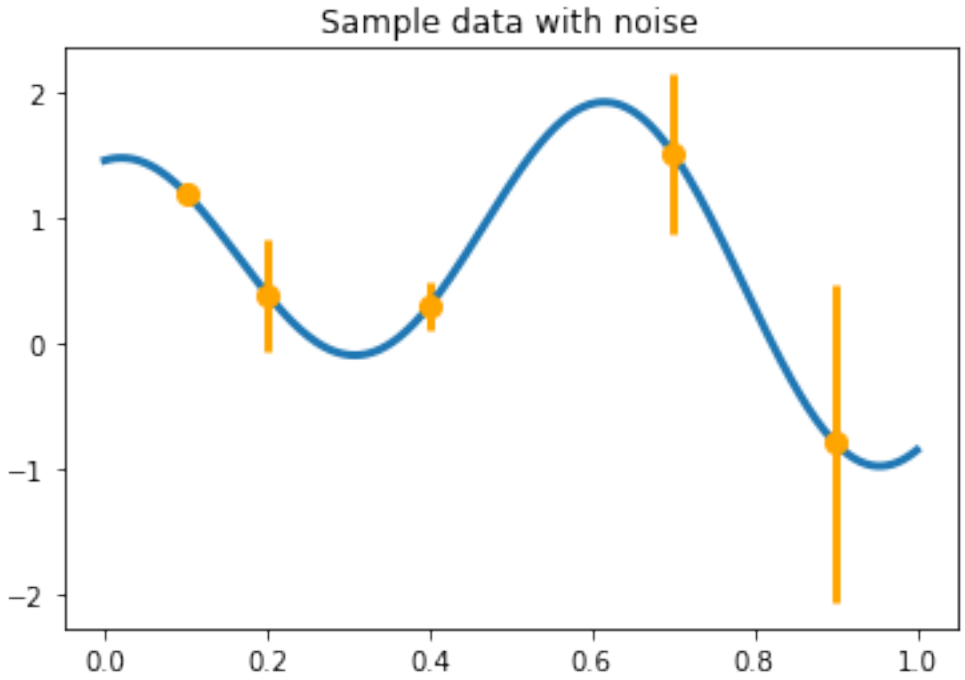
5.10 QEI (Q-Noisy Expected Improvement) Demo for Blog

```
from qmcpy import *
import numpy as np
from scipy.linalg import solve_triangular, cho_solve, cho_factor
from scipy.stats import norm
import matplotlib.pyplot as pyplot
%matplotlib inline

lw = 3
ms = 8
```

5.10.1 Problem setup

Here is the current data (x and y values with noise) from which we want to build a GP and run a Bayesian optimization.

```
def yf(x):
    return np.cos(10 * x) * np.exp(.2 * x) + np.exp(-5 * (x - .4) ** 2)

xplt = np.linspace(0, 1, 300)
yplt = yf(xplt)

x = np.array([.1, .2, .4, .7, .9])
y = yf(x)
v = np.array([.001, .05, .01, .1, .4])

pyplot.plot(xplt, yplt, linewidth=lw)
pyplot.plot(x, y, 'o', markersize=ms, color='orange')
pyplot.errorbar(x, y, yerr=2 * np.sqrt(v), marker='', linestyle='', color='orange',␣
↪linewidth=lw)
pyplot.title('Sample data with noise');
```



5.10.2 Computation of the qEI quantity using `qmcpy`

One quantity which can appear often during BO is a computation involving q "next points" to sample in a BO process; in the standard formulation this quantity might involve just $q = 1$, but $q > 1$ is also of interest for batched evaluation in parallel.

This quantity is defined as

$$\mathrm{EI}_q(x_1, \ldots, x_q; \mathbf{y}, \mathcal{X}, \boldsymbol{\epsilon}) = \int_{\mathbb{R}^q} \max_{1 \le i \le q} [(y_i - y^*)_+] \; p_{Y_{x_1,\ldots,x_q}|\mathbf{y},\mathcal{X},\boldsymbol{\epsilon}}(y_1, \ldots, y_q) \, \mathrm{d}y_1 \cdots \mathrm{d}y_q$$

The example we are considering here is with $q = 5$ but this quantity could be made larger. Each of these QEI computations (done in a vectorized fashion in production) would be needed in an optimization loop (likely powered by CMAES or some other high dimensional non-convex optimization tool). This optimization problem would take place in a qd dimensional space, which is one aspect which usually prevents q from being too large.

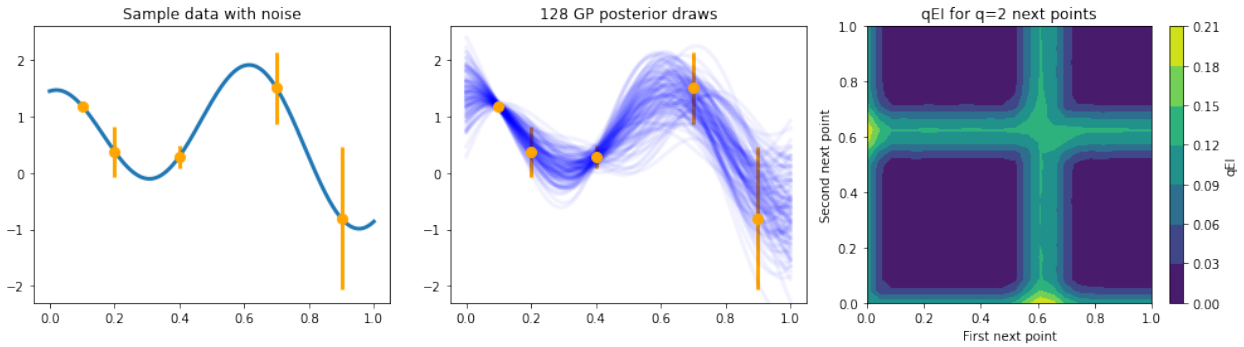
Note that some of this will look much more confusing in $d > 1$, but it is written here in a simplified version.

5.10.3 GP model definition (kernel information) and qEI definition

```
shape_parameter = 4.1
process_variance = .9
fudge_factor = 1e-10

def gaussian_kernel(x, z):
    return process_variance * np.exp(-shape_parameter ** 2 * (x[:, None] - z[None, :])␣
↪** 2)

def gp_posterior_params(x_to_draw):
    n = len(x_to_draw)

    kernel_prior_data = gaussian_kernel(x, x)
    kernel_cross_matrix = gaussian_kernel(x_to_draw, x)
    kernel_prior_plot = gaussian_kernel(x_to_draw, x_to_draw)
    prior_cholesky = np.linalg.cholesky(kernel_prior_data + np.diag(v))

    partial_cardinal_functions = solve_triangular(prior_cholesky, kernel_cross_matrix.T,␣
↪lower=True)
    posterior_covariance = kernel_prior_plot - np.dot(partial_cardinal_functions.T,␣
↪partial_cardinal_functions) + fudge_factor * np.eye(n)

    full_cardinal_functions = solve_triangular(prior_cholesky.T, partial_cardinal_
↪functions, lower=False)
    posterior_mean = np.dot(full_cardinal_functions.T, y)
    return posterior_mean,posterior_covariance

def gp_posterior_draws(x_to_draw, mc_strat, num_posterior_draws, posterior_mean,␣
↪posterior_covariance):
    q = len(x_to_draw)
    if mc_strat == 'iid':
        dd = IIDStdUniform(q)
    elif mc_strat == 'lattice':
        dd = Lattice(q)
    elif mc_strat == 'sobol':
        dd = Sobol(q)
    g = Gaussian(dd,posterior_mean,posterior_covariance)
    posterior_draws = g.gen_samples(num_posterior_draws)
    return posterior_draws

def compute_qei(posterior_draws):
    y_gp = np.fmax(np.max(posterior_draws.T - max(y), axis=0), 0)
    return y_gp
```

5.10.4 Demonstrate the concept of qEI on 2 points

```
num_posterior_draws = 2 ** 7
Np = (25, 24)
X, Y = np.meshgrid(np.linspace(0, 1, Np[1]), np.linspace(0, 1, Np[0]))
xp = np.array([X.reshape(-1), Y.reshape(-1)]).T
mu_post,sigma_cov = gp_posterior_params(xplt)
y_draws = gp_posterior_draws(xplt, 'lattice', num_posterior_draws,mu_post,sigma_cov).T
qei_vals = np.empty(len(xp))
for k, next_x in enumerate(xp):
    mu_post,sigma_cov = gp_posterior_params(next_x)
    gp_draws = gp_posterior_draws(next_x, 'sobol', num_posterior_draws,mu_post,sigma_cov)
    qei_vals[k] = compute_qei(gp_draws).mean()
Z = qei_vals.reshape(Np)

fig, axes = pyplot.subplots(1, 3, figsize=(14, 4))

ax = axes[0]
ax.plot(xplt, yplt, linewidth=lw)
ax.plot(x, y, 'o', markersize=ms, color='orange')
ax.errorbar(x, y, yerr=2 * np.sqrt(v), marker='', linestyle='', color='orange',␣
↪linewidth=lw)
ax.set_title('Sample data with noise')
ax.set_ylim((-2.3, 2.6))

ax = axes[1]
ax.plot(xplt, y_draws, linewidth=lw, color='b', alpha=.05)
ax.plot(x, y, 'o', markersize=ms, color='orange')
ax.errorbar(x, y, yerr=2 * np.sqrt(v), marker='', linestyle='', color='orange',␣
↪linewidth=lw)
ax.set_title(f'{num_posterior_draws} GP posterior draws')
ax.set_ylim((-2.3, 2.6))

ax = axes[2]
h = ax.contourf(X, Y, Z)
ax.set_xlabel('First next point')
ax.set_ylabel('Second next point')
ax.set_title('qEI for q=2 next points')
cax = fig.colorbar(h, ax=ax)
cax.set_label('qEI')

fig.tight_layout()
```



5.10.5 Choose some set of next points against which to test the computation

Here, we consider $q = 5$, which is much more costly to compute than the $q = 2$ demonstration above.

Note This will take some time to run. Use fewer `num_repeats` to reduce the cost.

```
# parameters
next_x = np.array([0.158,  0.416,  0.718,  0.935,  0.465])
num_posterior_draws_to_test = 2 ** np.arange(4, 20)
d = len(next_x)
mu_post,sigma_cov = gp_posterior_params(next_x)
```

```
# get reference answer with qmcpy
integrand = CustomFun(
    true_measure = Gaussian(Sobol(d),mu_post,sigma_cov),
    g = compute_qei)
stopping_criterion = CubQMCSobolG(integrand, abs_tol=5e-7)
reference_answer,data = stopping_criterion.integrate()
print(data)
```

```
Solution: 0.0244
CustomFun (Integrand Object)
Sobol (DiscreteDistribution Object)
    d               5
    randomize       1
    graycode        0
    seed            [68566 92413 32829 13022 69017]
    mimics          StdUniform
    dim0            0
Gaussian (TrueMeasure Object)
    mean            [ 0.807  0.371  1.204 -0.455  0.67 ]
    covariance      [[ 1.845e-02 -2.039e-03  1.150e-04  1.219e-04 -5.985e-03]
                    [-2.039e-03  1.355e-02  6.999e-04 -1.967e-03  2.302e-02]
                    [ 1.150e-04  6.999e-04  8.871e-02  2.043e-02  5.757e-03]
                    [ 1.219e-04 -1.967e-03  2.043e-02  2.995e-01 -9.482e-03]
                    [-5.985e-03  2.302e-02  5.757e-03 -9.482e-03  6.296e-02]]
    decomp_type     pca
CubQMCSobolG (StoppingCriterion Object)
    abs_tol         5.00e-07
    rel_tol         0
```

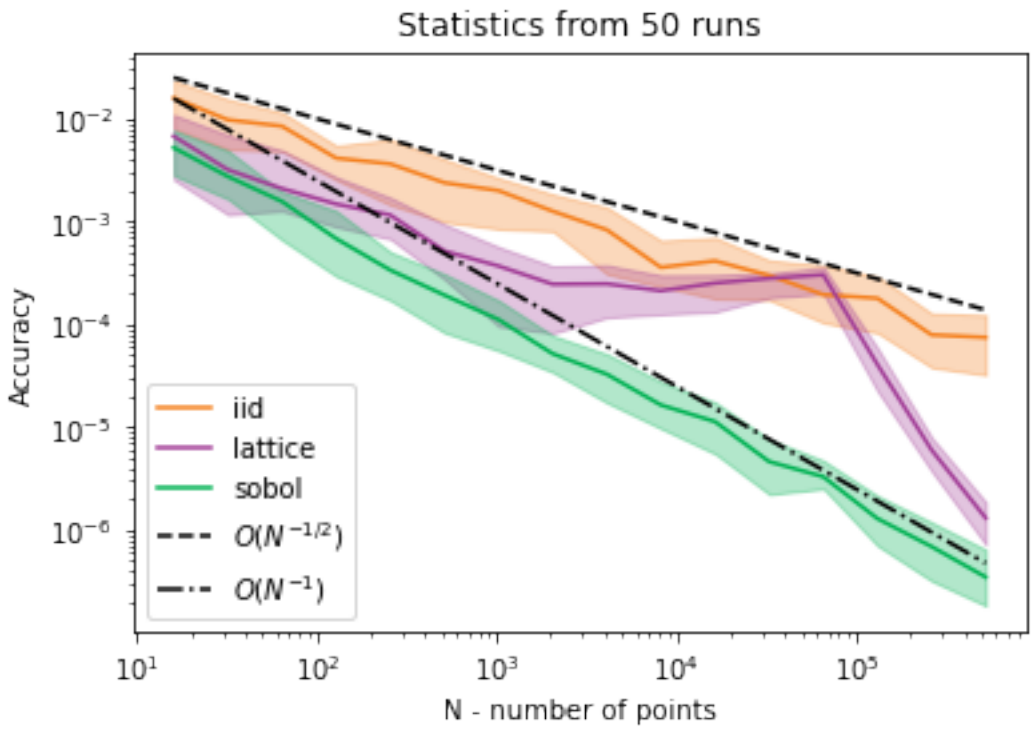
(continues on next page)

(continued from previous page)

```
    n_init          2^(10)
    n_max           2^(35)
LDTransformData (AccumulateData Object)
    n_total         2^(22)
    solution        0.024
    error_bound     4.15e-07
    time_integrate  4.426
```

```
# generate data
num_posterior_draws_to_test = 2 ** np.arange(4, 20)
vals = {}
num_repeats = 50
mc_strats = ('iid', 'lattice', 'sobol')
for mc_strat in mc_strats:
    vals[mc_strat] = []
    for num_posterior_draws in num_posterior_draws_to_test:
        all_estimates = []
        for _ in range(num_repeats):
            y_draws = gp_posterior_draws(next_x, mc_strat, num_posterior_draws,mu_post,
↪sigma_cov)
            all_estimates.append(compute_qei(y_draws).mean())
        vals[mc_strat].append(all_estimates)
    vals[mc_strat] = np.array(vals[mc_strat])
```

```
fig, ax = pyplot.subplots(1, 1, figsize=(6, 4))

colors = ('#F5811F', '#A23D97', '#00B253')
alpha = .3

for (name, results), color in zip(vals.items(), colors):
    bot = np.percentile(abs(results - reference_answer), 25, axis=1)
    med = np.percentile(abs(results - reference_answer), 50, axis=1)
    top = np.percentile(abs(results - reference_answer), 75, axis=1)
    ax.loglog(num_posterior_draws_to_test, med, label=name, color=color)
    ax.fill_between(num_posterior_draws_to_test, bot, top, color=color, alpha=alpha)
ax.loglog(num_posterior_draws_to_test, .1 * num_posterior_draws_to_test ** -.5, '--k',␣
↪label='$O(N^{-1/2})$')
ax.loglog(num_posterior_draws_to_test, .25 * num_posterior_draws_to_test ** -1.0, '-.k',␣
↪label='$O(N^{-1})$')
ax.set_xlabel('N - number of points')
ax.set_ylabel('Accuracy')
ax.legend(loc='lower left')
ax.set_title(f'Statistics from {num_repeats} runs');

# plt.savefig('qei_convergence.png');
```




```
# parameters
names = ['IID','Lattice','Sobol']
epsilons = [
      [2e-4, 5e-4, 1e-3, 2e-3, 5e-3, 1e-2], # iid nodes
      [5e-6, 1e-5, 2e-5, 5e-5, 1e-4, 2e-4, 5e-4, 1e-3, 2e-3, 5e-3, 1e-2], # lattice
      [5e-6, 1e-5, 2e-5, 5e-5, 1e-4, 2e-4, 5e-4, 1e-3, 2e-3, 5e-3, 1e-2]] # sobol
trials = 25
# initialize time data
times = {names[j]:np.zeros((len(epsilons[j]),trials),dtype=float) for j in␣
↪range(len(names))}
n_needed = {names[j]:np.zeros((len(epsilons[j]),trials),dtype=float) for j in␣
↪range(len(names))}
# run tests
for t in range(trials):
  print(t)
  for j in range(len(names)):
    for i in range(len(epsilons[j])):
      if j == 0:
        sc = CubMCG(CustomFun(Gaussian(IIDStdUniform(d),mu_post,sigma_cov),compute_qei),
↪abs_tol=epsilons[j][i],rel_tol=0)
      elif j == 1:
        sc = CubQMCLatticeG(CustomFun(Gaussian(Lattice(d),mu_post,sigma_cov),compute_
↪qei),abs_tol=epsilons[j][i],rel_tol=0)
      else:
        sc = CubQMCSobolG(CustomFun(Gaussian(Sobol(d),mu_post,sigma_cov),compute_qei),
↪abs_tol=epsilons[j][i],rel_tol=0)
      solution,data = sc.integrate()
      times[names[j]][i,t] = data.time_integrate
      n_needed[names[j]][i,t] = data.n_total
```

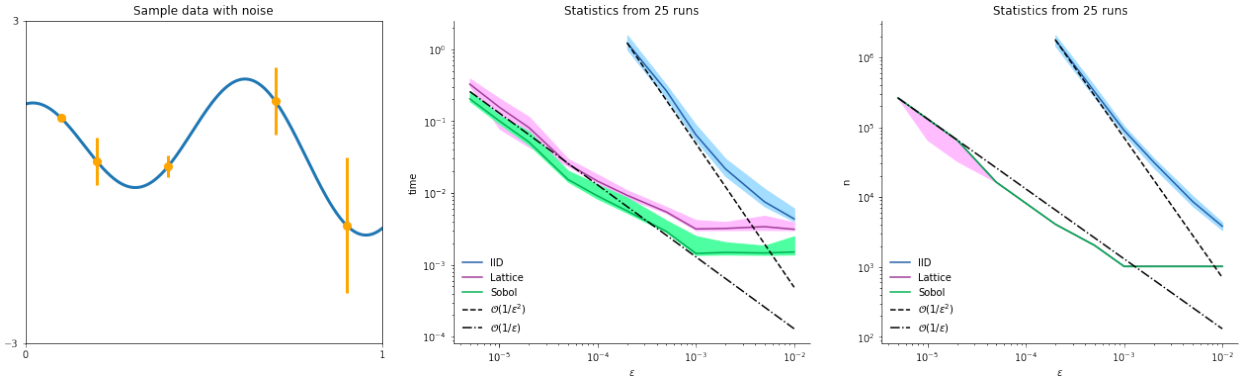
```
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
```

```
fig,axs = pyplot.subplots(1, 3, figsize=(22, 6))
colors = ('#245EAB', '#A23D97', '#00B253')
light_colors = ('#A3DDFF', '#FFBCFF', '#4DFFA0')
alpha = .3
def plot_fills(eps,data,name,color,light_color):
    bot = np.percentile(data, 5, axis=1)
    med = np.percentile(data, 50, axis=1)
    top = np.percentile(data, 95, axis=1)
    ax.loglog(eps, med, label=name, color=color)
    ax.fill_between(eps, bot, top, color=light_color)
    return med
for i,(nt_data,label) in enumerate(zip([times,n_needed],['time','n'])):
  ax = axs[i+1]
  # iid plot
  eps_iid = np.array(epsilons[0])
  data = nt_data['IID']
  med_iid = plot_fills(eps_iid,data,'IID',colors[0],light_colors[0])
  # lattice plot
  eps = np.array(epsilons[1])
  data = nt_data['Lattice']
  med_lattice = plot_fills(eps,data,'Lattice',colors[1],light_colors[1])
  # sobol plot
  eps = np.array(epsilons[2])
  data = nt_data['Sobol']
  med_sobol = plot_fills(eps,data,'Sobol',colors[2],light_colors[2])
  # iid bigO
```

(continues on next page)

(continued from previous page)

```
  ax.loglog(eps_iid, (med_iid[0]*eps_iid[0]**2)/(eps_iid**2), '--k', label=r'$\mathcal{O}
↪(1/\epsilon^2)$')
  # ld bigO
  ax.loglog(eps, ((med_lattice[0]*med_sobol[0])**.5 *eps[0]) / eps , '-.k', label=r'$\
↪mathcal{O}(1/\epsilon)$')
  # metas
  ax.set_xlabel(r'$\epsilon$')
  ax.set_ylabel(label)
  ax.spines['right'].set_visible(False)
  ax.spines['top'].set_visible(False)
  ax.legend(loc='lower left',frameon=False)
  ax.set_title(f'Statistics from {trials} runs')
# plot sample data
ax = axs[0]
ax.plot(xplt, yplt, linewidth=lw)
ax.plot(x, y, 'o', markersize=ms, color='orange')
ax.errorbar(x, y, yerr=2 * np.sqrt(v), marker='', linestyle='', color='orange',␣
↪linewidth=lw)
ax.set_title('Sample data with noise')
ax.set_xlim([0,1])
ax.set_xticks([0,1])
ax.set_ylim([-3, 3])
ax.set_yticks([-3,3]);
```



5.11 Basic Ray Tracing

- Introduction to Ray Tracing: a Simple Method for Creating 3D Images by Scratchapixel 2.0
- Computer Graphics from scratch by Gabriel Gambetta
- Ray Tracing: Graphics for the Masses by Paul Rademacher

```
!pip install pillow --quiet
```

```
from qmcpy import *
from numpy import *
from time import time
from PIL import Image
```

(continues on next page)

(continued from previous page)

```
from matplotlib import pyplot
from threading import Thread
%matplotlib inline
```

```
# constants
EPS = 1e-8 # numerical precision error tolerance
norm = lambda v: sqrt(dot(v,v))
e = array([0,0,0],dtype=float) # eye location at the orgin
```

```
class Camera(object):
    """ An object to render the scene. """
    def __init__(self, ax, scene, px=8, parallel_x_blocks=2, parallel_y_blocks=2, image_
↪angle=90, image_dist=1):
        """
        Args:
            ax (axes): matplotlib ax to plot image on
            scene (int): object with scene.render_px(p) method. See examples later in␣
↪the notebook.
            px (int): number of pixels in height and width.
                Resolution = px*px.
            parallel_x_blocks (int): number of cuts along the x axis in which to make␣
↪parallel.
            parallel_y_blocks (int): number of cuts along the y axis in which to make␣
↪parallel.
                parallel_x/y_blocks must be divisors of px.
                Number of threads = parallel_y_blocks * parallel_x_blocks.
        """
        self.ax = ax
        self.scene = scene
        self.px = px
        self.fl = image_dist # distance from eye to image
        self.i_hw = tan(image_angle/2)*self.fl # half width of the image
        self.px_hw = self.i_hw/px # half width of a pixel
        # set parallelization constants
        self.p_xb = parallel_x_blocks; self.p_yb = parallel_y_blocks
        bs_x = px/self.p_xb; bs_y = px/self.p_yb # block size for x,y
        if bs_x%1!=0 or bs_y%1!=0: raise Exception('parallel_x/y_blocks must divide px')
        self.bs_x = int(bs_x); self.bs_y = int(bs_y)
    def render(self):
        """ Render the image. """
        t0 = time() # start a timer
        img = Image.new('RGB',(self.px,self.px),(0,0,0))
        if self.p_xb==1 and self.p_yb==1:
            # use non-parallel processing (helpful for debugging)
            self.block_render(img,0,self.px,0,self.px)
        else:
            # parallel processing
            threads = [None]*(self.p_xb*self.p_yb)
            i_t = 0 # thread index
            for xb in range(0,self.px,self.bs_x):
                for yb in range(0,self.px,self.bs_y):
```

(continues on next page)

(continued from previous page)

```
                    threads[i_t] = Thread(target=self.block_render, args=(img,xb,xb+self.
↪bs_x,yb,yb+self.bs_y))
                    threads[i_t].start() # start threads
                    i_t += 1
            for i in range(len(threads)): threads[i].join() # wait for all threads to␣
↪complete
        self.ax.axis('off')
        self.ax.imshow(asarray(img))
        print('Render took %.1f seconds'%(time()-t0))
    def block_render(self, img, px_x_start, px_x_end, px_y_start, px_y_end):
        """
        Render a block of the image.

        Args:
            img (PIL.Image): the image to color pixels of.
            px_x_start (int): x index of pixel to start rendering at.
            px_x_end (int): x index of pixel to end rendering at.
            px_y_start (int): y index of pixel to start rendering at.
            px_y_end (int): y index of pixel to start rendering at.
        """
        for p_x in range(px_x_start,px_x_end):
            for p_y in range(px_y_start,px_y_end):
                p = array([-self.i_hw+2*self.i_hw*p_x/self.px,-self.i_hw+2*self.i_hw*p_y/
↪self.px,self.fl])
                color = self.scene.render_px(p)
                img.putpixel((p_x,self.px-p_y-1),color)
```

```
class Plane(object):
    def __init__(self, norm_axis, position, color):
        """
        Args:
            norm_axis (str): either 'x', 'y', or 'z'.
            position (str): constant position of plane along the norm_axis.
            color (tuple): length 3 tuple of rgb values.
        """
        dim_dict = {'x':0,'y':1,'z':2}
        self.d = dim_dict[norm_axis]
        self.pos = position # self.norm_axis coordinate of the floor
        self.color = color
    def hit(self, o, u):
        """
        Test if the beam o+tu hits the plane.

        Args:
            o (ndarray): length 3 origin point.
            u (ndarray): length 3 unit vector.

        Returns:
            tuple:
                - hit (bool): was an object hit?
                - hit_p (ndarray): point where beam intersects object.
                - color (tuple): length 3 tuple rgb value.
```

(continues on next page)

(continued from previous page)

```
        """
        k = u[self.d]
        if k != 0:
            t = (self.pos - o[self.d]) / u[self.d]
            if t > EPS:
                return True, o+t*u # ray intersects the plane
        return False, None # ray misses the plane
    def normal(self, o, u):
        """
        Get the unit normal vector to the plane at this point.

        Args:
            o (ndarray): length 3 origin point.
            u (ndarray): length 3 unit vector in direction of light.


        Returns:
            ndarray: length three unit normal vector.
        """
        v = array([0,0,0])
        v[self.d] = 1
        if dot(v,u)<0:
            v[self.d] = -1
        return v
```

```
class Ball(object):
    def __init__(self, center, radius, color):
        """
        Args:
            center (ndarray): length 3 center position of the ball.
            radius (float): radius of the ball.
            color (tuple): length 3 tuple of rgb values.
        """
        self.c = center
        self.r = radius
        self.color = color
    def hit(self, o, u):
        """
        Test if the beam o+tu hits the ball.

        Args:
            o (ndarray): length 3 origin point.
            u (ndarray): length 3 unit vector.

        Returns:
            tuple:
                - hit (bool): was an object was hit?
                - hit_p (ndarray): point where beam intersects object.
                - color (tuple): length 3 tuple rgb value.
        """
        q = o - self.c
        a = dot(u,u)
```

(continues on next page)

(continued from previous page)

```
        b = 2*dot(u,q)
        c = dot(q,q) - self.r**2
        d = b**2 - 4*a*c
        if d > 0: # ray intersects sphere
            tt = (-b + array([1,-1],dtype=float)*sqrt(d)) / (2**a)
            tt = tt[tt>EPS] # only want intersection from rays moving in positive␣
↪direction
            if len(tt) >= 1: # at least one positive intersection
                # beam going forward intersects ball
                t = min(tt)
                return True, o+t*u
        return False, None # ray does not intersect sphere or only intersects in␣
↪opposite direction
    def normal(self, o, u):
        """
        Get the unit normal vector to the sphere at thi point.

        Args:
            o (ndarray): length 3 origin point.
            u (ndarray): length 3 unit vector in direction of light.

        Returns:
            ndarray: length three unit normal vector.
        """
        v = (o-self.c)
        v_u = v/norm(v)
        return v_u
```

```
class PointLight(object):
    """ A lamp that is a point and emits. light in all directions. """
    def __init__(self, position, intensity):
        """
        Args:
            position (ndarray): length 3 coordinate of light position.
            intensity (float): intensity of the light, between 0 and 1.
        """
        self.p = position
        self.i = intensity
```

```
class CustomScene(object):
    def __init__(self, objs, light, n, d, mc_type):
        self.objs = objs
        self.light = light
        self.black = array([0,0,0],dtype=float) # the color black
        self.n = n
        self.d = d
        self.mc_type = mc_type
        # generate constant samples to be used for every ray tracing event
        if self.mc_type == 'IID':
            self.pts = IIDStdUniform(2*self.d).gen_samples(self.n)
        elif self.mc_type == 'SOBOL':
```

(continues on next page)

(continued from previous page)

```
            self.pts = Sobol(2*self.d,graycode=True).gen_samples(self.n)
        else:
            raise Exception("mc_type must be IID or Sobol")
    def find_closest_obj(self,o,v):
        """
        Find the closest object to point o heading in direction v

        Args:
            o (ndarray): length 3 coordinate of point we will try and find closest␣
↪object to

        Returns:
            tuple:
                hit (bool): weather any objects were hit.
                hit_p (ndarray): length 3 coordinates of where obj was hit.
                hit_dist (float): distance from hit_p to o.
                hit_obj (object): the object that was hit.
        """
        hit,hit_p,hit_dist,hit_obj = False,None,inf,None
        for obj in self.objs:
            obj_hit,obj_p = obj.hit(o,v)
            if obj_hit:
                v2 = obj_p-o # vector from o to object position
                obj_dist = sqrt(dot(v2,v2))
                if obj_dist < hit_dist:
                    hit,hit_p,hit_dist,hit_obj = True,obj_p,obj_dist,obj
        return hit,hit_p,hit_dist,hit_obj
    def get_obj_color(self,obj,p,l):
        """
        Get the objects color at point p with light in direction l.

        Args:
            obj (object): object on which p lies
            p (ndarray): length 3 coordinate of point on the object
            l (ndarray): length 3 vector of direction from p to light source

        Returns:
            ndarray: length 3 RGB color
        """
        n_v = obj.normal(p,l) # normal vector to obj at point p
        color = obj.color*self.light.i*dot(n_v,l) / (norm(n_v)*norm(l))
        return color
    def beam_r(self,o,v,n,d,pts,nidx,didx):
        """
        Recursive (Quasi-)Monte Carlo simulation of a light beam

        Args:
            o (ndarray): length 3 coordinate of current light beam position
            v (ndarray): length 3 vector of light beam direction
            n (ndarray): number of rays to cast when it cannot find light directly
            d (int): remaining bounces before beam gives up
            pts (ndarray): n samples x d dimension ndarray of samples generated by QMCPy
```

(continues on next page)

(continued from previous page)

```
            nidx (int): 2*(index of the beam)
            didx (int): index of the sample
        """
        hit,hit_p,hit_dist,hit_obj = self.find_closest_obj(o,v)
        if hit: # an object was hit
            l = self.light.p-hit_p # vector from position where beam hit to the lamp
            l_dist = norm(l) # distance from hit location to lamp
            l_u = l/l_dist # unit vector of l
            itw,itw_p,itw_dist,itw_obj = self.find_closest_obj(hit_p,l_u) # find any␣
↪object in the way
            if itw and itw_dist<= l_dist: # object between hit object and the lamp
                if d==0:
                    # no remaining bounces --> return black (give up)
                    return self.black
                else:
                    # beam has remaining bounces
                    color_total = self.black.copy()
                    for i in range(n):
                        theta_0 = 2*pi*pts[nidx+i,didx]
                        theta_1 = 2*pi*pts[nidx+i,didx+1]
                        x = sin(theta_0)*sin(theta_1)
                        y = sin(theta_0)*cos(theta_0)
                        z = sin(theta_1)
                        v_rand = array([x,y,z],dtype=float) # random direction
                        ho_n = hit_obj.normal(hit_p,l_u)
                        if dot(v_rand,ho_n) < 0: v_rand = -v_rand # flip direction to␣
↪correct hemisphere
                        obj_color = self.get_obj_color(hit_obj,hit_p,l_u)
                        color_total += obj_color*self.beam_r(hit_p,v_rand,n=1,d=d-1,
↪pts=pts,nidx=nidx+i,didx=didx+2)
                    return color_total/n # take the average of many simulations
            else: # nothin between the object and the light source
                # get the color based on point, normal to obj, and direction to light
                return self.get_obj_color(hit_obj,hit_p,l_u)
        return self.black # nothing hit --> return black '
    def render_px(self,p):
        """
        Get pixel value for ball-lamp-floor scene

        Args:
            p (ndarray): length 3 array coordinates of center of pixel to render
        """
        u = (p-e)/norm(p-e) # unit vector in direction of eye to pixel
        color_0_1 = self.beam_r(e,u,n=self.n,d=self.d,pts=self.pts,nidx=0,didx=0)
        color = (color_0_1*256).astype(int)
        return color[0],color[1],color[2]
```

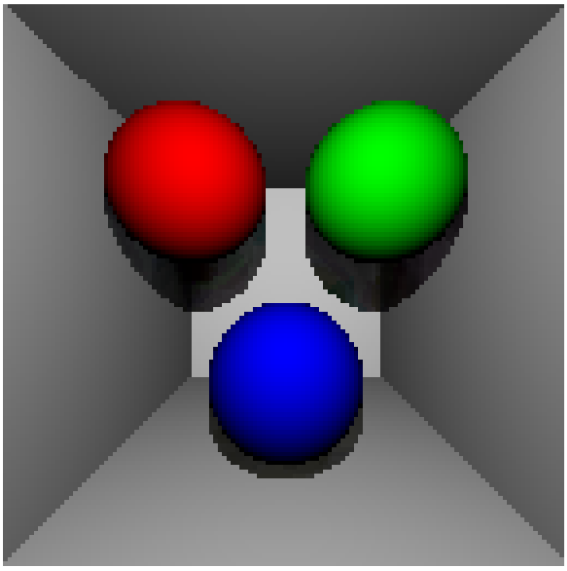
```
# create a scene
objs = [
    Plane(norm_axis='y', position=-50, color=array([.75,.75,.75],dtype=float)), # floor
    Plane(norm_axis='y', position=50,  color=array([.75,.75,.75],dtype=float)), # ceiling
    Plane(norm_axis='x', position=50,  color=array([.75,.75,.75],dtype=float)), # right␣
↪wall
```

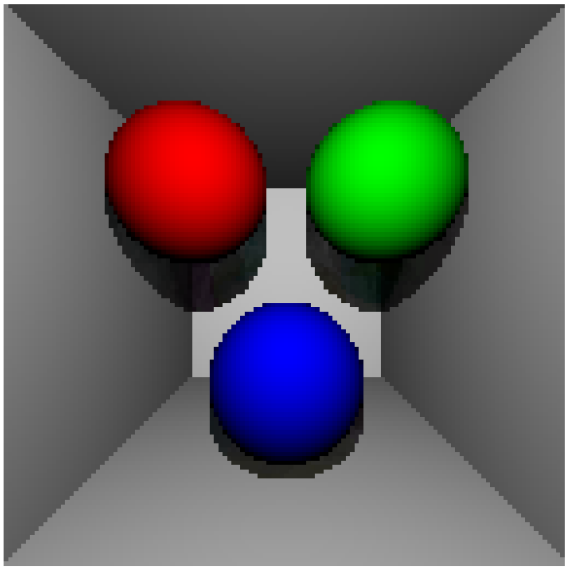
(continues on next page)

(continued from previous page)

```
    Plane(norm_axis='x', position=-50, color=array([.75,.75,.75],dtype=float)), # left␣
↪wall
    Plane(norm_axis='z', position=150,  color=array([.75,.75,.75],dtype=float)), # back␣
↪wall
    Ball(center=array([-25,25,75],dtype=float), radius=20, color=array([1,0,0],
↪dtype=float)), # ball
    Ball(center=array([25,25,75],dtype=float),  radius=20, color=array([0,1,0],
↪dtype=float)), # ball
    Ball(center=array([0,-25,75],dtype=float),  radius=20, color=array([0,0,1],
↪dtype=float)), # ball
]
light = PointLight(position=array([0,25,0],dtype=float), intensity=1)
# parameters
n = 16 # number of beams
d = 8 # max bounces of any given beam
px = 128
# render image
fig,ax = pyplot.subplots(ncols=2,nrows=1,figsize=(20,10))# render scene
# IID (MC)
scene = CustomScene(objs,light,n,d,mc_type='IID')
camera = Camera(ax[0], scene, px=px, parallel_x_blocks=1, parallel_y_blocks=1, image_
↪angle=pi/2, image_dist=1)
camera.render()
# Sobol (QMC)
scene = CustomScene(objs,light,n,d,mc_type='SOBOL')
camera = Camera(ax[1], scene, px=px, parallel_x_blocks=1, parallel_y_blocks=1, image_
↪angle=pi/2, image_dist=1)
camera.render()
```

```
Render took 4.6 seconds
Render took 5.1 seconds
```





5.12 A closer look at QMCPy's Sobol' generator

```
from qmcpy import *
from numpy import *
from matplotlib import pyplot
from time import time
import os
```

5.12.1 Basic usage

```
s = DigitalNetB2(5,seed=7)
s
```

```
DigitalNetB2 (DiscreteDistribution Object)
    d               5
    dvec            [0 1 2 3 4]
    randomize       LMS_DS
    graycode        0
    entropy         7
    spawn_key       ()
```

```
s.gen_samples(4) # generate Sobol' samples
```

```
array([[0.56269008, 0.17377997, 0.48903356, 0.72162356, 0.2540441 ],
       [0.346653  , 0.65070632, 0.98032014, 0.47528211, 0.89141566],
       [0.82074548, 0.95490574, 0.62413225, 0.21383165, 0.2009243 ],
       [0.10422261, 0.49458097, 0.09375278, 0.96749778, 0.5872364 ]])
```

```
s.gen_samples(n_min=2,n_max=4) # generate from specific range. If range is not powers of␣
↪2, use graycode
```

```
array([[0.82074548, 0.95490574, 0.62413225, 0.21383165, 0.2009243 ],
       [0.10422261, 0.49458097, 0.09375278, 0.96749778, 0.5872364 ]])
```

```
t0 = time()
s.gen_samples(2**25)
print('Time: %.2f'%(time()-t0))
```

```
Time: 1.77
```

5.12.2 Randomize with digital shift / linear matrix scramble

```
s = DigitalNetB2(2,randomize='LMS_DS') # linear matrix scramble with digital shift␣
↪(default)
s.gen_samples(2)
```

```
array([[0.19108946, 0.54321668],
       [0.79173853, 0.39566573]])
```

```
s = DigitalNetB2(2,randomize='LMS') # just linear matrix scrambling
s.gen_samples(2, warn=False) # suppress warning that the first point is still the orgin
```

```
array([[0.        , 0.        ],
       [0.81182601, 0.87542258]])
```

```
s = DigitalNetB2(2,randomize='DS') # just digital shift
s.gen_samples(2)
```

```
array([[0.49126963, 0.3658057 ],
       [0.99126963, 0.8658057 ]])
```

5.12.3 Support for graycode and natural ordering

```
s = DigitalNetB2(2,randomize=False,graycode=False)
s.gen_samples(n_min=4,n_max=8,warn=False) # don't warn about non-randomized samples␣
↪including the orgin
```

```
array([[0.125, 0.625],
       [0.625, 0.125],
       [0.375, 0.375],
       [0.875, 0.875]])
```

```
s = DigitalNetB2(2,randomize=False,graycode=True)
s.gen_samples(n_min=4,n_max=8,warn=False)
```

```
array([[0.375, 0.375],
       [0.875, 0.875],
       [0.625, 0.125],
       [0.125, 0.625]])
```

5.13 Custom Dimensions

```
s = DigitalNetB2(3,randomize=False)
s.gen_samples(n_min=4,n_max=8)
```

```
array([[0.125, 0.625, 0.375],
       [0.625, 0.125, 0.875],
       [0.375, 0.375, 0.625],
       [0.875, 0.875, 0.125]])
```

```
s = DigitalNetB2([1,2],randomize=False) # use only the second and third dimensions in␣
↪the sequence
s.gen_samples(n_min=4,n_max=8)
```

```
array([[0.625, 0.375],
       [0.125, 0.875],
       [0.375, 0.625],
       [0.875, 0.125]])
```

5.13.1 Custom generating matrices

```
# a previously created generating matrix (not the default)
z = load('../qmcpy/discrete_distribution/digital_net_b2/generating_matrices/sobol_mat.51.
↪30.30.msb.npy')
# max dimension 51, max samples 2^30, most significant bit in top of column
print(z.dtype)
z[:2,:2]
```

```
int64
```

```
array([[536870912, 805306368],
       [536870912, 268435456]])
```

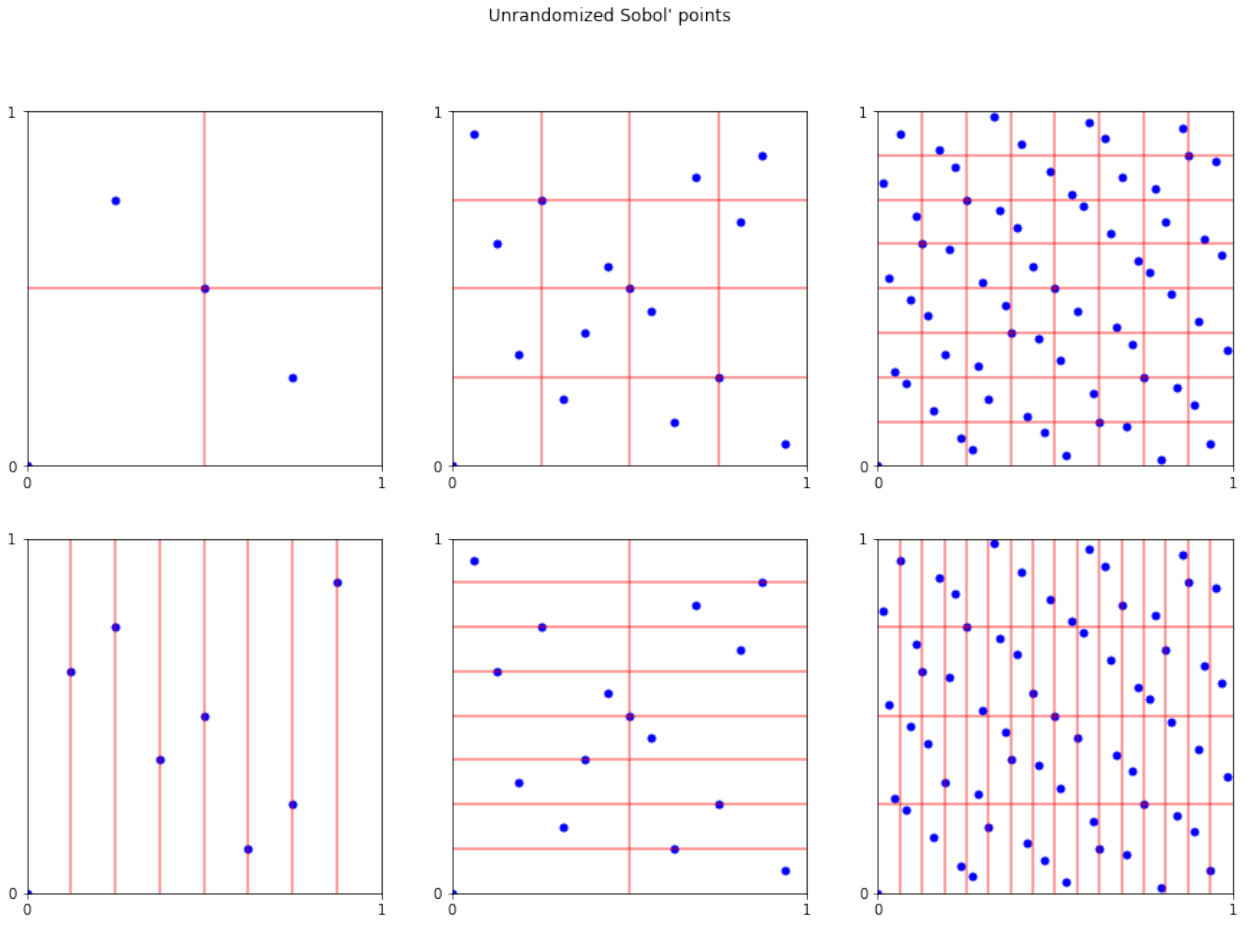
```
z_custom = z[:10,:] # say this is our custom generating matrix. Make sure the datatype␣
↪is numpy.int64
d_max,m_max = z_custom.shape
t_max = log2(z[0,0])+1 # number of bits in the first integer
f_path = 'my_sobol_mat.%d.%d.%d.msb.npy'%(d_max,t_max,m_max)
print(f_path)
save(f_path, z_custom)# save it to a file with proper naming convention
s = DigitalNetB2(3,generating_matrices=f_path) # plug in the path
print(s.gen_samples(4))
os.remove(f_path)
```

```
my_sobol_mat.10.30.30.msb.npy
[[0.51716073 0.27786139 0.31060685]
 [0.27369895 0.83466347 0.73047579]
 [0.24273408 0.09222157 0.77344039]
 [0.98569235 0.51987794 0.19732142]]
```

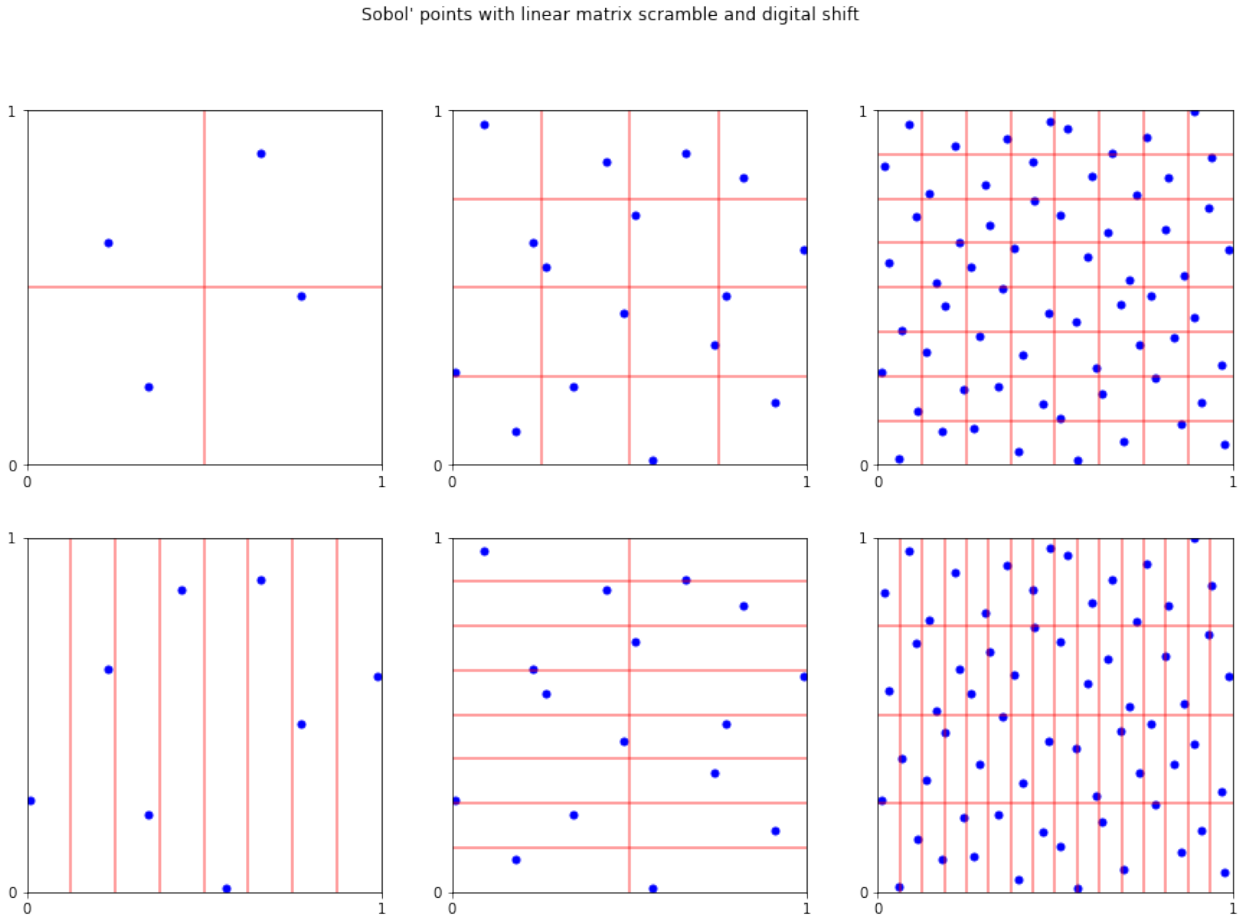
Elementay intervals

```
def plt_ei(x,ax,x_cuts,y_cuts):
    for ix in arange(1,x_cuts,dtype=float): ax.axvline(x=ix/x_cuts,color='r',alpha=.5)
    for iy in arange(1,y_cuts,dtype=float): ax.axhline(y=iy/y_cuts,color='r',alpha=.5)
    ax.scatter(x[:,0],x[:,1],color='b',s=25)
    ax.set_xlim([0,1])
    ax.set_xticks([0,1])
    ax.set_ylim([0,1])
    ax.set_yticks([0,1])
    ax.set_aspect(1)
```

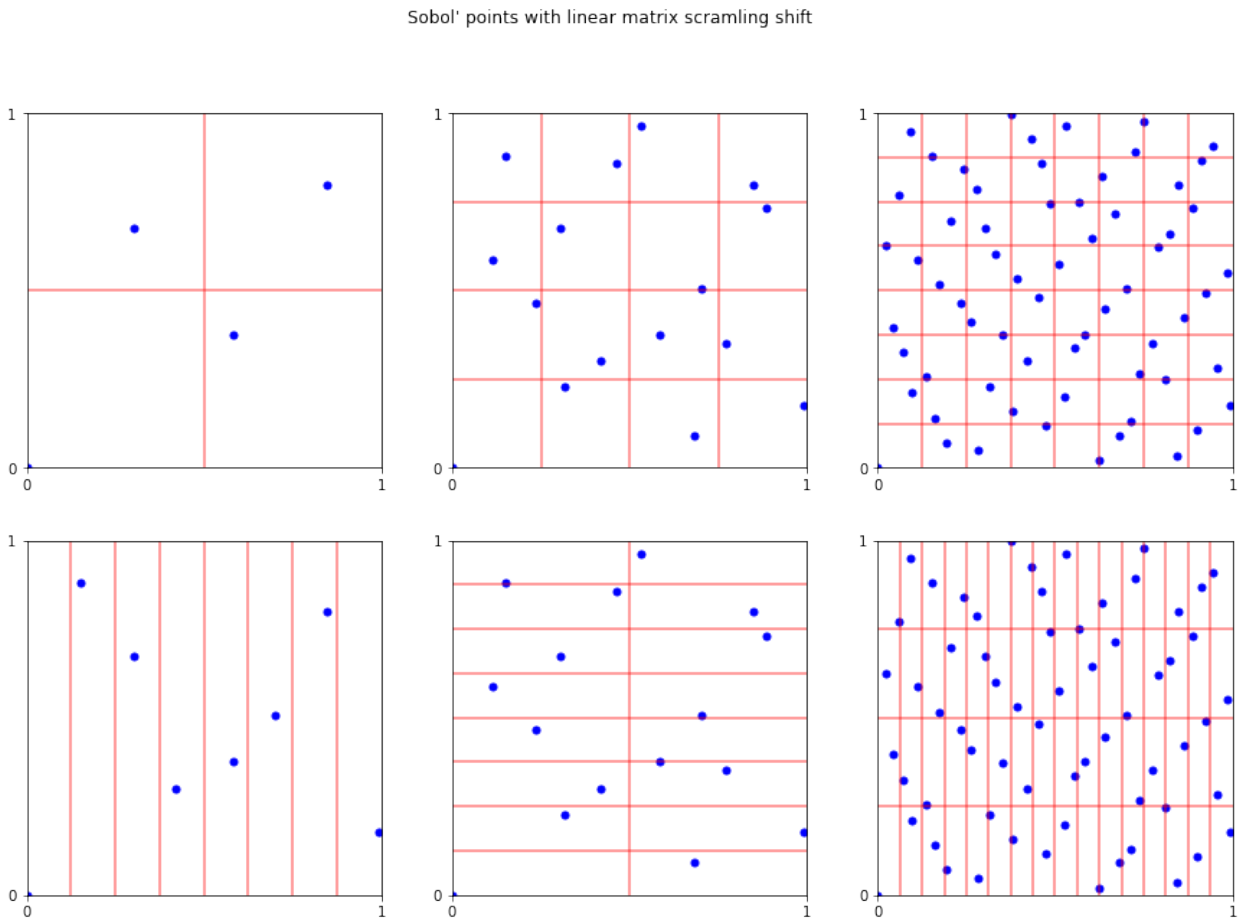
```
# unrandomized
fig,ax = pyplot.subplots(ncols=3,nrows=2,figsize=(15,10))
s = DigitalNetB2(2,randomize=False)
plt_ei(s.gen_samples(2**2,warn=False),ax[0,0],2,2)
plt_ei(s.gen_samples(2**3,warn=False),ax[1,0],8,0)
plt_ei(s.gen_samples(2**4,warn=False),ax[0,1],4,4)
plt_ei(s.gen_samples(2**4,warn=False),ax[1,1],2,8)
plt_ei(s.gen_samples(2**6,warn=False),ax[0,2],8,8)
plt_ei(s.gen_samples(2**6,warn=False),ax[1,2],16,4)
fig.suptitle("Unrandomized Sobol' points");
```



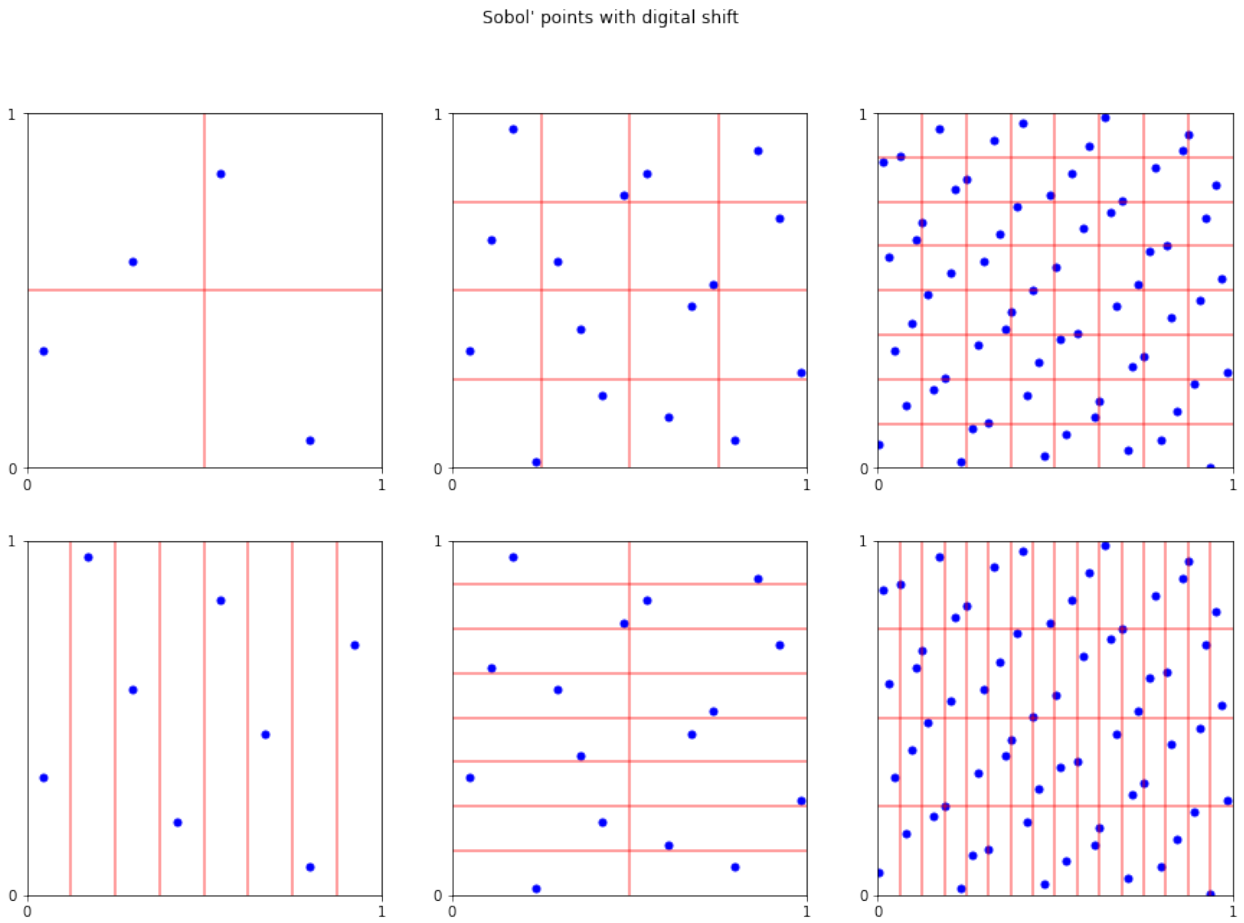
```
fig,ax = pyplot.subplots(ncols=3,nrows=2,figsize=(15,10))
s = DigitalNetB2(2,randomize='LMS_DS')
plt_ei(s.gen_samples(2**2,warn=False),ax[0,0],2,2)
plt_ei(s.gen_samples(2**3,warn=False),ax[1,0],8,0)
plt_ei(s.gen_samples(2**4,warn=False),ax[0,1],4,4)
plt_ei(s.gen_samples(2**4,warn=False),ax[1,1],2,8)
plt_ei(s.gen_samples(2**6,warn=False),ax[0,2],8,8)
plt_ei(s.gen_samples(2**6,warn=False),ax[1,2],16,4)
fig.suptitle("Sobol' points with linear matrix scramble and digital shift");
```



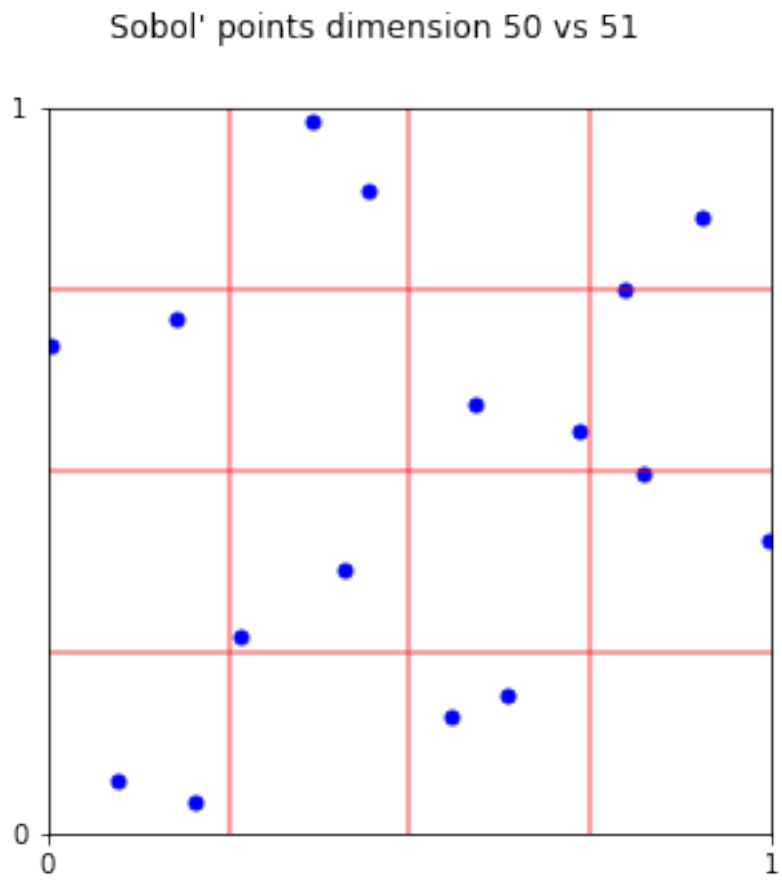

```
fig,ax = pyplot.subplots(ncols=3,nrows=2,figsize=(15,10))
s = DigitalNetB2(2,graycode=True,randomize='LMS')
plt_ei(s.gen_samples(2**2,warn=False),ax[0,0],2,2)
plt_ei(s.gen_samples(2**3,warn=False),ax[1,0],8,0)
plt_ei(s.gen_samples(2**4,warn=False),ax[0,1],4,4)
plt_ei(s.gen_samples(2**4,warn=False),ax[1,1],2,8)
plt_ei(s.gen_samples(2**6,warn=False),ax[0,2],8,8)
plt_ei(s.gen_samples(2**6,warn=False),ax[1,2],16,4)
fig.suptitle("Sobol' points with linear matrix scramling shift");
```



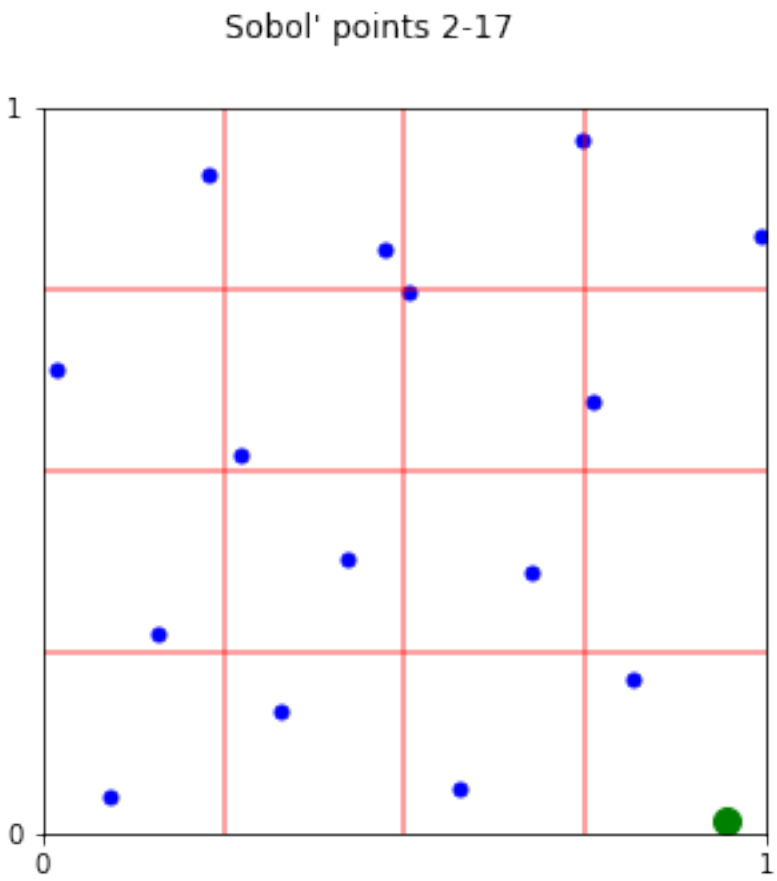

```
fig,ax = pyplot.subplots(ncols=3,nrows=2,figsize=(15,10))
s = DigitalNetB2(2,graycode=True,randomize='DS')
plt_ei(s.gen_samples(2**2,warn=False),ax[0,0],2,2)
plt_ei(s.gen_samples(2**3,warn=False),ax[1,0],8,0)
plt_ei(s.gen_samples(2**4,warn=False),ax[0,1],4,4)
plt_ei(s.gen_samples(2**4,warn=False),ax[1,1],2,8)
plt_ei(s.gen_samples(2**6,warn=False),ax[0,2],8,8)
plt_ei(s.gen_samples(2**6,warn=False),ax[1,2],16,4)
fig.suptitle("Sobol' points with digital shift");
```




```
fig,ax = pyplot.subplots(figsize=(15,5))
s = DigitalNetB2([50,51],randomize='LMS_DS')
plt_ei(s.gen_samples(2**4),ax,4,4)
fig.suptitle("Sobol' points dimension 50 vs 51");
# nice properties do not necessary hold in higher dimensions
```




```
fig,ax = pyplot.subplots(figsize=(15,5))
s = DigitalNetB2(2,randomize='LMS_DS',graycode=True)
plt_ei(s.gen_samples(n_min=1,n_max=16),ax,4,4)
x16 = s.gen_samples(n_min=16,n_max=17)
ax.scatter(x16[:,0],x16[:,1],color='g',s=100)
fig.suptitle("Sobol' points 2-17");
# bbetter to take points 1-16 instead of 2-16
```



5.13.2 Skipping points vs. randomization

The first point in a Sobol' sequence is $\vec{0}$. Therefore, some software packages skip this point as various transformations will map 0 to NAN. For example, the inverse CDF of a Gaussian density at $\vec{0}$ is $-\infty$. However, we argue that it is better to randomize points than to simply skip the first point for the following reasons:

1. Skipping the first point does not give the uniformity advantages when taking powers of 2. For example, notice the green point in the above plot of "Sobol' points 2-17".
2. Randomizing Sobol' points will return 0 with probability 0.

A more thorough explanation can be found in Art Owen's paper On dropping the first Sobol' point

So always randomize your Sobol' unless you specifically need unrandomized points. In QMCPy the Sobol' generator defaults to randomization with a linear matrix scramble.

The below code runs tests comparing unrandomized vs. randomization with linear matrix scramble with digital shift vs. randomization with just digital shift. Furthermore, we compare taking the first 2^n points vs. dropping the first point and taking 2^n points vs. dropping the first point and taking $2^n - 1$ points.

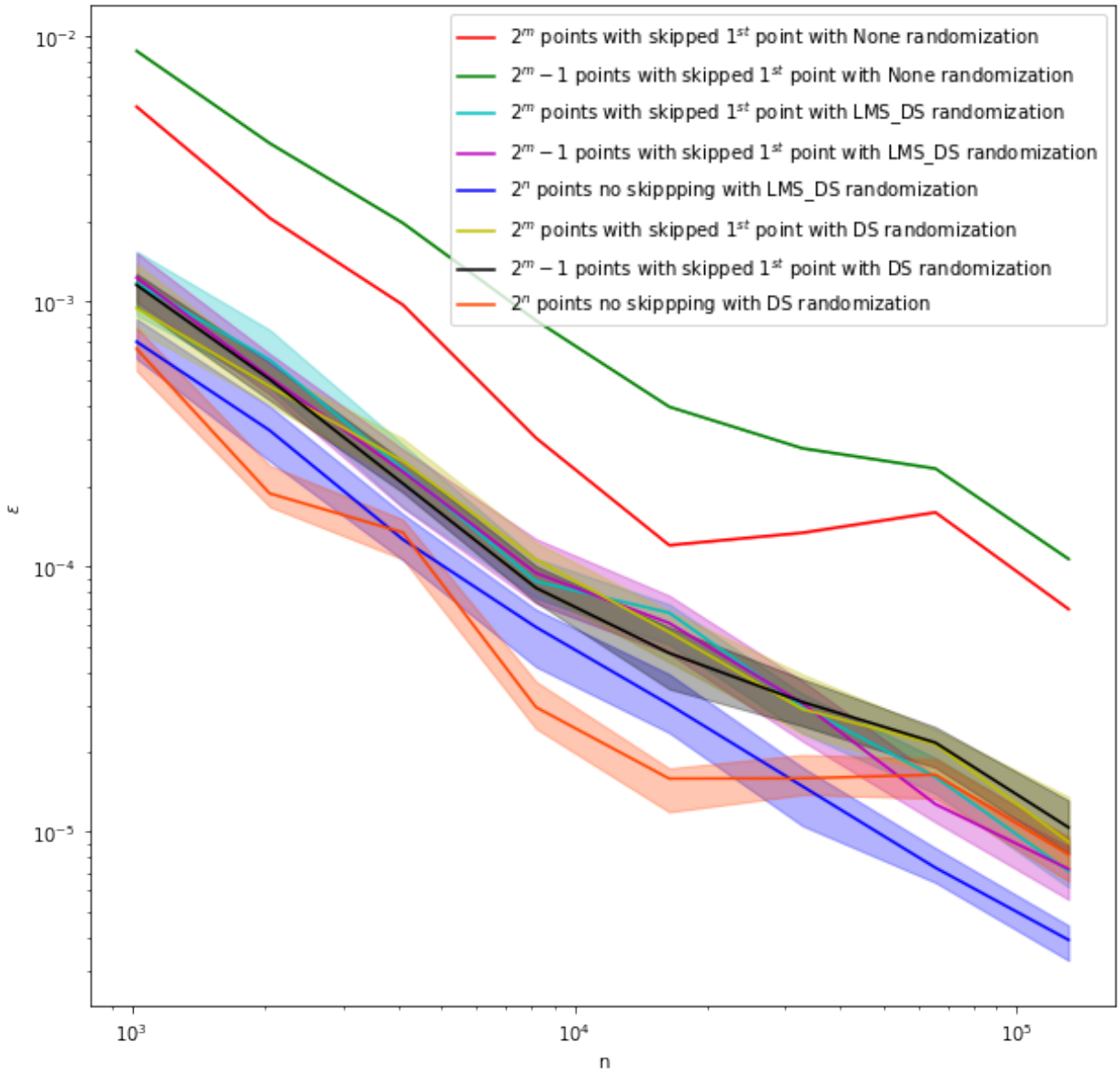
The 2D keister function is used for testing purposes as it can be exactly integrated using mathematica:

```
N[Integrate[E^(-x1^2 - x2^2) Cos[Sqrt[x1^2 + x2^2]], {x1, -Infinity, Infinity}, {x2,
-Infinity, Infinity}]]
```

Plots the median (line) and fills the middle 10% of observations

```
def plt_k2d_sobol(ax,rtype,colors,plts=[1,2,3]):
    trials = 100
    solution = 1.808186429263620
    ms = arange(10,18)
    ax.set_xscale('log',base=2)
    ax.set_yscale('log',base=10)
    epsilons = {}
    if 1 in plts:
        epsilons['$2^m$ points with skipped $1^{st}$ point'] = zeros((trials,len(ms)),
↪dtype=double)
    if 2 in plts:
        epsilons['$2^m-1$ points with skipped $1^{st}$ point'] = zeros((trials,len(ms)),
↪dtype=double)
    if 3 in plts:
        epsilons['$2^n$ points no skippping'] = zeros((trials,len(ms)),dtype=double)
    for i,m in enumerate(ms):
        for t in range(trials):
            s = DigitalNetB2(2,randomize=rtype,graycode=True)
            k = Keister(s)
            if 1 in plts:
                epsilons['$2^m$ points with skipped $1^{st}$ point'][t,i] = \
                    abs(k.f(s.gen_samples(n_min=1,n_max=1+2**m)).mean()-solution)
            if 2 in plts:
                epsilons['$2^m-1$ points with skipped $1^{st}$ point'][t,i] = \
                    abs(k.f(s.gen_samples(n_min=1,n_max=2**m)).mean()-solution)
            if 3 in plts:
                epsilons['$2^n$ points no skippping'][t,i] = \
                    abs(k.f(s.gen_samples(n=2**m)).mean()-solution)
    for i,(label,eps) in enumerate(epsilons.items()):
        bot = percentile(eps, 40, axis=0)
        med = percentile(eps, 50, axis=0)
        top = percentile(eps, 60, axis=0)
        ax.loglog(2**ms,med,label=label+' with %s randomization'%rtype,color=colors[i])
        ax.fill_between(2**ms, bot, top, color=colors[i], alpha=.3)
```

```
# compare randomization fig,ax = pyplot.subplots(figsize=(10,10))
fig,ax = pyplot.subplots(figsize=(10,10))
plt_k2d_sobol(ax,None,['r','g'],[1,2])
plt_k2d_sobol(ax,'LMS_DS',['c','m','b'],[1,2,3])
plt_k2d_sobol(ax,'DS',['y','k','orangered'],[1,2,3])
ax.legend()
ax.set_xlabel('n')
ax.set_ylabel('$\\epsilon$');
```



5.14 Some True Measures

In this notebook we explore some of the new, lesser-known, `TrueMeasure` instances in QMCPy. Specifically, we look at the Kumaraswamy, Continuous Bernoulli, and Johnson's S_U measures.

5.14.1 Mathematics

Denote by f the one-dimensional PDF, F the one-dimensional CDF, and $\Psi = F^{-1}$ the inverse CDF transform that takes samples mimicking $\mathcal{U}[0,1]$ to mimic the desired one-dimensional true measure. For each of these true measures we assume the dimensions are independent, so the density and CDF and computed by taking the product across dimensions and the inverse transform is applied elementwise.

Kumaraswamy

Parameters $a, b > 0$

$$f(x) = abx^{a-1}(1-x^a)^{b-1}$$
$$F(x) = 1-(1-x^a)^b$$
$$\Psi(x) = (1-(1-x)^{1/b})^{1/a}$$
$$\Psi'(x) = \frac{\left(1-(1-x)^{1/b}\right)^{1/a-1}(1-x)^{1/b-1}}{ab}$$

Continuous Bernoulli

Parameter $\lambda \in (0,1)$

If $\lambda = 1/2$, then

$$f(x) = 2\lambda^x(1-\lambda)^{(1-x)}$$
$$F(x) = x$$
$$\Psi(x) = x$$
$$\Psi'(x) = 1$$

If $\lambda \neq 1/2$, then

$$f(x) = \frac{2\tanh^{-1}(1-2\lambda)}{1-2\lambda}\lambda^x(1-\lambda)^{(1-x)}$$
$$F(x) = \frac{\lambda^x(1-\lambda)^{(1-x)}+\lambda-1}{2\lambda-1}$$
$$\Psi(x) = \log\left(\frac{(2\lambda-1)x-\lambda+1}{1-\lambda}\right) \,/\, \log\left(\frac{\lambda}{1-\lambda}\right)$$
$$\Psi'(x) = \frac{1}{\log(\lambda/(1-\lambda))}\cdot\frac{2\lambda-1}{(2\lambda-1)x-\lambda+1}$$

Johnson's :math: `S_U <https://en.wikipedia.org/wiki/Johnson%27s_SU-distribution>`__

Parameters $\gamma, \xi, \delta > 0, \lambda > 0$

$$f(x) = \frac{\delta\exp\left(-\frac{1}{2}\left(\gamma+\delta\sinh^{-1}\left(\frac{x-\xi}{\lambda}\right)\right)^2\right)}{\lambda\sqrt{2\pi\left(1+\left(\frac{x-\xi}{\lambda}\right)^2\right)}}$$
$$F(x) = \Phi\left(\gamma+\delta\sinh^{-1}\left(\frac{x-\xi}{\lambda}\right)\right)$$
$$\Psi(x) = \lambda\sinh\left(\frac{\Phi^{-1}(x)-\gamma}{\delta}\right)+\xi$$
$$\Psi'(x) = \frac{\lambda}{\delta}\cosh\left(\frac{\Phi^{-1}(x)-\gamma}{\delta}\right) \,/\, \phi\left(\Phi^{-1}(x)\right)$$

where ϕ is the standard normal PDF and Φ is the standard normal CDF.

5.14.2 Imports

```
from qmcpy import *
from scipy.special import gamma
from numpy import *
```

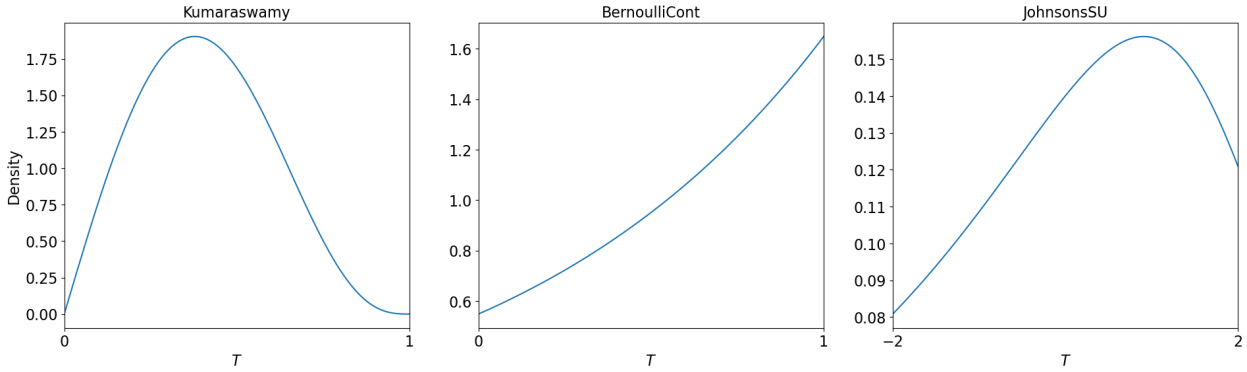
```
import matplotlib
from matplotlib import pyplot
%matplotlib inline
pyplot.rc('font', size=16)          # controls default text sizes
pyplot.rc('axes', titlesize=16)     # fontsize of the axes title
pyplot.rc('axes', labelsize=16)     # fontsize of the x and y labels
pyplot.rc('xtick', labelsize=16)    # fontsize of the tick labels
pyplot.rc('ytick', labelsize=16)    # fontsize of the tick labels
pyplot.rc('legend', fontsize=16)    # legend fontsize
pyplot.rc('figure', titlesize=16)   # fontsize of the figure title
```

5.14.3 1D Density Plot

```
def plt_1d(tm,lim,ax):
    print(tm)
    n_mesh = 100
    ticks = linspace(*lim,n_mesh)
    y = tm._weight(ticks[:,None])
    ax.plot(ticks,y)
    ax.set_xlim(lim)
    ax.set_xlabel("$T$")
    ax.set_xticks(lim)
    ax.set_title(type(tm).__name__)
```

```
fig,ax = pyplot.subplots(figsize=(23,6),nrows=1,ncols=3)
sobol = Sobol(1)
kumaraswamy = Kumaraswamy(sobol,a=2,b=4)
plt_1d(kumaraswamy,lim=[0,1],ax=ax[0])
bern = BernoulliCont(sobol,lam=.75)
plt_1d(bern,lim=[0,1],ax=ax[1])
jsu = JohnsonsSU(sobol,gamma=1,xi=2,delta=1,lam=2)
plt_1d(jsu,lim=[-2,2],ax=ax[2])
ax[0].set_ylabel("Density");
```

```
Kumaraswamy (TrueMeasure Object)
    a               2^(1)
    b               2^(2)
BernoulliCont (TrueMeasure Object)
    lam             0.750
JohnsonsSU (TrueMeasure Object)
    gamma           1
    xi              2^(1)
    delta           1
    lam             2^(1)
```

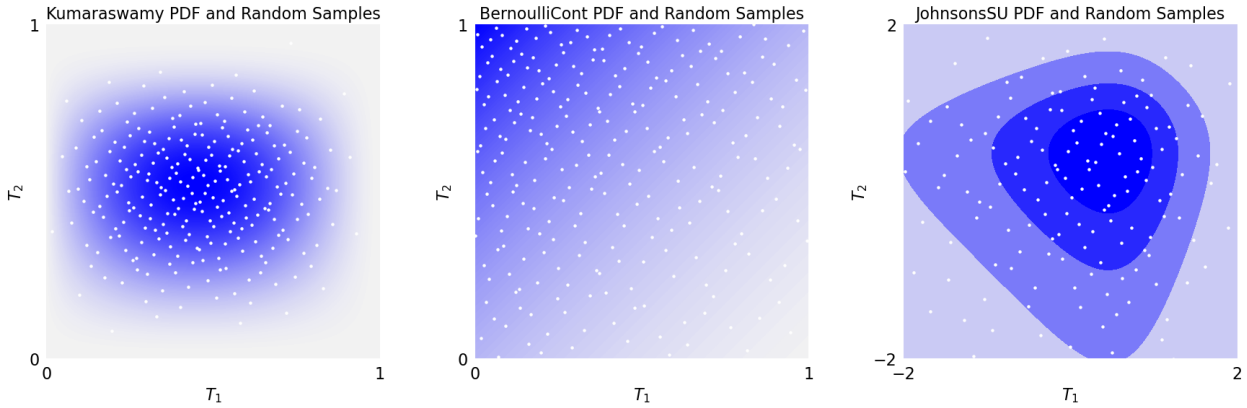


5.14.4 2D Density Plot

```
def plt_2d(tm,n,lim,ax):
    print(tm)
    n_mesh = 502
    # Points
    t = tm.gen_samples(n)
    # PDF
    mesh = zeros(((n_mesh)**2,3),dtype=float)
    grid_tics = linspace(*lim,n_mesh)
    x_mesh,y_mesh = meshgrid(grid_tics,grid_tics)
    mesh[:,0] = x_mesh.flatten()
    mesh[:,1] = y_mesh.flatten()
    mesh[:,2] = tm._weight(mesh[:,:2])
    z_mesh = mesh[:,2].reshape((n_mesh,n_mesh))
    #   colors
    clevel = arange(mesh[:,2].min(),mesh[:,2].max(),.025)
    cmap = matplotlib.colors.LinearSegmentedColormap.from_list("", [(.95,.95,.95),(0,0,
↪1)])
    # cmap = pyplot.get_cmap('Blues')
    #   contour + scatter plot
    ax.contourf(x_mesh,y_mesh,z_mesh,clevel,cmap=cmap,extend='both')
    ax.scatter(t[:,0],t[:,1],s=5,color='w')
    #   axis
    for nsew in ['top','bottom','left','right']: ax.spines[nsew].set_visible(False)
    ax.xaxis.set_ticks_position('none')
    ax.yaxis.set_ticks_position('none')
    ax.set_aspect(1)
    ax.set_xlim(lim)
    ax.set_xticks(lim)
    ax.set_ylim(lim)
    ax.set_yticks(lim)
    #   labels
    ax.set_xlabel('$T_1$')
    ax.set_ylabel('$T_2$')
    ax.set_title('%s PDF and Random Samples'%type(tm).__name__)
    #   metas
    fig.tight_layout()
```

```
fig,ax = pyplot.subplots(figsize=(18,6),nrows=1,ncols=3)
sobol = Sobol(2)
kumaraswamy = Kumaraswamy(sobol,a=[2,3],b=[3,5])
plt_2d(kumaraswamy,n=2**8,lim=[0,1],ax=ax[0])
bern = BernoulliCont(sobol,lam=[.25,.75])
plt_2d(bern,n=2**8,lim=[0,1],ax=ax[1])
jsu = JohnsonsSU(sobol,gamma=[1,1],xi=[1,1],delta=[1,1],lam=[1,1])
plt_2d(jsu,n=2**8,lim=[-2,2],ax=ax[2])
```

```
Kumaraswamy (TrueMeasure Object)
    a               [2 3]
    b               [3 5]
BernoulliCont (TrueMeasure Object)
    lam             [0.25 0.75]
JohnsonsSU (TrueMeasure Object)
    gamma           [1 1]
    xi              [1 1]
    delta           [1 1]
    lam             [1 1]
```



5.14.5 1D Expected Values

```
def compute_expected_val(tm,true_value,abs_tol):
    if tm.d!=1: raise Exception("tm must be 1 dimensional for this test")
    cf = CustomFun(tm, g=lambda x:x)
    sol,data = CubQMCSobolG(cf,abs_tol=abs_tol).integrate()
    error = abs(true_value-sol)
    if error>abs_tol:
        raise Exception("QMC error %.3f not within tolerance."%error)
    print("%s integration within tolerance"%type(tm).__name__)
    print("\tEstimated mean: %.4f"%sol)
    print("\t%s"%str(tm).replace('\n','\n\t'))
```

```
abs_tol = 1e-5
# kumaraswamy
a,b = 2,6
kuma = Kumaraswamy(Sobol(1),a,b)
```

(continues on next page)

(continued from previous page)

```
kuma_tv = b*gamma(1+1/a)*gamma(b)/gamma(1+1/a+b)
compute_expected_val(kuma,kuma_tv,abs_tol)
# Continuous Bernoulli
lam = .75
bern = BernoulliCont(Sobol(1),lam=lam)
bern_tv = 1/2 if lam==1/2 else lam/(2*lam-1)+1/(2*arctanh(1-2*lam))
compute_expected_val(bern,bern_tv,abs_tol)
# Johnson's SU
_gamma,xi,delta,lam = 1,2,3,4
jsu = JohnsonsSU(Sobol(1),gamma=_gamma,xi=xi,delta=delta,lam=lam)
jsu_tv = xi-lam*exp(1/(2*delta**2))*sinh(_gamma/delta)
compute_expected_val(jsu,jsu_tv,abs_tol)
```

```
Kumaraswamy integration within tolerance
    Estimated mean: 0.3410
    Kumaraswamy (TrueMeasure Object)
        a               2^(1)
        b               6
BernoulliCont integration within tolerance
    Estimated mean: 0.5898
    BernoulliCont (TrueMeasure Object)
        lam             0.750
JohnsonsSU integration within tolerance
    Estimated mean: 0.5642
    JohnsonsSU (TrueMeasure Object)
        gamma           1
        xi              2^(1)
        delta           3
        lam             2^(2)
```

5.14.6 Importance Sampling with a Single Kumaraswamy

```
# compose with a Kumaraswamy transformation
abs_tol = 1e-4
cf = CustomFun(Uniform(Kumaraswamy(Sobol(1,seed=7),a=.5,b=.5)), g=lambda x:x)
sol,data = CubQMCSobolG(cf,abs_tol=abs_tol).integrate()
print(data)
true_val = .5 # expected value of standard uniform
error = abs(true_val-sol)
if error>abs_tol: raise Exception("QMC error %.3f not within tolerance."%error)
```

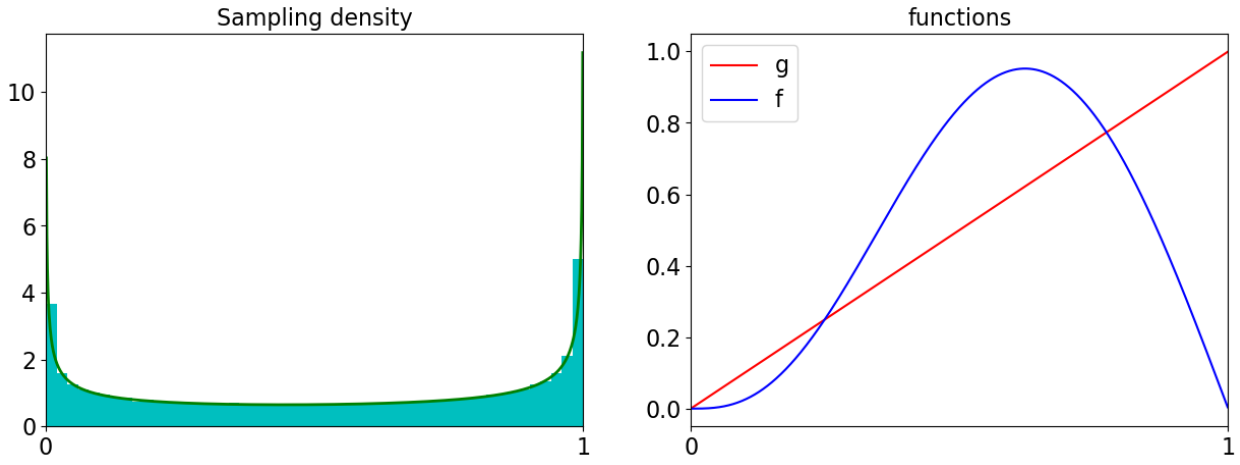
```
LDTransformData (AccumulateData Object)
    solution        0.500
    comb_bound_low  0.500
    comb_bound_high 0.500
    comb_flags      1
    n_total         2^(11)
    n               2^(11)
    time_integrate  0.012
CubQMCSobolG (StoppingCriterion Object)
```

(continues on next page)

(continued from previous page)

```
    abs_tol         1.00e-04
    rel_tol         0
    n_init          2^(10)
    n_max           2^(35)
CustomFun (Integrand Object)
Uniform (TrueMeasure Object)
    lower_bound     0
    upper_bound     1
    transform       Kumaraswamy (TrueMeasure Object)
                       a               2^(-1)
                       b               2^(-1)
Sobol (DiscreteDistribution Object)
    d               1
    dvec            0
    randomize       LMS_DS
    graycode        0
    entropy         7
    spawn_key       ()
```

```
# plot the above functions
x = linspace(0,1,1000).reshape(-1,1)
x = x[1:-1] # plotting locations
# plot
fig,ax = pyplot.subplots(ncols=2,figsize=(15,5))
#    density
rho = cf.true_measure.transform._weight(x)
tfvals = cf.true_measure.transform._transform_r(x)
ax[0].hist(tfvals,density=True,bins=50,color='c')
ax[0].plot(x,rho,color='g',linewidth=2)
ax[0].set_title('Sampling density')
#    functions
gs = cf.g(x)
fs = cf.f(x)
ax[1].plot(x,gs,color='r',label='g')
ax[1].plot(x,fs,color='b',label='f')
ax[1].legend()
ax[1].set_title('functions');
# metas
for i in range(2):
    ax[i].set_xlim([0,1])
    ax[i].set_xticks([0,1])
```



5.14.7 Importance Sampling with 2 (Composed) Kumaraswamys

```
# compose multiple Kumaraswamy distributions
abs_tol = 1e-3
dd = Sobol(1,seed=7)
t1 = Kumaraswamy(dd,a=.5,b=.5) # first transformation
t2 = Kumaraswamy(t1,a=5,b=2) # second transformation
cf = CustomFun(Uniform(t2), g=lambda x:x)
sol,data = CubQMCSobolG(cf,abs_tol=abs_tol).integrate() # expected value of standard␣
↪uniform
print(data)
true_val = .5
error = abs(true_val-sol)
if error>abs_tol: raise Exception("QMC error %.3f not within tolerance."%error)
```

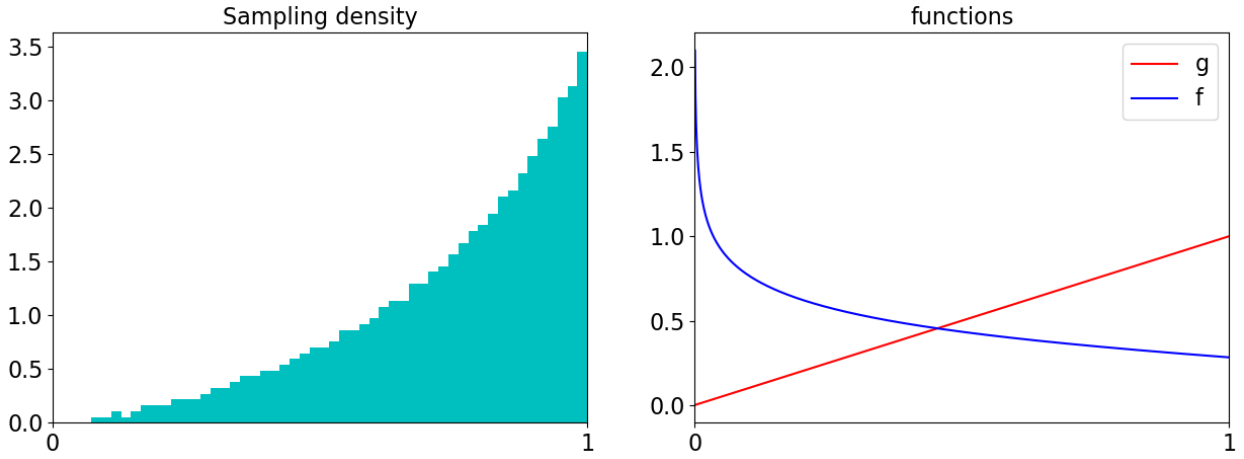
```
LDTransformData (AccumulateData Object)
    solution        0.500
    comb_bound_low  0.499
    comb_bound_high 0.501
    comb_flags      1
    n_total         2^(10)
    n               2^(10)
    time_integrate  0.007
CubQMCSobolG (StoppingCriterion Object)
    abs_tol         0.001
    rel_tol         0
    n_init          2^(10)
    n_max           2^(35)
CustomFun (Integrand Object)
Uniform (TrueMeasure Object)
    lower_bound     0
    upper_bound     1
    transform       Kumaraswamy (TrueMeasure Object)
                       a               5
                       b               2^(1)
                       transform       Kumaraswamy (TrueMeasure Object)
```

(continues on next page)

(continued from previous page)

```
                                        a               2^(-1)
                                        b               2^(-1)
Sobol (DiscreteDistribution Object)
    d               1
    dvec            0
    randomize       LMS_DS
    graycode        0
    entropy         7
    spawn_key       ()
```

```
x = linspace(0,1,1000).reshape(-1,1)
x = x[1:-1] # plotting locations
# plot
fig,ax = pyplot.subplots(ncols=2,figsize=(15,5))
#    density
tfvals = cf.true_measure.transform._transform_r(x)
ax[0].hist(tfvals,density=True,bins=50,color='c')
ax[0].set_title('Sampling density')
#    functions
gs = cf.g(x)
fs = cf.f(x)
ax[1].plot(x,gs,color='r',label='g')
ax[1].plot(x,fs,color='b',label='f')
ax[1].legend()
for i in range(2):
    ax[i].set_xlim([0,1])
    ax[i].set_xticks([0,1])
ax[1].set_title('functions');
```



5.14.8 Can we Improve the Keister function?

```
abs_tol = 1e-4
d = 1
```

```
# standard method
keister_std = Keister(Sobol(d))
sol_std,data_std = CubQMCSobolG(keister_std,abs_tol=abs_tol).integrate()
print("Standard method estimate of %.4f took %.2e seconds and %.2e samples"%\
      (sol_std,data_std.time_integrate,data_std.n_total))
print(keister_std.true_measure)
```

```
Standard method estimate of 1.3804 took 1.20e-02 seconds and 1.64e+04 samples
Gaussian (TrueMeasure Object)
    mean            0
    covariance      2^(-1)
    decomp_type     PCA
```

```
# Kumaraswamy importance sampling
dd = Sobol(d,seed=7)
t1 = Kumaraswamy(dd,a=.8,b=.8) # first transformation
t2 = Gaussian(t1)
keister_kuma = Keister(t2)
sol_kuma,data_kuma = CubQMCSobolG(keister_kuma,abs_tol=abs_tol).integrate()
print("Kumaraswamy IS estimate of %.4f took %.2e seconds and %.2e samples"%\
      (sol_kuma,data_kuma.time_integrate,data_kuma.n_total))
t_frac = data_kuma.time_integrate/data_std.time_integrate
n_frac = data_kuma.n_total/data_std.n_total
print('That is %.1f%% of the time and %.1f%% of the samples compared to default keister.
↪'%(t_frac*100,n_frac*100))
print(keister_kuma.true_measure)
```

```
Kumaraswamy IS estimate of 1.3804 took 1.21e-02 seconds and 2.05e+03 samples
That is 100.9% of the time and 12.5% of the samples compared to default keister.
Gaussian (TrueMeasure Object)
    mean            0
    covariance      2^(-1)
    decomp_type     PCA
    transform       Gaussian (TrueMeasure Object)
                       mean            0
                       covariance      1
                       decomp_type     PCA
                       transform       Kumaraswamy (TrueMeasure Object)
                                          a               0.800
                                          b               0.800
```

```
# Continuous Bernoulli importance sampling
dd = Sobol(d,seed=7)
t1 = BernoulliCont(dd,lam=.25) # first transformation
#t2 = BernoulliCont(t1,lam=.75) # first transformation
t3 = Gaussian(t1)
keister_cb = Keister(t3)
```

(continues on next page)

(continued from previous page)

```
sol_cb,data_cb = CubQMCSobolG(keister_cb,abs_tol=abs_tol).integrate()
print("Continuous Bernoulli IS estimate of %.4f took %.2e seconds and %.2e samples"%\
      (sol_cb,data_cb.time_integrate,data_cb.n_total))
t_frac = data_cb.time_integrate/data_std.time_integrate
n_frac = data_cb.n_total/data_std.n_total
print('That is %.1f%% of the time and %.1f%% of the samples compared to default keister.
↪'%(t_frac*100,n_frac*100))
print(keister_cb.true_measure)
```

```
Continuous Bernoulli IS estimate of 1.3804 took 4.15e-02 seconds and 8.19e+03 samples
That is 346.8% of the time and 50.0% of the samples compared to default keister.
Gaussian (TrueMeasure Object)
    mean            0
    covariance      2^(-1)
    decomp_type     PCA
    transform       Gaussian (TrueMeasure Object)
                       mean            0
                       covariance      1
                       decomp_type     PCA
                       transform       BernoulliCont (TrueMeasure Object)
                                          lam             2^(-2)
```

```
#  Kumaraswamy importance sampling
dd = Sobol(d,seed=7)
t1 = JohnsonsSU(dd,xi=2,delta=1,gamma=2,lam=1) # first transformation
keister_jsu = Keister(t1)
sol_jsu,data_jsu = CubQMCSobolG(keister_jsu,abs_tol=abs_tol).integrate()
print("Kumaraswamy IS estimate of %.4f took %.2e seconds and %.2e samples"%\
      (sol_jsu,data_jsu.time_integrate,data_jsu.n_total))
t_frac = data_jsu.time_integrate/data_std.time_integrate
n_frac = data_jsu.n_total/data_std.n_total
print('That is %.1f%% of the time and %.1f%% of the samples compared to default keister.
↪'%(t_frac*100,n_frac*100))
print(keister_jsu.true_measure)
```

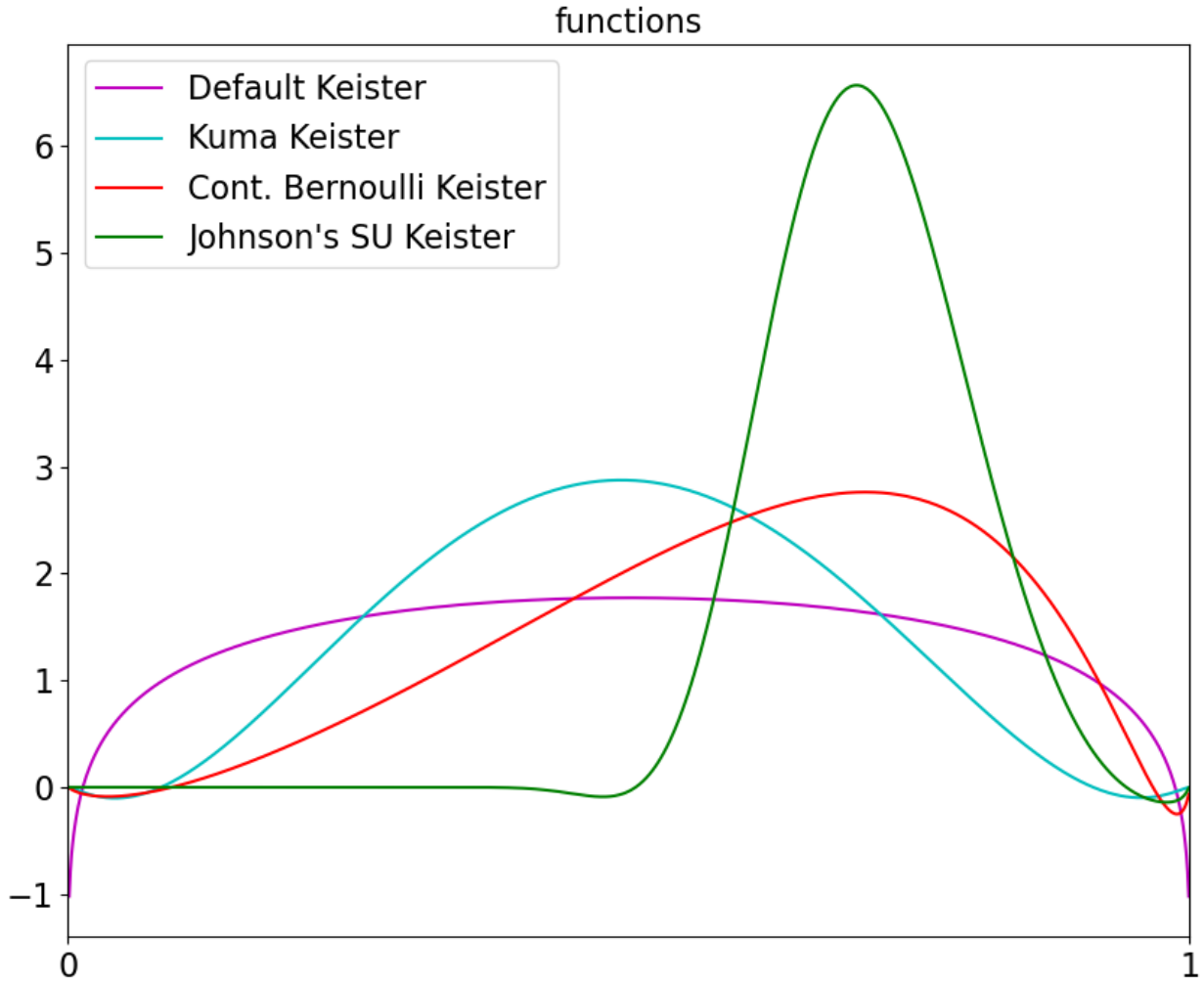
```
Kumaraswamy IS estimate of 1.3804 took 4.08e-02 seconds and 8.19e+03 samples
That is 340.9% of the time and 50.0% of the samples compared to default keister.
Gaussian (TrueMeasure Object)
    mean            0
    covariance      2^(-1)
    decomp_type     PCA
    transform       JohnsonsSU (TrueMeasure Object)
                       gamma           2^(1)
                       xi              2^(1)
                       delta           1
                       lam             1
```

```
x = linspace(0,1,1000).reshape(-1,1)
x = x[1:-1] # plotting locations
# plot
fig,ax = pyplot.subplots(figsize=(10,8))
```

(continues on next page)

(continued from previous page)

```
#    functions
fs = [keister_std.f,keister_kuma.f,keister_cb.f,keister_jsu.f]
labels = ['Default Keister','Kuma Keister','Cont. Bernoulli Keister',"Johnson's SU␣
↪Keister"]
colors = ['m','c','r','g']
for f,label,color in zip(fs,labels,colors): ax.plot(x,f(x),color=color,label=label)
ax.legend()
ax.set_xlim([0,1])
ax.set_xticks([0,1])
ax.set_title('functions');
```




```
import matplotlib.pyplot as plt
import numpy as np
import qmcpy as qp
```

```
seed = 7
```

5.15 Comparison of multilevel (Quasi-)Monte Carlo for an Asian option problem

Compute the exact value of the Asian option with single level QMC, for an increasing number of time steps:

```
for level in range(5):
    aco = qp.AsianOption(qp.Sobol(2*2**level, seed=seed), volatility=.2, start_price=100,
↪ strike_price=100, interest_rate=.05)
    approx_solution, data = qp.CubQMCSobolG(aco, abs_tol=1e-4).integrate()
    print("Asian Option true value (%d time steps): %.5f (to within 1e-4)"%(2*2**level,␣
↪approx_solution))
```

```
Asian Option true value (2 time steps): 5.63591 (to within 1e-4)
Asian Option true value (4 time steps): 5.73171 (to within 1e-4)
Asian Option true value (8 time steps): 5.75526 (to within 1e-4)
Asian Option true value (16 time steps): 5.76113 (to within 1e-4)
Asian Option true value (32 time steps): 5.76260 (to within 1e-4)
```

This function compares 4 different algorithms: Multilevel Monte Carlo (`CubMCML`), Multilevel Quasi-Monte Carlo (`CubQMCML`), continuation Multilevel Monte Carlo (`CubMCMLCont`) and Multilevel Quasi-Monte Carlo (`CubQMCMLCont`):

```
def eval_option(option_mc, option_qmc, abs_tol):
    stopping_criteria = {
        "MLMC" : qp.CubMCML(option_mc, abs_tol=abs_tol, levels_max=15),
        "continuation MLMC" : qp.CubMCMLCont(option_mc, abs_tol=abs_tol, levels_max=15),
        "MLQMC" : qp.CubQMCML(option_qmc, abs_tol=abs_tol, levels_max=15),
        "continuation MLQMC" : qp.CubQMCMLCont(option_qmc, abs_tol=abs_tol, levels_
↪max=15)
    }

    levels = []
    times = []
    for name, stopper in stopping_criteria.items():
        sol, data = stopper.integrate()
        levels.append(data.levels)
        times.append(data.time_integrate)
        print("\t%-20s solution %-10.4f number of levels %-6d time %.3f"%(name, sol,␣
↪levels[-1], times[-1]))

    return levels, times
```

Define the Multilevel Asian options:

```
option_mc = qp.MLCallOptions(qp.IIDStdUniform(seed=seed), option="asian")
option_qmc = qp.MLCallOptions(qp.Lattice(seed=seed), option="asian")
```

Run and compare each of the 4 algorithms for the Asian option problem:

```
eval_option(option_mc, option_qmc, abs_tol=5e-3);
```

```
MLMC                 solution 5.7620     number of levels 10     time 12.119
continuation MLMC    solution 5.7580     number of levels 7      time 7.775
MLQMC                solution 5.7606     number of levels 8      time 55.780
continuation MLQMC   solution 5.7594     number of levels 7      time 18.445
```

Repeat this comparison for a sequence of decreasing tolerances, with 5 different random seeds each. This will allow us to visualize the asymptotic cost complexity of each method.

```
repetitions = 5
tolerances = 5*np.logspace(-1, -3, num=5)

levels = {}
times = {}
for t in range(len(tolerances)):
    for r in range(repetitions):
        print("tolerance = %10.4e, repetition = %d/%d"%(tolerances[t], r + 1,␣
↪repetitions))
        levels[t, r], times[t, r] = eval_option(option_mc, option_qmc, tolerances[t])
```

```
tolerance = 5.0000e-01, repetition = 1/5
    MLMC                 solution 5.5049     number of levels 3      time 0.006
    continuation MLMC    solution 5.6865     number of levels 3      time 0.008
    MLQMC                solution 5.7204     number of levels 3      time 0.144
    continuation MLQMC   solution 5.7099     number of levels 3      time 0.000
tolerance = 5.0000e-01, repetition = 2/5
    MLMC                 solution 5.7316     number of levels 4      time 0.005
    continuation MLMC    solution 5.6755     number of levels 4      time 0.009
    MLQMC                solution 5.7196     number of levels 3      time 0.139
    continuation MLQMC   solution 5.7014     number of levels 3      time 0.000
tolerance = 5.0000e-01, repetition = 3/5
    MLMC                 solution 5.6791     number of levels 3      time 0.004
    continuation MLMC    solution 5.8100     number of levels 3      time 0.006
    MLQMC                solution 5.6972     number of levels 3      time 0.142
    continuation MLQMC   solution 5.7100     number of levels 3      time 0.000
tolerance = 5.0000e-01, repetition = 4/5
    MLMC                 solution 5.8077     number of levels 3      time 0.004
    continuation MLMC    solution 5.6728     number of levels 4      time 0.009
    MLQMC                solution 5.7058     number of levels 3      time 0.155
    continuation MLQMC   solution 5.7118     number of levels 3      time 0.000
tolerance = 5.0000e-01, repetition = 5/5
    MLMC                 solution 5.4125     number of levels 3      time 0.004
    continuation MLMC    solution 5.8328     number of levels 4      time 0.009
    MLQMC                solution 5.6996     number of levels 3      time 0.153
    continuation MLQMC   solution 5.7150     number of levels 3      time 0.000
tolerance = 1.5811e-01, repetition = 1/5
    MLMC                 solution 5.7598     number of levels 6      time 0.022
    continuation MLMC    solution 5.7586     number of levels 4      time 0.016
    MLQMC                solution 5.7358     number of levels 4      time 0.276
    continuation MLQMC   solution 5.7015     number of levels 3      time 0.021
tolerance = 1.5811e-01, repetition = 2/5
    MLMC                 solution 5.6662     number of levels 6      time 0.022
    continuation MLMC    solution 5.7764     number of levels 4      time 0.028
    MLQMC                solution 5.7354     number of levels 4      time 0.237
```

(continues on next page)

(continued from previous page)

```
    continuation MLQMC   solution 5.7456     number of levels 4      time 0.063
tolerance = 1.5811e-01, repetition = 3/5
    MLMC                 solution 5.7615     number of levels 6      time 0.020
    continuation MLMC    solution 5.6784     number of levels 3      time 0.012
    MLQMC                solution 5.7238     number of levels 4      time 0.249
    continuation MLQMC   solution 5.7459     number of levels 4      time 0.069
tolerance = 1.5811e-01, repetition = 4/5
    MLMC                 solution 5.7532     number of levels 6      time 0.022
    continuation MLMC    solution 5.7984     number of levels 4      time 0.016
    MLQMC                solution 5.7209     number of levels 4      time 0.272
    continuation MLQMC   solution 5.7137     number of levels 3      time 0.017
tolerance = 1.5811e-01, repetition = 5/5
    MLMC                 solution 5.7755     number of levels 5      time 0.017
    continuation MLMC    solution 5.7146     number of levels 4      time 0.016
    MLQMC                solution 5.7429     number of levels 4      time 0.232
    continuation MLQMC   solution 5.7075     number of levels 3      time 0.019
tolerance = 5.0000e-02, repetition = 1/5
    MLMC                 solution 5.7686     number of levels 7      time 0.139
    continuation MLMC    solution 5.7658     number of levels 6      time 0.125
    MLQMC                solution 5.7454     number of levels 5      time 0.412
    continuation MLQMC   solution 5.7357     number of levels 4      time 0.024
tolerance = 5.0000e-02, repetition = 2/5
    MLMC                 solution 5.7597     number of levels 7      time 0.138
    continuation MLMC    solution 5.7266     number of levels 4      time 0.068
    MLQMC                solution 5.7454     number of levels 5      time 0.434
    continuation MLQMC   solution 5.7327     number of levels 4      time 0.025
tolerance = 5.0000e-02, repetition = 3/5
    MLMC                 solution 5.7660     number of levels 7      time 0.144
    continuation MLMC    solution 5.7496     number of levels 4      time 0.104
    MLQMC                solution 5.7476     number of levels 5      time 0.516
    continuation MLQMC   solution 5.7350     number of levels 4      time 0.024
tolerance = 5.0000e-02, repetition = 4/5
    MLMC                 solution 5.7354     number of levels 7      time 0.136
    continuation MLMC    solution 5.7693     number of levels 5      time 0.079
    MLQMC                solution 5.7501     number of levels 5      time 0.381
    continuation MLQMC   solution 5.7338     number of levels 4      time 0.024
tolerance = 5.0000e-02, repetition = 5/5
    MLMC                 solution 5.7545     number of levels 7      time 0.122
    continuation MLMC    solution 5.7705     number of levels 5      time 0.067
    MLQMC                solution 5.7431     number of levels 5      time 0.385
    continuation MLQMC   solution 5.7446     number of levels 5      time 0.108
tolerance = 1.5811e-02, repetition = 1/5
    MLMC                 solution 5.7596     number of levels 8      time 1.203
    continuation MLMC    solution 5.7656     number of levels 6      time 1.068
    MLQMC                solution 5.7566     number of levels 6      time 1.041
    continuation MLQMC   solution 5.7557     number of levels 6      time 0.369
tolerance = 1.5811e-02, repetition = 2/5
    MLMC                 solution 5.7599     number of levels 8      time 1.327
    continuation MLMC    solution 5.7504     number of levels 5      time 0.860
    MLQMC                solution 5.7585     number of levels 7      time 1.350
    continuation MLQMC   solution 5.7553     number of levels 6      time 0.316
tolerance = 1.5811e-02, repetition = 3/5
```

(continues on next page)

(continued from previous page)

```
    MLMC                solution 5.7632     number of levels 8      time 1.261
    continuation MLMC   solution 5.7595     number of levels 6      time 1.172
    MLQMC               solution 5.7545     number of levels 6      time 1.080
    continuation MLQMC  solution 5.7464     number of levels 5      time 0.150
tolerance = 1.5811e-02, repetition = 4/5
    MLMC                solution 5.7655     number of levels 8      time 1.122
    continuation MLMC   solution 5.7494     number of levels 6      time 1.079
    MLQMC               solution 5.7597     number of levels 7      time 1.148
    continuation MLQMC  solution 5.7476     number of levels 5      time 0.220
tolerance = 1.5811e-02, repetition = 5/5
    MLMC                solution 5.7635     number of levels 8      time 1.509
    continuation MLMC   solution 5.7565     number of levels 6      time 1.181
    MLQMC               solution 5.7548     number of levels 6      time 0.913
    continuation MLQMC  solution 5.7483     number of levels 5      time 0.140
tolerance = 5.0000e-03, repetition = 1/5
    MLMC                solution 5.7619     number of levels 10     time 14.429
    continuation MLMC   solution 5.7596     number of levels 7      time 9.738
    MLQMC               solution 5.7614     number of levels 8      time 6.359
    continuation MLQMC  solution 5.7594     number of levels 7      time 3.222
tolerance = 5.0000e-03, repetition = 2/5
    MLMC                solution 5.7634     number of levels 10     time 14.773
    continuation MLMC   solution 5.7591     number of levels 7      time 13.011
    MLQMC               solution 5.7610     number of levels 8      time 7.475
    continuation MLQMC  solution 5.7600     number of levels 7      time 3.833
tolerance = 5.0000e-03, repetition = 3/5
    MLMC                solution 5.7644     number of levels 10     time 17.306
    continuation MLMC   solution 5.7585     number of levels 8      time 13.856
    MLQMC               solution 5.7615     number of levels 8      time 6.449
    continuation MLQMC  solution 5.7588     number of levels 7      time 3.534
tolerance = 5.0000e-03, repetition = 4/5
    MLMC                solution 5.7617     number of levels 10     time 16.390
    continuation MLMC   solution 5.7583     number of levels 7      time 12.321
    MLQMC               solution 5.7611     number of levels 8      time 6.453
    continuation MLQMC  solution 5.7596     number of levels 7      time 2.596
tolerance = 5.0000e-03, repetition = 5/5
    MLMC                solution 5.7638     number of levels 10     time 15.755
    continuation MLMC   solution 5.7583     number of levels 8      time 14.382
    MLQMC               solution 5.7609     number of levels 8      time 7.762
    continuation MLQMC  solution 5.7587     number of levels 7      time 4.633
```

Compute and plot the asymptotic cost complexity.

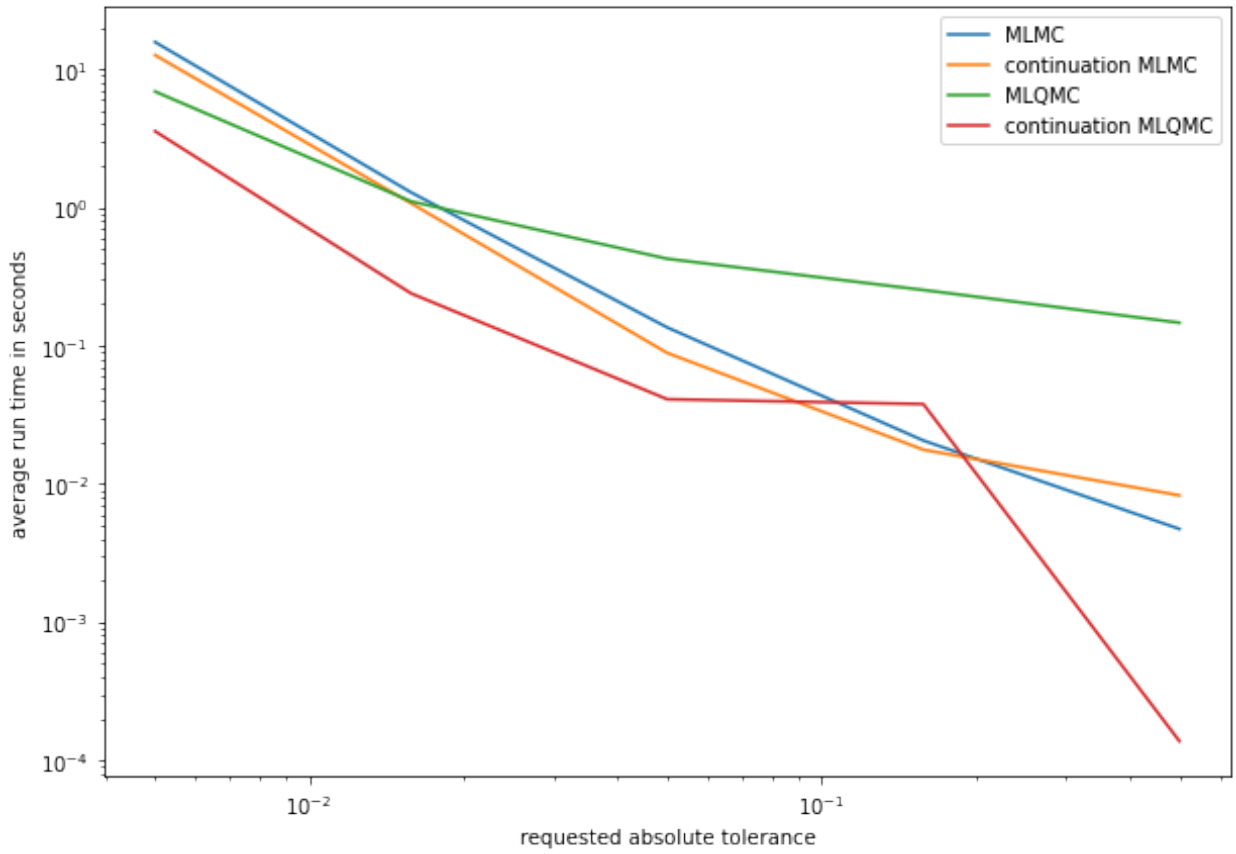
```
avg_time = {}
for method in range(4):
    avg_time[method] = [np.mean([times[t, r][method] for r in range(repetitions)]) for t␣
↪in range(len(tolerances))]
```

```
plt.figure(figsize=(10,7))
plt.plot(tolerances, avg_time[0], label="MLMC")
plt.plot(tolerances, avg_time[1], label="continuation MLMC")
plt.plot(tolerances, avg_time[2], label="MLQMC")
plt.plot(tolerances, avg_time[3], label="continuation MLQMC")
```

(continues on next page)

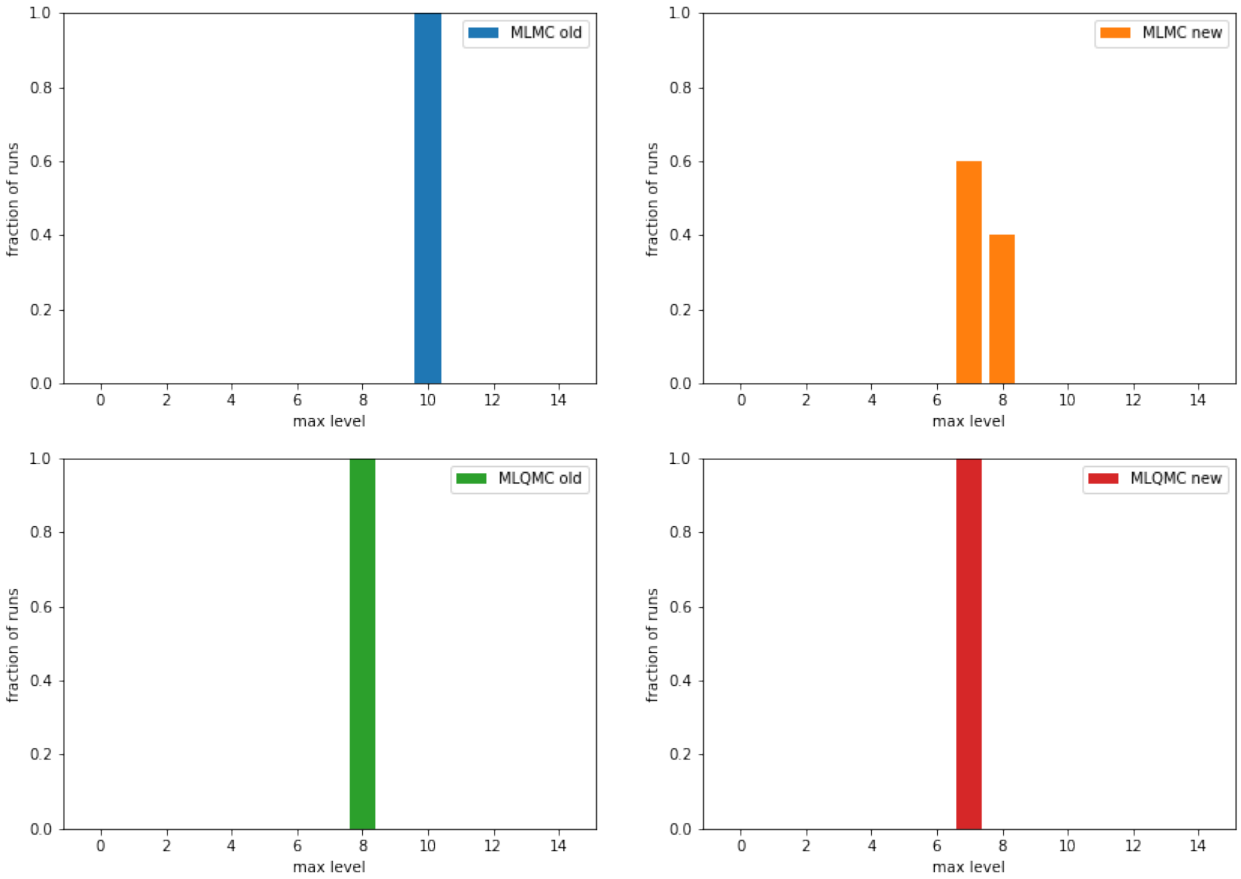
(continued from previous page)

```
plt.xscale("log")
plt.yscale("log")
plt.xlabel("requested absolute tolerance")
plt.ylabel("average run time in seconds")
plt.legend();
```




```
max_levels = {}
for method in range(4):
    levels_rep = np.array([levels[len(tolerances)-1, r][method] for r in␣
↪range(repetitions)])
    max_levels[method] = [np.count_nonzero(levels_rep == level)/repetitions for level in␣
↪range(15)]
```

```
plt.figure(figsize=(14,10))
plt.subplot(2,2,1); plt.bar(range(15), max_levels[0], label="MLMC old", color="C0"); plt.
↪xlabel("max level"); plt.ylabel("fraction of runs"); plt.ylim(0, 1); plt.legend()
plt.subplot(2,2,2); plt.bar(range(15), max_levels[1], label="MLMC new", color="C1"); plt.
↪xlabel("max level"); plt.ylabel("fraction of runs"); plt.ylim(0, 1); plt.legend()
plt.subplot(2,2,3); plt.bar(range(15), max_levels[2], label="MLQMC old", color="C2");␣
↪plt.xlabel("max level"); plt.ylabel("fraction of runs"); plt.ylim(0, 1); plt.legend()
plt.subplot(2,2,4); plt.bar(range(15), max_levels[3], label="MLQMC new", color="C3");␣
↪plt.xlabel("max level"); plt.ylabel("fraction of runs"); plt.ylim(0, 1); plt.legend();
```



5.16 Control Variates in QMCPy

This notebook demonstrates QMCPy's current support for control variates.

5.16.1 Setup

```
from qmcpy import *
from numpy import *
from qmcpy import *
from numpy import *
```

```
from matplotlib import pyplot
%matplotlib inline
size = 20
pyplot.rc('font', size=size)          # controls default text sizes
pyplot.rc('axes', titlesize=size)     # fontsize of the axes title
pyplot.rc('axes', labelsize=size)     # fontsize of the x and y labels
pyplot.rc('xtick', labelsize=size)    # fontsize of the tick labels
pyplot.rc('ytick', labelsize=size)    # fontsize of the tick labels
pyplot.rc('legend', fontsize=size)    # legend fontsize
pyplot.rc('figure', titlesize=size)   # fontsize of the figure title
```

```
def compare(problem,discrete_distrib,stopping_crit,abs_tol):
  g1,cvs,cvmus = problem(discrete_distrib)
  sc1 = stopping_crit(g1,abs_tol=abs_tol)
  name = type(sc1).__name__
  print('Stopping Criterion: %-15s absolute tolerance: %-5.1e'%(name,abs_tol))
  sol,data = sc1.integrate()
  print('\tW CV:  Solution %-10.2f time %-10.2f samples %.1e'%(sol,data.time_integrate,
↪data.n_total))
  sc1 = stopping_crit(g1,abs_tol=abs_tol,control_variates=cvs,control_variate_
↪means=cvmus)
  solcv,datacv = sc1.integrate()
  print('\tWO CV: Solution %-10.2f time %-10.2f samples %.1e'%(solcv,datacv.time_
↪integrate,datacv.n_total))
  print('\tControl variates took %.1f%% the time and %.1f%% the samples\n'%\
        (100*datacv.time_integrate/data.time_integrate,100*datacv.n_total/data.n_total))
```

5.16.2 Problem 1: Polynomial Function

We will integrate

$$g(t) = 10t_1 - 5t_2^2 + 2t_3^3$$

with true measure $\mathcal{U}[0,2]^3$ and control variates

$$\hat{g}_1(t) = t_1$$

and

$$\hat{g}_2(t) = t_2^2$$

using the same true measure.

```
# parameters
def poly_problem(discrete_distrib):
  g1 = CustomFun(Uniform(discrete_distrib,0,2),lambda t: 10*t[:,0]-5*t[:,1]**2+t[:,2]**3)
  cv1 = CustomFun(Uniform(discrete_distrib,0,2),lambda t: t[:,0])
  cv2 = CustomFun(Uniform(discrete_distrib,0,2),lambda t: t[:,1]**2)
  return g1,[cv1,cv2],[1,4/3]
compare(poly_problem,IIDStdUniform(3,seed=7),CubMCCLT,abs_tol=1e-2)
compare(poly_problem,IIDStdUniform(3,seed=7),CubMCCLT,abs_tol=1e-2)
compare(poly_problem,Sobol(3,seed=7),CubQMCSobolG,abs_tol=1e-8)
```

```
Stopping Criterion: CubMCCLT        absolute tolerance: 1.0e-02
    W CV:  Solution 5.33       time 0.79       samples 6.7e+06
    WO CV: Solution 5.34       time 0.09       samples 4.2e+05
    Control variates took 11.5% the time and 6.2% the samples

Stopping Criterion: CubMCCLT        absolute tolerance: 1.0e-02
    W CV:  Solution 5.33       time 0.68       samples 6.7e+06
    WO CV: Solution 5.34       time 0.07       samples 4.2e+05
    Control variates took 10.7% the time and 6.2% the samples
```

(continues on next page)

(continued from previous page)

```
Stopping Criterion: CubQMCSobolG    absolute tolerance: 1.0e-08
    W CV:  Solution 5.33       time 0.12       samples 2.6e+05
    WO CV: Solution 5.33       time 0.08       samples 1.3e+05
    Control variates took 65.5% the time and 50.0% the samples
```

5.16.3 Problem 2: Keister Function

This problem will integrate the Keister function while using control variates

$$g_1(x) = \sin(\pi x)$$

and

$$g_2(x) = -3(x - 1/2)^2 + 1.$$

The following code does this problem in one-dimension for visualization purposes, but control variates are compatible with any dimension.

```
def keister_problem(discrete_distrib):
  k = Keister(discrete_distrib)
  cv1 = CustomFun(Uniform(discrete_distrib),lambda x: sin(pi*x).sum(1))
  cv2 = CustomFun(Uniform(discrete_distrib),lambda x: (-3*(x-.5)**2+1).sum(1))
  return k,[cv1,cv2],[2/pi,3/4]
compare(keister_problem,IIDStdUniform(1,seed=7),CubMCCLT,abs_tol=5e-4)
compare(keister_problem,IIDStdUniform(1,seed=7),CubMCCLT,abs_tol=4e-4)
compare(keister_problem,Sobol(1,seed=7),CubQMCSobolG,abs_tol=1e-7)
```

```
Stopping Criterion: CubMCCLT        absolute tolerance: 5.0e-04
    W CV:  Solution 1.38       time 1.21       samples 9.5e+06
    WO CV: Solution 1.38       time 0.14       samples 4.5e+05
    Control variates took 11.8% the time and 4.8% the samples

Stopping Criterion: CubMCCLT        absolute tolerance: 4.0e-04
    W CV:  Solution 1.38       time 1.99       samples 1.5e+07
    WO CV: Solution 1.38       time 0.21       samples 6.8e+05
    Control variates took 10.5% the time and 4.6% the samples

Stopping Criterion: CubQMCSobolG    absolute tolerance: 1.0e-07
    W CV:  Solution 1.38       time 0.44       samples 1.0e+06
    WO CV: Solution 1.38       time 0.47       samples 1.0e+06
    Control variates took 108.0% the time and 100.0% the samples
```

5.16.4 Problem 3: Option Pricing

We will use a European Call Option as a control variate for pricing the Asian Call Option using various stopping criterion, as done for problem 1

```
call_put = 'call'
start_price = 100
strike_price = 125
```

(continues on next page)

(continued from previous page)

```
volatility = .75
interest_rate = .01 # 1% interest
t_final = 1 # 1 year
def option_problem(discrete_distrib):
  eurocv = EuropeanOption(discrete_distrib,volatility,start_price,strike_price,interest_
↪rate,t_final,call_put)
  aco = AsianOption(discrete_distrib,volatility,start_price,strike_price,interest_rate,t_
↪final,call_put)
  mu_eurocv = eurocv.get_exact_value()
  return aco,[eurocv],[mu_eurocv]
compare(option_problem,IIDStdUniform(4,seed=7),CubMCCLT,abs_tol=5e-2)
compare(option_problem,IIDStdUniform(4,seed=7),CubMCCLT,abs_tol=5e-2)
compare(option_problem,Sobol(4,seed=7),CubQMCSobolG,abs_tol=1e-3)
```

```
Stopping Criterion: CubMCCLT        absolute tolerance: 5.0e-02
    W CV:  Solution 9.54       time 0.89       samples 2.1e+06
    WO CV: Solution 9.55       time 0.82       samples 1.1e+06
    Control variates took 91.7% the time and 51.9% the samples

Stopping Criterion: CubMCCLT        absolute tolerance: 5.0e-02
    W CV:  Solution 9.54       time 0.84       samples 2.1e+06
    WO CV: Solution 9.55       time 0.82       samples 1.1e+06
    Control variates took 97.0% the time and 51.9% the samples

Stopping Criterion: CubQMCSobolG    absolute tolerance: 1.0e-03
    W CV:  Solution 9.55       time 0.44       samples 5.2e+05
    WO CV: Solution 9.55       time 0.57       samples 5.2e+05
    Control variates took 128.1% the time and 100.0% the samples
```

5.17 Elliptic PDE

```
import copy
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
from scipy.special import gamma, kv
from qmcpy.integrand._integrand import Integrand
from qmcpy.accumulate_data.mlmc_data import MLMCData
from qmcpy.accumulate_data.mlqmc_data import MLQMCData
import qmcpy as qp
```

```
# matplotlib options
rc_fonts = {
    "text.usetex": True,
    "font.size": 14,
    "mathtext.default": "regular",
    "axes.titlesize": 14,
    "axes.labelsize": 14,
    "legend.fontsize": 14,
```

(continues on next page)

(continued from previous page)

```
    "xtick.labelsize": 12,
    "ytick.labelsize": 12,
    "figure.titlesize": 16,
    "font.family": "serif",
    "font.serif": "computer modern roman",
}
mpl.rcParams.update(rc_fonts)
```

```
# set random seed for reproducability
np.random.seed(9999)
```

We will apply various multilevel Monte Carlo and multilevel quasi-Monte Carlo methods to approximate the expected value of a quantity of interest derived from the solution of a one-dimensional partial differential equation (PDE), where the diffusion coefficient of the PDE is a lognormal Gaussian random field. This example problem serves as an important benchmark problem for various methods in the uncertainty quantification and quasi-Monte Carlo literature. It is often referred to as *the fruitfly problem* of uncertainty quantification.

5.17.1 1. Problem definition

Let Q be a quantity of interest derived from the solution $u(x,\omega)$ of the one-dimensional partial differential equation (PDE)

$$-\frac{d}{dx}\left(a(x,\omega)\frac{d}{dx}u(x,\omega)\right) = f(x), \quad 0 \le x \le 1,$$

$$u(0,\cdot) = u_0,$$

$$u(1,\cdot) = u_1.$$

The notation $u(x,\omega)$ is used to indicate that the solution depends on both the spatial variable x and the uncertain parameter ω. This uncertainty is present because the diffusion coefficient, $a(x,\omega)$, is given by a lognormal Gaussian random field with given covariance function. A common choice for the covariance function is the so-called Matérn covariance function

$$c(x,y) = \hat{c}\left(\frac{\|x-y\|}{\lambda}\right) \qquad \hat{c}(r) = \frac{2^{1-\nu}}{\Gamma(\nu)} r^{\nu} K_{\nu}(r)$$

with Γ the gamma function and K_ν the Bessel function of the second kind. This covariance function has two parameters: λ, the length scale, and ν, the smoothness parameter.

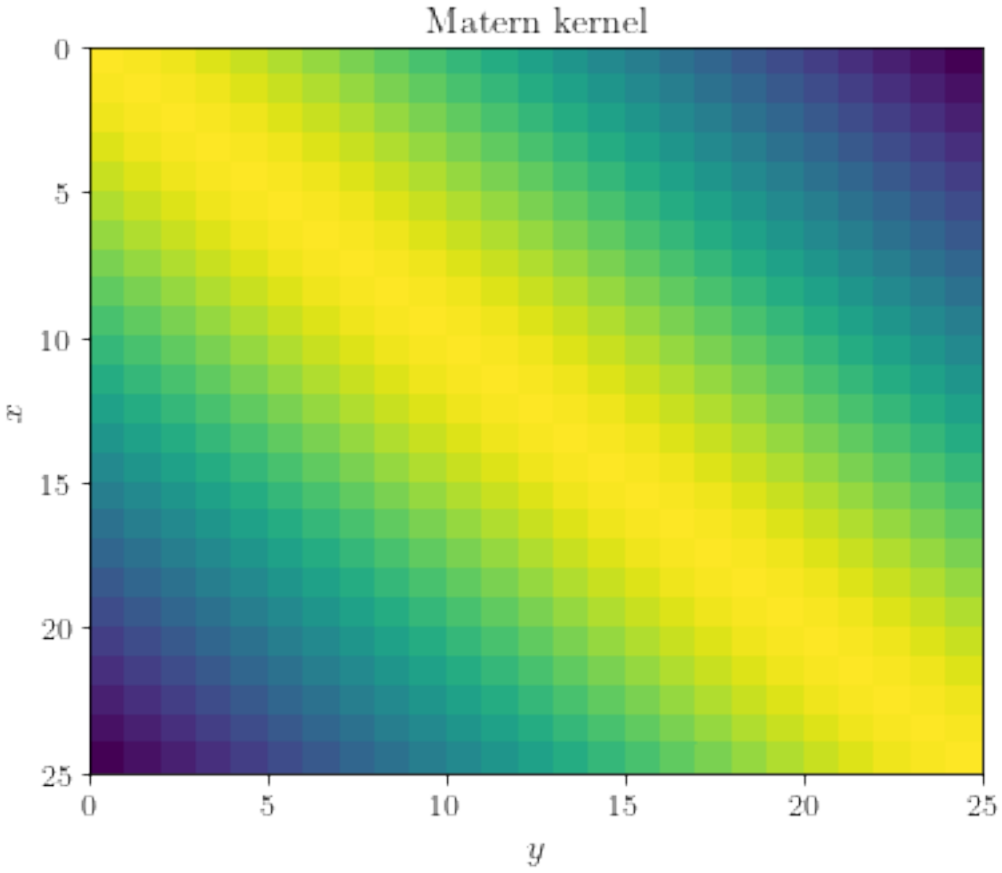
We begin by defining the Matérn covariance function `Matern(x, y)`:

```
def Matern(x, y, smoothness=1, lengthscale=1):
    distance = abs(x - y)
    r = distance/lengthscale
    prefactor = 2**(1-smoothness)/gamma(smoothness)
    term1 = r**smoothness
    term2 = kv(smoothness, r)
    np.fill_diagonal(term2, 1)
    cov = prefactor * term1 * term2
    np.fill_diagonal(cov, 1)
    return cov
```

Let's take a look at the covariance matrix obtained by evaluating the covariance function in `n=25` equidistant points in `[0, 1]`.

```
def get_covariance_matrix(pts, smoothness=1, lengthscale=1):
    X, Y = np.meshgrid(pts, pts)
    return Matern(X, Y, smoothness, lengthscale)
```

```
n = 25
pts = np.linspace(0, 1, num=n)
fig, ax = plt.subplots(figsize=(6, 5))
ax.pcolor(get_covariance_matrix(pts).T)
ax.invert_yaxis()
ax.set_ylabel(r"$x$")
ax.set_xlabel(r"$y$")
ax.set_title(f"Matern kernel")
plt.show()
```



A lognormal Gaussian random field $a(x,\omega)$ can be expressed as $a(x,\omega) = \exp(b(x,\omega))$, where $b(x,\omega)$ is a Gaussian random field. Samples of the Gaussian random field $b(x,\omega)$ can be computed from a factorization of the covariance matrix. Specifically, suppose we have a spectral (eigenvalue) expansion of the covariance matrix C as

$$C = VWV^T,$$

then samples of the Gaussian random field can be computed as

$$\boldsymbol{b} = S\boldsymbol{x},$$

where $S = VW^{1/2}$ and $\boldsymbol{x}$ is a vector of standard normal independent and identically distributed random variables.

This is easy to see, since

$$\mathbb{E}[\boldsymbol{b}] = \mathbb{E}[S\boldsymbol{x}] = S\mathbb{E}[\boldsymbol{x}] = \boldsymbol{0}$$

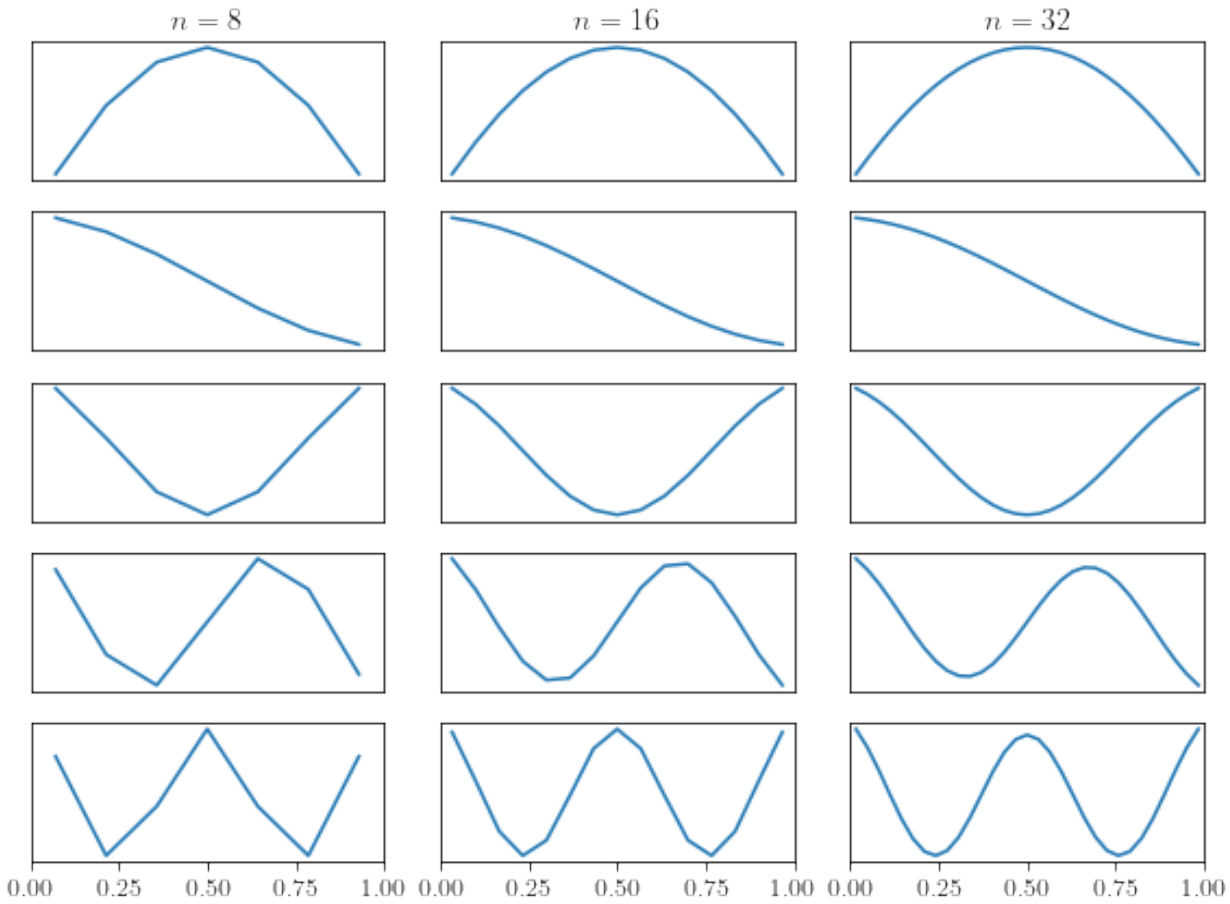
$$\mathbb{E}[\boldsymbol{b}\boldsymbol{b}^T] = \mathbb{E}[S\boldsymbol{x}\boldsymbol{x}^T S^T] = S\mathbb{E}[\boldsymbol{x}\boldsymbol{x}^T]S^T = SS^T = VWV^T = C.$$

First, let's compute an eigenvalue decomposition of the covariance matrix.

```
def get_eigenpairs(n, smoothness=1, lengthscale=1):
    h = 1/(n-1)
    pts = np.linspace(h/2, 1 - h/2, num=n - 1)
    cov = get_covariance_matrix(pts, smoothness, lengthscale)
    w, v = np.linalg.eig(cov)
    # ensure all eigenvectors are correctly oriented
    for col in range(v.shape[1]):
        if v[0, col] < 0:
            v[:, col] *= -1
    return pts, w, v
```

Next, we plot the eigenfunctions for different values of n, the number of grid points.

```
n = [8, 16, 32] # list of number of gridpoints to plot
m = 5 # number of eigenfunctions to plot
fig, axes = plt.subplots(m, len(n), figsize=(8, 6))
for j, k in enumerate(n):
    x, w, v = get_eigenpairs(k)
    for i in range(m):
        axes[i, j].plot(x, v[:, i])
        axes[i, j].set_xlim(0, 1)
        axes[i, j].get_yaxis().set_ticks([])
        if i < m - 1:
            axes[i, j].get_xaxis().set_ticks([])
        if i == 0:
            axes[i, j].set_title(r"$n = " + repr(k) + r"$")
plt.tight_layout()
```

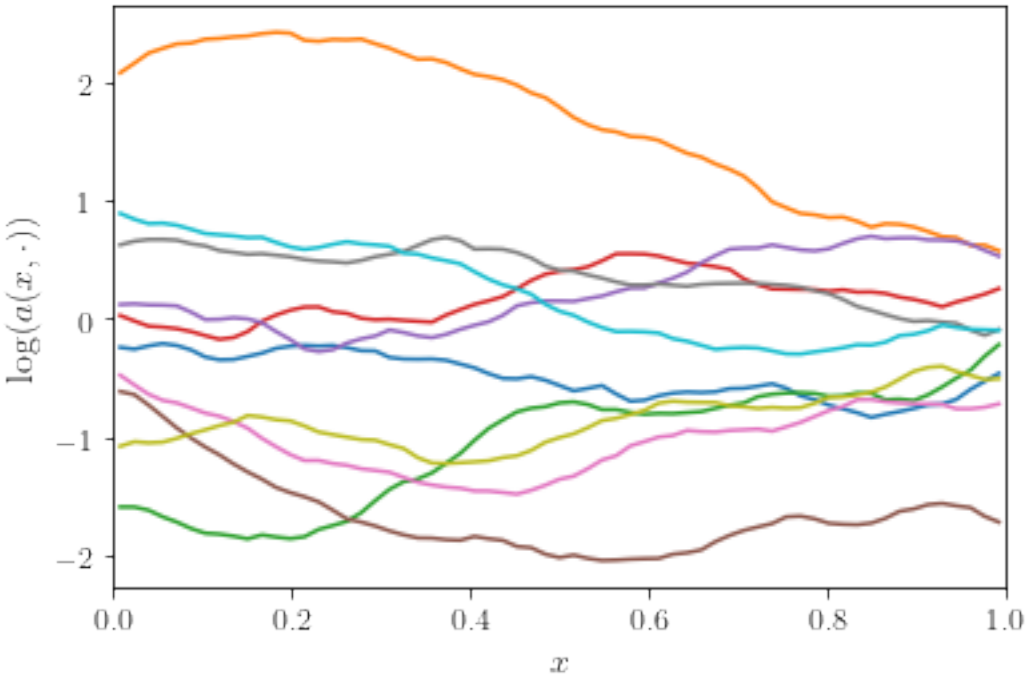


With this eigenvalue decomposition, we can compute samples of the Gaussian random field $b(x, \omega)$, and hence, also of the lognormal Gaussian random field $a(x, \omega) = \exp(b(x, \omega))$, since $\boldsymbol{b} = VW^{1/2}\boldsymbol{x}$.

```
def evaluate(w, v, y=None):
    if y is None:
        y = np.random.randn(len(w) - 1)
    m = len(y)
    return v[:, :m] @ np.diag(np.sqrt(w[:m])) @ y
```

Let's plot a couple of realizations of the Gaussian random field $b(x, \omega)$.

```
n = 64
x, w, v = get_eigenpairs(n)
fig, ax = plt.subplots(figsize=(6, 4))
for _ in range(10):
    ax.plot(x, evaluate(w, v))
ax.set_xlim(0, 1)
ax.set_xlabel(r"$x$")
ax.set_ylabel(r"$\log(a(x, \cdot))$")
plt.show()
```



Now that we are able to compute realizations of the Gaussian random field, a next step is to compute a numerical solution of the PDE

$$-\frac{d}{dx}\left(a(x,\omega)\frac{d}{dx}u(x,\omega)\right) = f(x), \quad 0 \le x \le 1.$$

Using a straightforward finite-difference approximation, it is easy to show that the numerical solution $\boldsymbol{u}$ is the solution of a tridiagonal system. The solutions of such a tridiagonal system can be easily obtained in $O(n)$ (linear) time using the tridiagonal matrix algorithm (also known as the Thomas algorithm). More details can be found here.

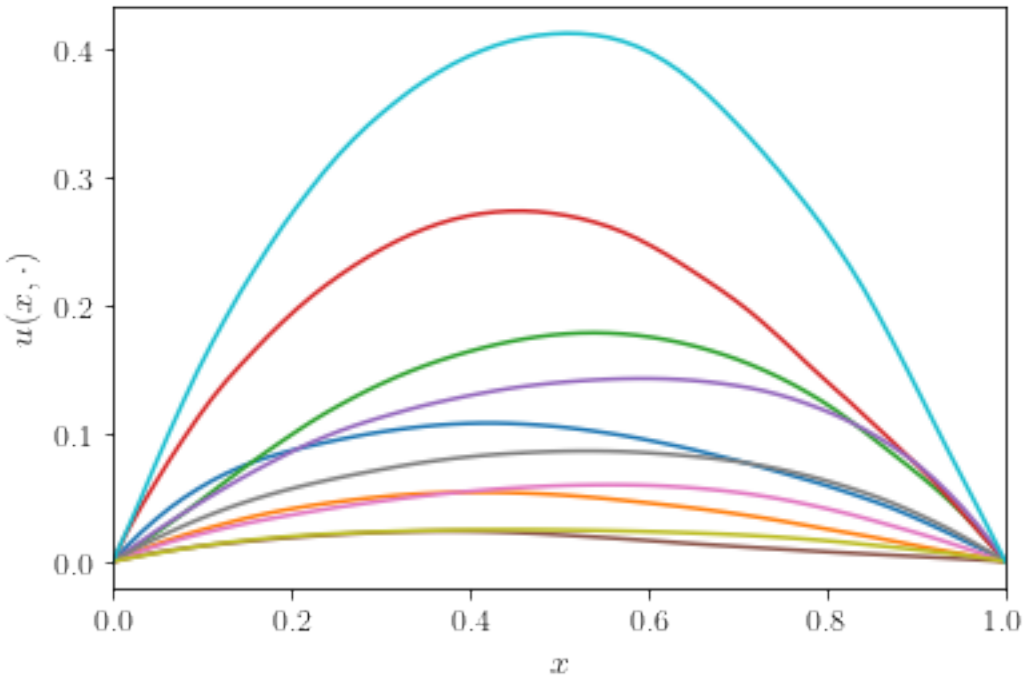
```
def thomas(a, b, c, d):
    n = len(b)
    x = np.zeros(n)
    for i in range(1, n):
        w = a[i-1]/b[i-1]
        b[i] -= w*c[i-1]
        d[i] -= w*d[i-1]
    x[n-1] = d[n-1]/b[n-1]
    for i in reversed(range(n-1)):
        x[i] = (d[i] - c[i]*x[i+1])/b[i]
    return x
```

For the remainder of this notebook, we will assume that the source term $f(x) = 1$ and Dirichlet boundary conditions $u(0) = u(1) = 0$.

```
def pde_solve(a):
    n = len(a)
    b = np.full((n-1, 1), 1/n**2)
    x = thomas(-a[1:n-1], a[:n-1] + a[1:], -a[1:n-1], b)
    return np.insert(x, [0, n-1], [0, 0])
```

Let's compute and plot a couple of solutions $u(x,\omega)$.

```
n = 64
_, w, v = get_eigenpairs(n)
x = np.linspace(0, 1, num=n)
fig, ax = plt.subplots(figsize=(6, 4))
for _ in range(10):
    a = np.exp(evaluate(w, v))
    u = pde_solve(a)
    ax.plot(x, u)
ax.set_xlim(0, 1)
ax.set_xlabel(r"$x$")
ax.set_ylabel(r"$u(x, \cdot)$")
plt.show()
```



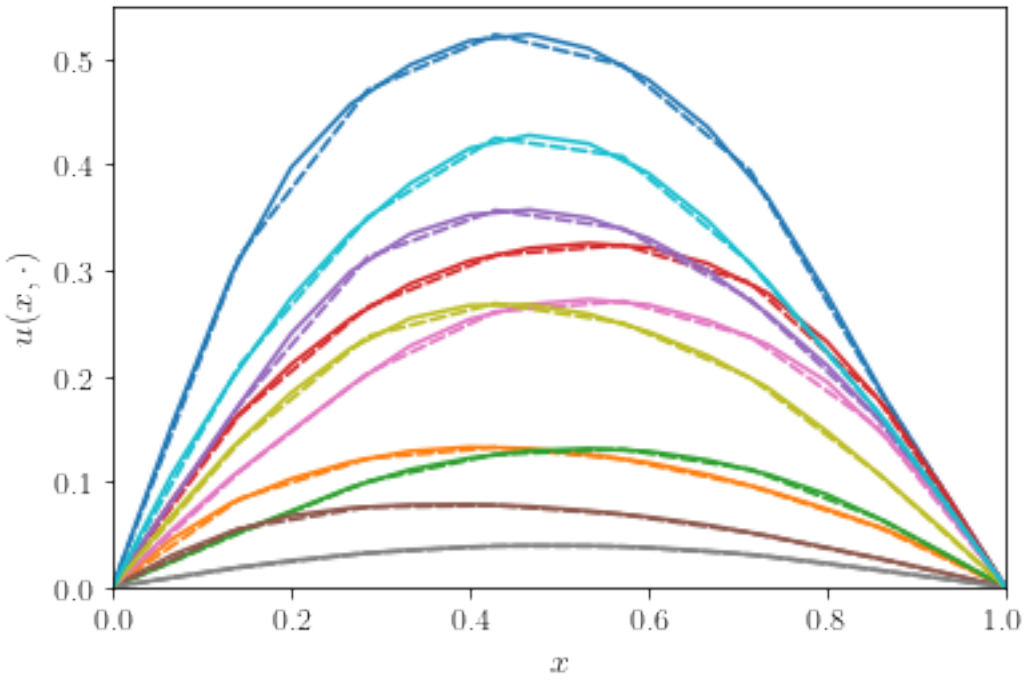
In the multilevel Monte Carlo method, we will rely on the ability to generate "correlated" solutions of the PDE with varying mesh sizes. Such a correlated solutions can be used as efficient control variates to reduce the variance (or statistical error) in the approximation of the expected value $\mathbb{E}[Q]$. Since we are using a factorization of the covariance matrix to generate realizations of the Gaussian random field, it is quite easy to obtain correlated samples: when sampling from the "coarse" solution level, use the same set of random numbers used to sample from the "fine" solution level, but truncated to the appropriate size. Since the eigenvalue decomposition will reveal the most important modes in the covariance matrix, that same eigenvalue decomposition on a "coarse" approximation level will contain the same eigenfunctions, represented on the coarse grid. Let's illustrate this property on an example using `n = 16` grid points for the fine solution level and `n = 8` grid points for the coarse solution level.

```
nf = 16
nc = nf//2
_, wf, vf = get_eigenpairs(nf)
_, wc, vc = get_eigenpairs(nc)
xf = np.linspace(0, 1, num=nf)
xc = np.linspace(0, 1, num=nc)
fig, ax = plt.subplots(figsize=(6, 4))
```

(continues on next page)

(continued from previous page)

```
for _ in range(10):
    yf = np.random.randn(nf - 1)
    af = np.exp(evaluate(wf, vf, y=yf))
    uf = pde_solve(af)
    ax.plot(xf, uf)
    yc = yf[:nc - 1]
    ac = np.exp(evaluate(wc, vc, y=yc))
    uc = pde_solve(ac)
    ax.plot(xc, uc, color=ax.lines[-1].get_color(), linestyle="dashed", dash_capstyle=
↪"round")
ax.set_xlim(0, 1)
ax.set_ylim(bottom=0)
ax.set_xlabel(r"$x$")
ax.set_ylabel(r"$u(x, \cdot)$")
plt.show()
```



The better the coarse solution matches the fine grid solution, the more efficient the multilevel methods in Section 3 will perform.

5.17.2 2. Single-level methods

Let's begin by using the single-level Monte Carlo and quasi-Monte Carlo methods to compute the expected value $\mathbb{E}[Q]$. As quantity of interest Q we take the solution of the PDE at $x = 1/2$, i.e., $Q = u(1/2, \cdot)$.

To integrate the elliptic PDE problem into `QMCPy`, we construct a simple class as follows:

```
class EllipticPDE(Integrand):

    def __init__(self, sampler, smoothness=1, lengthscale=1):
```

(continues on next page)

(continued from previous page)

```
        self.parameters = ["smoothness", "lengthscale", "n"]
        self.smoothness = smoothness
        self.lengthscale = lengthscale
        self.n = len(sampler.gen_samples(n=1)[0]) + 1
        self.compute_eigenpairs()
        self.sampler = sampler
        self.true_measure = qp.Gaussian(self.sampler)
        super(EllipticPDE, self).__init__(dimension_indv=1,dimension_comb=1,
↪parallel=False)

    def compute_eigenpairs(self):
        _, w, v = get_eigenpairs(self.n)
        self.eigenpairs = w, v

    def g(self, x):
        n, d = x.shape
        return np.array([self.__g(x[j, :].T) for j in range(n)])

    def __g(self, x):
        w, v = self.eigenpairs
        a = np.exp(evaluate(w, v, y=x))
        u = pde_solve(a)
        return u[len(u)//2]

    def _spawn(self, level, sampler):
        return EllipticPDE(sampler, smoothness=self.smoothness, lengthscale=self.
↪lengthscale)
```

```
# Custom print function
def print_data(data):
    for key, val in vars(data).items():
        kv = getattr(data, key)
        if hasattr(kv, "parameters"):
            print(f"{key}: {type(val).__name__}")
            for param in kv.parameters:
                print(f"\t{param}: {getattr(kv, param)}")
    for param in data.parameters:
        print(f"{param}: {getattr(data, param)}")
```

```
# Main function to test different methods
def test(problem, sampler, stopping_criterium, abs_tol=5e-3, verbose=True, **kwargs):
    integrand = problem(sampler)
    solution, data = stopping_criterium(integrand, abs_tol=abs_tol, **kwargs).integrate()
    if verbose:
        print(data)
        print("\nComputed solution %.3f in %.2f s"%(solution, data.time_integrate))
```

Next, let's apply simple Monte Carlo to approximate the expected value $\mathbb{E}[Q]$. The Monte Carlo estimator for $\mathbb{E}[Q]$ is simply the sample average over a finite set of samples, i.e.,

$$Q_N^{\text{MC}} := \frac{1}{N} \sum_{n=0}^{N-1} Q^{(n)},$$

where $Q^{(n)} := u(1/2, \boldsymbol{x}^{(n)})$ and we explicitly denote the dependency of Q on the standard normal random numbers $\boldsymbol{x}$ used to sample from the Gaussian random field. We will continue to increase the number of samples N until a certain error criterion is satisfied.

```
# MC
test(EllipticPDE, qp.IIDStdUniform(32), qp.CubMCCLT)
```

```
MeanVarData (AccumulateData Object)
    solution        0.190
    error_bound     0.005
    n_total         19186
    n               18162
    levels          1
    time_integrate  4.307
CubMCCLT (StoppingCriterion Object)
    abs_tol         0.005
    rel_tol         0
    n_init          2^(10)
    n_max           10000000000
    inflate         1.200
    alpha           0.010
EllipticPDE (Integrand Object)
    smoothness      1
    lengthscale     1
    n               33
Gaussian (TrueMeasure Object)
    mean            0
    covariance      1
    decomp_type     PCA
IIDStdUniform (DiscreteDistribution Object)
    d               2^(5)
    entropy         326194454235761696154918970274080109080
    spawn_key       ()

Computed solution 0.190 in 4.31 s
```

The solution should be ≈ 0.189.

Similarly, the quasi-Monte Carlo estimator for $\mathbb{E}[Q]$ is defined as

$$\mathcal{Q}_N^{\text{QMC}} := \frac{1}{N} \sum_{n=0}^{N-1} Q^{(n)},$$

where $Q^{(n)} := u(1/2, \boldsymbol{t}^{(n)})$ with $\boldsymbol{t}^{(n)}$ the nth low-discrepancy point transformed to the distribution of interest. For our elliptic PDE, this means that the quasi-Monte Carlo points, generated inside the unit cube $[0,1)^d$, are mapped to $\mathbb{R}^d$.

Because the quasi-Monte Carlo estimator doesn't come with a reliable error estimator, we run K different quasi-Monte Carlo estimators in parallel. The sample variance over these K different estimators can then be used as an error estimator.

```
# QMC
test(EllipticPDE, qp.Lattice(32), qp.CubQMCCLT, n_init=32)
```

```
MeanVarDataRep (AccumulateData Object)
    solution        0.189
    comb_bound_low  0.185
    comb_bound_high 0.192
    comb_flags      1
    n_total         2^(11)
    n               2^(11)
    n_rep           2^(7)
    time_integrate  0.490
CubQMCCLT (StoppingCriterion Object)
    inflate         1.200
    alpha           0.010
    abs_tol         0.005
    rel_tol         0
    n_init          2^(5)
    n_max           2^(30)
    replications    2^(4)
EllipticPDE (Integrand Object)
    smoothness      1
    lengthscale     1
    n               33
Gaussian (TrueMeasure Object)
    mean            0
    covariance      1
    decomp_type     PCA
Lattice (DiscreteDistribution Object)
    d               2^(5)
    dvec            [ 0  1  2 ... 29 30 31]
    randomize       1
    order           natural
    entropy         296354698279282161707952401923456574428
    spawn_key       ()

Computed solution 0.189 in 0.49 s
```

5.17.3 3. Multilevel methods

Implicit to the Monte Carlo and quasi-Monte Carlo methods above is a discretization parameter used in the numerical solution of the PDE. Let's denote this parameter by ℓ, $0 \leq \ell \leq L$. Multilevel methods are based on a telescopic sum expansion for the expected value $\mathbb{E}[Q_L]$, as follows:

$$\mathbb{E}[Q_L] = \mathbb{E}[Q_0] + \mathbb{E}[Q_1 - Q_0] + ... + \mathbb{E}[Q_L - Q_{L-1}].$$

Using a Monte Carlo method for each of the terms on the right hand side yields a multilevel Monte Carlo method. Similarly, using a quasi-Monte Carlo method for each term on the right hand side yields a multilevel quasi-Monte Carlo method.

3.1 Multilevel (quasi-)Monte Carlo

Our class `EllipticPDE` needs some changes to be integrated with the multilevel methods in `QMCPy`.

```
class MLEllipticPDE(Integrand):

    def __init__(self, sampler, smoothness=1, lengthscale=1, _level=None):
        self.l = _level
        self.parameters = ["smoothness", "lengthscale", "n", "nb_of_levels"]
        self.smoothness = smoothness
        self.lengthscale = lengthscale
        dim = sampler.d + 1
        self.nb_of_levels = int(np.log2(dim + 1))
        self.n = [2**(l+1) + 1 for l in range(self.nb_of_levels)]
        self.compute_eigenpairs()
        self.sampler = sampler
        self.true_measure = qp.Gaussian(self.sampler)
        self.leveltype = "adaptive-multi"
        self.sums = np.zeros(6)
        self.cost = 0
        super(MLEllipticPDE, self).__init__(dimension_indv=1,dimension_comb=1,
↪parallel=False)

    def _spawn(self, level, sampler):
        return MLEllipticPDE(sampler, smoothness=self.smoothness, lengthscale=self.
↪lengthscale, _level=level)

    def compute_eigenpairs(self):
        self.eigenpairs = {}
        for l in range(self.nb_of_levels):
            _, w, v = get_eigenpairs(self.n[l])
            self.eigenpairs[l] = w, v

    def g(self, x): # This function is called by keyword reference for the level␣
↪parameter "l"!
        n, d = x.shape

        Qf = np.array([self.__g(x[j, :].T, self.l) for j in range(n)])
        dQ = Qf
        if self.l > 0: # Compute multilevel difference
            dQ -= np.array([self.__g(x[j, :].T, self.l - 1) for j in range(n)])

        self.update_sums(dQ, Qf)
        self.cost = n*nf

        return dQ

    def __g(self, x, l):
        w, v = self.eigenpairs[l]
        n = self.n[l]
        a = np.exp(evaluate(w, v, y=x[:n-1]))
        u = pde_solve(a)
        return u[len(u)//2]
```

(continues on next page)

(continued from previous page)

```
    def update_sums(self, dQ, Qf):
        self.sums[0] = dQ.sum()
        self.sums[1] = (dQ**2).sum()
        self.sums[2] = (dQ**3).sum()
        self.sums[3] = (dQ**4).sum()
        self.sums[4] = Qf.sum()
        self.sums[5] = (Qf**2).sum()

    def _dimension_at_level(self, l):
        return self.n[l]
```

Let's apply multilevel Monte Carlo to the elliptic PDE problem.

```
test(MLEllipticPDE, qp.IIDStdUniform(32), qp.CubMCML)
```

```
MLMCData (AccumulateData Object)
    solution        0.191
    n_total         81898
    levels          3
    n_level         [31258.  5902.  4173.]
    mean_level      [1.901e-01 6.267e-04 1.575e-04]
    var_level       [5.150e-02 3.690e-04 9.224e-05]
    cost_per_sample [16. 16. 16.]
    alpha           1.993
    beta            2.000
    gamma           2^(-1)
    time_integrate  2.223
CubMCML (StoppingCriterion Object)
    rmse_tol        0.002
    n_init          2^(8)
    levels_min      2^(1)
    levels_max      10
    theta           2^(-1)
MLEllipticPDE (Integrand Object)
    smoothness      1
    lengthscale     1
    n               [ 3  5  9 17 33]
    nb_of_levels    5
Gaussian (TrueMeasure Object)
    mean            0
    covariance      1
    decomp_type     PCA
IIDStdUniform (DiscreteDistribution Object)
    d               2^(5)
    entropy         52139997603444977626041483839545508893
    spawn_key       ()

Computed solution 0.191 in 2.22 s
```

Now it's easy to switch to multilevel quasi-Monte Carlo. Just change the discrete distribution from `IIDStdUniform` to `Lattice`.

```
test(MLEllipticPDE, qp.Lattice(32), qp.CubQMCML, n_init=32)
```

```
MLQMCData (AccumulateData Object)
    solution        0.190
    n_total         74752
    n_level         [2048.  256.   32.]
    levels          3
    mean_level      [ 1.900e-01  2.662e-05 -4.765e-05]
    var_level       [8.161e-07 4.455e-07 1.539e-07]
    bias_estimate   2.06e-04
    time_integrate  3.550
CubQMCML (StoppingCriterion Object)
    rmse_tol        0.002
    n_init          2^(5)
    n_max           10000000000
    replications    2^(5)
MLEllipticPDE (Integrand Object)
    smoothness      1
    lengthscale     1
    n               [ 3  5  9 17 33]
    nb_of_levels    5
Gaussian (TrueMeasure Object)
    mean            0
    covariance      1
    decomp_type     PCA
Lattice (DiscreteDistribution Object)
    d               2^(5)
    dvec            [ 0  1  2 ... 29 30 31]
    randomize       1
    order           natural
    entropy         119695915373660257480153154875270156239
    spawn_key       ()

Computed solution 0.190 in 3.55 s
```

3.2 Continuation multilevel (quasi-)Monte Carlo

In the continuation multilevel (quasi-)Monte Carlo method, we run the standard multilevel (quasi-)Monte Carlo method for a sequence of larger tolerances to obtain better estimates of the algorithmic parameters. The continuation multilevel heuristic will generally compute the same solution just a bit faster.

```
test(MLEllipticPDE, qp.IIDStdUniform(32), qp.CubMCMLCont)
```

```
MLMCData (AccumulateData Object)
    solution        0.185
    n_total         18182
    levels          2^(2)
    n_level         [16273.  1288.   337.   256.]
    mean_level      [1.863e-01 1.710e-04 7.073e-04 2.501e-04]
    var_level       [5.082e-02 3.083e-04 2.395e-05 9.872e-07]
    cost_per_sample [16. 16. 16. 16.]
```

(continues on next page)

(continued from previous page)

```
    alpha           2^(-1)
    beta            4.143
    gamma           2^(-1)
    time_integrate  0.863
CubMCMLCont (StoppingCriterion Object)
    rmse_tol        0.002
    n_init          2^(8)
    levels_min      2^(1)
    levels_max      10
    n_tols          10
    tol_mult        1.668
    theta_init      2^(-1)
    theta           0.010
MLEllipticPDE (Integrand Object)
    smoothness      1
    lengthscale     1
    n               [ 3  5  9 17 33]
    nb_of_levels    5
Gaussian (TrueMeasure Object)
    mean            0
    covariance      1
    decomp_type     PCA
IIDStdUniform (DiscreteDistribution Object)
    d               2^(5)
    entropy         567672617367519583229042510455682796 67
    spawn_key       ()

Computed solution 0.185 in 0.86 s
```

```
test(MLEllipticPDE, qp.Lattice(32), qp.CubQMCMLCont, n_init=32)
```

```
MLQMCData (AccumulateData Object)
    solution        0.189
    n_total         41984
    n_level         [1024.  256.   32.]
    levels          3
    mean_level      [ 1.891e-01  1.470e-04 -1.685e-04]
    var_level       [1.582e-06 7.087e-07 3.647e-07]
    bias_estimate   4.66e-04
    time_integrate  1.260
CubQMCMLCont (StoppingCriterion Object)
    rmse_tol        0.002
    n_init          2^(5)
    n_max           10000000000
    replications    2^(5)
    levels_min      2^(1)
    levels_max      10
    n_tols          10
    tol_mult        1.668
    theta_init      2^(-1)
    theta           0.058
```

(continues on next page)

(continued from previous page)

```
MLEllipticPDE (Integrand Object)
    smoothness      1
    lengthscale     1
    n               [ 3  5  9 17 33]
    nb_of_levels    5
Gaussian (TrueMeasure Object)
    mean            0
    covariance      1
    decomp_type     PCA
Lattice (DiscreteDistribution Object)
    d               2^(5)
    dvec            [ 0  1  2 ... 29 30 31]
    randomize       1
    order           natural
    entropy         112259460192958188918668417604759964310
    spawn_key       ()

Computed solution 0.189 in 1.26 s
```

5.17.4 4. Convergence tests

Finally, we will run some convergence tests to see how these methods behave as a function of the error tolerance.

```
# Main function to test convergence for given problem
def test_convergence(problem, sampler, stopping_criterium, abs_tol=1e-3, verbose=True,␣
↪smoothness=1, lengthscale=1, **kwargs):
    integrand = problem(sampler, smoothness=smoothness, lengthscale=lengthscale)
    stopping_crit = stopping_criterium(integrand, abs_tol=abs_tol, **kwargs)

    # get accumulate_data
    try:
        stopping_crit.data = MLQMCData(stopping_crit, stopping_crit.integrand, stopping_
↪crit.true_measure, stopping_crit.discrete_distrib, stopping_crit.levels_min, stopping_
↪crit.levels_max, stopping_crit.n_init, stopping_crit.replications)
    except:
        stopping_crit.data = MLMCData(stopping_crit, stopping_crit.integrand, stopping_
↪crit.true_measure, stopping_crit.discrete_distrib, stopping_crit.levels_min, stopping_
↪crit.n_init, -1., -1., -1.)

    # manually call "integrate()"
    tol = []
    n_samp = []
    for t in range(stopping_crit.n_tols):
        stopping_crit.rmse_tol = stopping_crit.tol_mult**(stopping_crit.n_tols-t-
↪1)*stopping_crit.target_tol # update tol
        stopping_crit._integrate() # call _integrate()
        tol.append(copy.copy(stopping_crit.rmse_tol))
        n_samp.append(copy.copy(stopping_crit.data.n_level))

        if verbose:
            print("tol = {:5.3e}, number of samples = {}".format(tol[-1], n_samp[-1]))
```

(continues on next page)

(continued from previous page)

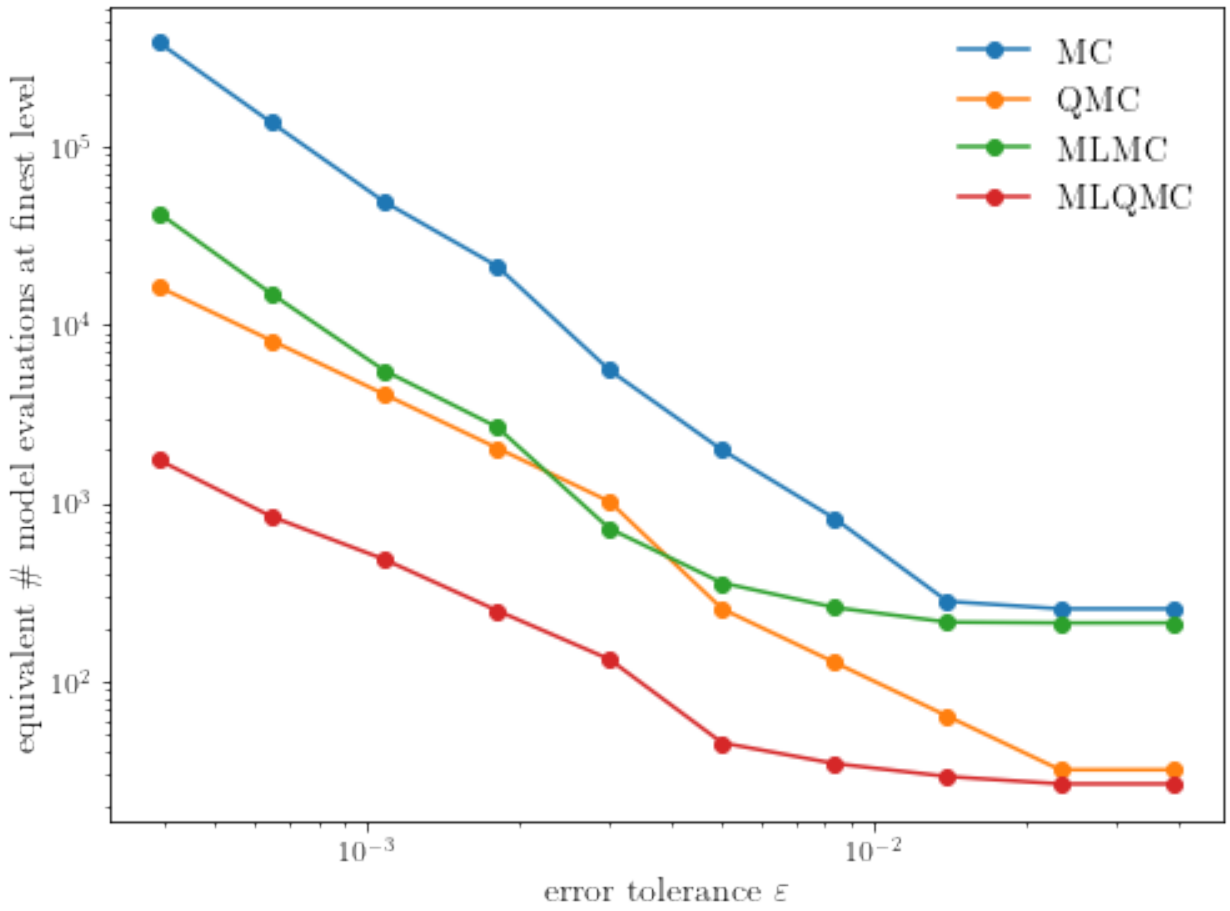
```
    return tol, n_samp
```

```
# Execute the convergence test
def execute_convergence_test(smoothness=1, lengthscale=1):

    # Convergence test for MLMC
    tol_mlmc, n_samp_mlmc = test_convergence(MLEllipticPDE, qp.IIDStdUniform(32), qp.
↪CubMCMLCont, verbose=False)

    # Convergence test for MLQMC
    tol_mlqmc, n_samp_mlqmc = test_convergence(MLEllipticPDE, qp.Lattice(32), qp.
↪CubQMCMLCont, verbose=False, n_init=32)

    # Compute cost per level
    max_levels = max(max([len(n_samp) for n_samp in n_samp_mlmc]), max([len(n_samp) for␣
↪n_samp in n_samp_mlqmc]))
    cost_per_level = np.array([2**level + int(2**(level-1)) for level in range(max_
↪levels)])
    cost_per_level = cost_per_level/cost_per_level[-1]

    # Compute total cost for each tolerance and store the result
    cost = {}
    cost["mc"] = (tol_mlmc, [n_samp_mlmc[tol][0] for tol in range(len(tol_mlmc))]) #␣
↪where we assume V[Q_0] = V[Q_L]
    cost["qmc"] = (tol_mlqmc, [n_samp_mlqmc[tol][0] for tol in range(len(tol_mlqmc))]) #␣
↪where we assume V[Q_0] = V[Q_L]
    cost["mlmc"] = (tol_mlmc, [sum([n_samp*cost_per_level[j] for j, n_samp in␣
↪enumerate(n_samp_mlmc[tol])]) for tol in range(len(tol_mlmc))])
    cost["mlqmc"] = (tol_mlqmc, [sum([n_samp*cost_per_level[j] for j, n_samp in␣
↪enumerate(n_samp_mlqmc[tol])]) for tol in range(len(tol_mlqmc))])

    return cost
```

```
# Plot the result
def plot_convergence(cost):
    fig, ax = plt.subplots(figsize=(8, 6))
    ax.plot(cost["mc"][0], cost["mc"][1], marker="o", label="MC")
    ax.plot(cost["qmc"][0], cost["qmc"][1], marker="o", label="QMC")
    ax.plot(cost["mlmc"][0], cost["mlmc"][1], marker="o", label="MLMC")
    ax.plot(cost["mlqmc"][0], cost["mlqmc"][1], marker="o", label="MLQMC")
    ax.legend(frameon=False)
    ax.set_xscale("log")
    ax.set_yscale("log")
    ax.set_xlabel(r"error tolerance $\varepsilon$")
    ax.set_ylabel(r"equivalent \# model evaluations at finest level")
    plt.show()
```

This command takes a while to execute (about 1 minute on my laptop):

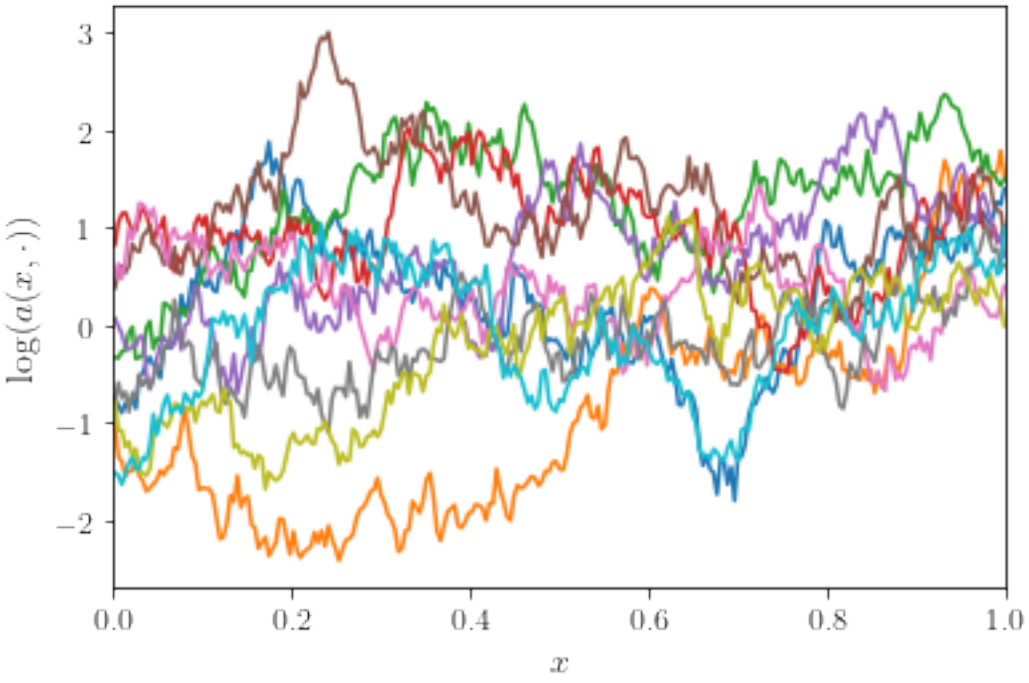
```
plot_convergence(execute_convergence_test())
```



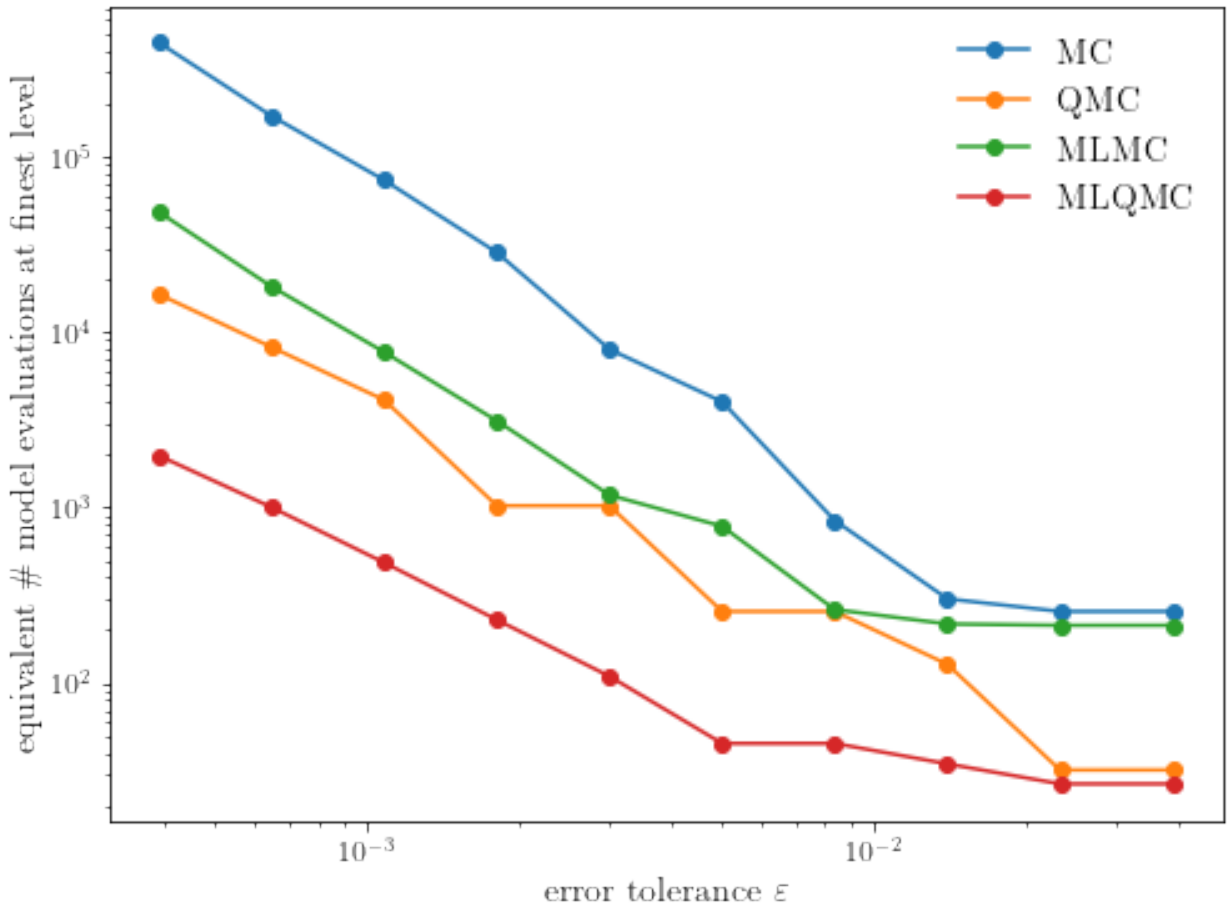
The benefit of the low-discrepancy point set depends on the smoothness of the random field: the smoother the random field, the better. Here's an example for a Gaussian random field with a smaller smoothness $\nu = 1/2$ and smaller length scale $\lambda = 1/3$.

```
smoothness = 1/2
lengthscale = 1/3
```

```
n = 256
x, w, v = get_eigenpairs(n, smoothness=smoothness, lengthscale=lengthscale)
fig, ax = plt.subplots(figsize=(6, 4))
for _ in range(10):
    ax.plot(x, evaluate(w, v))
ax.set_xlim(0, 1)
ax.set_xlabel(r"$x$")
ax.set_ylabel(r"$\log(a(x, \cdot))$")
plt.show()
```




```
plot_convergence(execute_convergence_test(lengthscale=lengthscale,␣
↪smoothness=smoothness))
```



While the multilevel quasi-Monte Carlo method is still the fastest method, the asymptotic cost complexity of the QMC-based methods reduces to approximately the same rate as the MC-based methods.

The benefits of the multilevel methods over single-level methods will be even larger for two- or three-dimensional PDE problems, since it will be even more computationally efficient to take samples on a coarse grid.

5.18 Gaussian Diagnostics

Experiments to demonstate Guassian assumption used in `cubBayesLattice`

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import fmin as fminsearch
from numpy import prod, sin, cos, pi
from scipy.stats import norm as gaussnorm
from matplotlib import cm
```

```
from qmcpy.integrand import Keister
from qmcpy.discrete_distribution.lattice import Lattice
```

```
# print(plt.style.available)
# plt.style.use('./presentation.mplstyle')  # use custom settings
```

(continues on next page)

(continued from previous page)

```
plt.style.use('seaborn-poster')

# plt.rcParams.update({'font.size': 12})
# plt.rcParams.update({'lines.linewidth': 2})
# plt.rcParams.update({'lines.markersize': 6})
```

Let us define the objective function. (`cubBayesLattice`) finds optimal parameters by minimizing the objective function

```
def ObjectiveFunction(theta, order, xun, ftilde):
    tol = 100 * np.finfo(float).eps
    n = len(ftilde)
    arbMean = True
    Lambda = kernel2(theta, order, xun)

    # compute RKHSnorm
    # temp = abs(ftilde(Lambda  != 0).^ 2)./ (Lambda(Lambda != 0))
    temp = abs(ftilde[Lambda > tol] ** 2) / (Lambda[Lambda > tol])

    # compute loss: MLE
    if arbMean == True:
        RKHSnorm = sum(temp[1:]) / n
        temp_1 = sum(temp[1:])
    else:
        RKHSnorm = sum(temp) / n
        temp_1 = sum(temp)

    # ignore all zero eigenvalues
    loss1 = sum(np.log(Lambda[Lambda > tol])) / n
    loss2 = np.log(temp_1)
    loss = (loss1 + loss2)
    if np.imag(loss) != 0:
        # keyboard
        raise('error ! : loss value is complex')

    # print('L1 %1.3f L2 %1.3f L %1.3f r %1.3e theta %1.3e\n'.format(loss1, loss2, loss,␣
↪order, theta))
    return loss, Lambda, RKHSnorm
```

Series approximation of the shift invariant kernel

```
def kernel2(theta, r, xun):
    n = xun.shape[0]
    m = np.arange(1, (n / 2))
    tilde_g_h1 = m ** (-r)
    tilde_g = np.hstack([0, tilde_g_h1, 0, tilde_g_h1[::-1]])
    g = np.fft.fft(tilde_g)
    temp_ = (theta / 2) * g[(xun * n).astype(int)]
    C1 = prod(1 + temp_, 1)
    # eigenvalues must be real : Symmetric pos definite Kernel
    vlambda = np.real(np.fft.fft(C1))
    return vlambda
```

Gaussian random function

```
def f_rand(xpts, rfun, a, b, c, seed):
    dim = xpts.shape[1]
    np.random.seed(seed)  # initialize random number generator for reproducability
    N1 = int(2 ** np.floor(16 / dim))
    Nall = N1 ** dim
    kvec = np.zeros([dim, Nall])  # initialize kvec
    kvec[0, 0:N1] = range(0, N1)  # first dimension
    Nd = N1
    for d in range(1, dim):
        Ndm1 = Nd
        Nd = Nd * N1
        kvec[0:d+1, 0:Nd] = np.vstack([
            np.tile(kvec[0:d, 0:Ndm1], (1, N1)),
            np.reshape(np.tile(np.arange(0, N1), (Ndm1, 1)), (1, Nd), order="F")
        ])

    kvec = kvec[:, 1: Nall]  # remove the zero wavenumber
    whZero = np.sum(kvec == 0, axis=0)
    abfac = a ** (dim - whZero) * b ** whZero
    kbar = np.prod(np.maximum(kvec, 1), axis=0)
    totfac = abfac / (kbar ** rfun)

    f_c = a * np.random.randn(1, Nall - 1) * totfac
    f_s = a * np.random.randn(1, Nall - 1) * totfac

    f_0 = c + (b ** dim) * np.random.randn()
    argx = np.matmul((2 * np.pi * xpts), kvec)
    f_c_ = f_c * np.cos(argx)
    f_s_ = f_s * np.sin(argx)
    fval = f_0 + np.sum(f_c_ + f_s_, axis=1)
    return fval
```

Periodization transforms

```
def doPeriodTx(x, integrand, ptransform):
    ptransform = ptransform.upper()
    if ptransform == 'BAKER':  # Baker's transform
        xp = 1 - 2 * abs(x - 1 / 2)
        w = 1
    elif ptransform == 'C0':  # C^0 transform
        xp = 3 * x ** 2 - 2 * x ** 3
        w = prod(6 * x * (1 - x), 1)
    elif ptransform == 'C1':  # C^1 transform
        xp = x ** 3 * (10 - 15 * x + 6 * x ** 2)
        w = prod(30 * x ** 2 * (1 - x) ** 2, 1)
    elif ptransform == 'C1SIN':  # Sidi C^1 transform
        xp = x - sin(2 * pi * x) / (2 * pi)
        w = prod(2 * sin(pi * x) ** 2, 1)
    elif ptransform == 'C2SIN':  # Sidi C^2 transform
        xp = (8 - 9 * cos(pi * x) + cos(3 * pi * x)) / 16  # psi3
        w = prod((9 * sin(pi * x) * pi - sin(3 * pi * x) * 3 * pi) / 16, 1)  # psi3_1
    elif ptransform == 'C3SIN':  # Sidi C^3 transform
```

(continues on next page)

(continued from previous page)

```
        xp = (12 * pi * x - 8 * sin(2 * pi * x) + sin(4 * pi * x)) / (12 * pi)  # psi4
        w = prod((12 * pi - 8 * cos(2 * pi * x) * 2 * pi + sin(4 * pi * x) * 4 * pi) /␣
↪(12 * pi), 1)  # psi4_1
    elif ptransform == 'NONE':
        xp = x
        w = 1
    else:
        raise (f"The {ptransform} periodization transform is not implemented")
    y = integrand(xp) * w
    return y
```

Utility function to draw qqplot or normplot

```
def create_quant_plot(type, vz_real, fName, dim, iii, r, rOpt, theta, thetaOpt):
    hFigNormplot, axFigNormplot = plt.subplots()

    n = len(vz_real)
    if type == 'normplot':
        axFigNormplot.normplot(vz_real)
    else:
        q = (np.arange(1, n + 1) - 1 / 2) / n
        stNorm = gaussnorm.ppf(q)  # norminv: quantiles of standard normal
        axFigNormplot.scatter(stNorm, sorted(vz_real), s=20)  # marker='.',
        axFigNormplot.plot([-3, 3], [-3, 3], marker='_', linewidth=4, color='red')
        axFigNormplot.set_xlabel('Standard Gaussian Quantiles')
        axFigNormplot.set_ylabel('Data Quantiles')

    if theta:
        plt_title = f'$d={dim}, n={n}, r={r:1.2f}, r_{{opt}}={rOpt:1.2f}, \\theta=
↪{theta:1.2f}, \\theta_{{opt}}={thetaOpt:1.2f}$'
        plt_filename = f'{fName}-QQPlot_n-{n}_d-{dim}_r-{r * 100}_th-{100 * theta}_case-
↪{iii}.jpg'
    else:
        plt_title = f'$d={dim}, n={n}, r_{{opt}}={rOpt:1.2f}, \\theta_{{opt}}=
↪{thetaOpt:1.2f}$'
        plt_filename = f'{fName}-QQPlot_n-{n}_d-{dim}_case-{iii}.jpg'

    axFigNormplot.set_title(plt_title)
    hFigNormplot.savefig(plt_filename)
```

Utility function to plot the objective function and minimum

```
def create_surf_plot(fName, lnthth, lnordord, objfun, objobj, lnParamsOpt, r, theta,␣
↪iii):
    figH, axH = plt.subplots(subplot_kw={"projection": "3d"})
    axH.view_init(40, 30)
    shandle = axH.plot_surface(lnthth, lnordord, objobj, cmap=cm.coolwarm,
                               linewidth=0, antialiased=False, alpha=0.8)
    xt = np.array([.2, 0.4, 1, 3, 7])
    axH.set_xticks(np.log(xt))
    axH.set_xticklabels(xt.astype(str))
    yt = np.array([1.4, 1.6, 2, 2.6, 3.7])
```

(continues on next page)

(continued from previous page)

```
    axH.set_yticks(np.log(yt - 1))
    axH.set_yticklabels(yt.astype(str))
    axH.set_xlabel('$\\theta$')
    axH.set_ylabel('$r$')

    axH.scatter(lnParamsOpt[0], lnParamsOpt[1], objfun(lnParamsOpt) * 1.002,
                s=200, color='orange', marker='*', alpha=0.8)
    if theta:
        filename = f'{fName}-ObjFun_n-{npts}_d-{dim}_r-{r * 100}_th-{100 * theta}_case-
↪{iii}.jpg'
    else:
        filename = f'{fName}-ObjFun_n-{npts}_d-{dim}_case-{iii}.jpg'
    figH.savefig(filename)
```

Minimum working example to demonstrate Gaussian diagnostics concept

```
def gaussian_diagnostics_engine(whEx, dim, npts, r, fpar, nReps, nPlots):
    whEx = whEx - 1
    fNames = ['ExpCos', 'Keister', 'rand']
    ptransforms = ['none', 'C1sin', 'none']
    fName = fNames[whEx]
    ptransform = ptransforms[whEx]

    rOptAll = [0]*nRep
    thOptAll = [0]*nRep

    # parameters for random function
    # seed = 202326
    if whEx == 2:
        rfun = r / 2
        f_mean = fpar[2]
        f_std_a = fpar[0]  # this is square root of the a in the talk
        f_std_b = fpar[1]  # this is square root of the b in the talk
        theta = (f_std_a / f_std_b) ** 2
    else:
        theta = None

    for iii in range(nReps):
        seed = np.random.randint(low=1, high=1e6)  # different each rep
        shift = np.random.rand(1, dim)

        distribution = Lattice(dimension=dim, order='linear')
        xpts, xlat = distribution.gen_samples(n_min=0, n_max=npts, warn=False, return_
↪unrandomized=True)

        if fName == 'ExpCos':
            integrand = lambda x: np.exp(np.sum(np.cos(2 * np.pi * x), axis=1))
        elif fName == 'Keister':
            keister = Keister(Lattice(dimension=dim, order='linear'))
            integrand = lambda x: keister.f(x)
        elif fName == 'rand':
            integrand = lambda x: f_rand(x, rfun, f_std_a, f_std_b, f_mean, seed)
```

(continues on next page)

(continued from previous page)

```
        else:
            print('Invalid function name')
            return

        y = doPeriodTx(xpts, integrand, ptransform)

        ftilde = np.fft.fft(y)  # fourier coefficients
        ftilde[0] = 0  # ftilde = \mV**H(\vf - m \vone), subtract mean
        if dim == 1:
            hFigIntegrand = plt.figure()
            plt.scatter(xpts, y, 10)
            plt.title(f'{fName}_n-{npts}_Tx-{ptransform}')
            hFigIntegrand.savefig(f'{fName}_n-{npts}_Tx-{ptransform}_rFun-{rfun:1.2f}.png
↪')

        def objfun(lnParams):
            loss, Lambda, RKHSnorm = ObjectiveFunction(np.exp(lnParams[0]), 1 + np.
↪exp(lnParams[1]), xlat, ftilde)
            return loss

        ## Plot the objective function
        lnthetarange = np.arange(-2, 2.2, 0.2)  # range of log(theta) for plotting
        lnorderrange = np.arange(-1, 1.1, 0.1)  # range of log(r) for plotting
        [lnthth, lnordord] = np.meshgrid(lnthetarange, lnorderrange)
        objobj = np.zeros(lnthth.shape)
        for ii in range(lnthth.shape[0]):
            for jj in range(lnthth.shape[1]):
                objobj[ii, jj] = objfun([lnthth[ii, jj], lnordord[ii, jj]])

        objMinAppx, which = objobj.min(), objobj.argmin()
        # [whichrow, whichcol] = ind2sub(lnthth.shape, which)
        [whichrow, whichcol] = np.unravel_index(which, lnthth.shape)
        lnthOptAppx = lnthth[whichrow, whichcol]
        thetaOptAppx = np.exp(lnthOptAppx)
        lnordOptAppx = lnordord[whichrow, whichcol]
        orderOptAppx = 1 + np.exp(lnordOptAppx)
        # print(objMinAppx)  # minimum objective function by brute force search

        ## Optimize the objective function
        result = fminsearch(objfun, x0=[lnthOptAppx, lnordOptAppx], xtol=1e-3, full_
↪output=True, disp=False)
        lnParamsOpt, objMin = result[0], result[1]
        # print(objMin)  # minimum objective function by Nelder-Mead
        thetaOpt = np.exp(lnParamsOpt[0])
        rOpt = 1 + np.exp(lnParamsOpt[1])
        rOptAll[iii] = rOpt
        thOptAll[iii] = thetaOpt
        print(f'{iii}: thetaOptAppx={thetaOptAppx:7.5f}, rOptAppx={orderOptAppx:7.5f}, '
              f'objMinAppx={objMinAppx:7.5f}, objMin={objMin:7.5f}')

        if iii <= nPlots:
            create_surf_plot(fName, lnthth, lnordord, objfun, objobj, lnParamsOpt, r,␣
↪theta, iii)
```

(continues on next page)

(continued from previous page)

```
        vlambda = kernel2(thetaOpt, rOpt, xlat)
        s2 = sum(abs(ftilde[2:] ** 2) / vlambda[2:]) / (npts ** 2)
        vlambda = s2 * vlambda

        # apply transform
        # $\vZ = \frac 1n \mV \mLambda**{-\frac 12} \mV**H(\vf - m \vone)$
        # np.fft also includes 1/n division
        vz = np.fft.ifft(ftilde / np.sqrt(vlambda))
        vz_real = np.real(vz)  # vz must be real as intended by the transformation

        if iii <= nPlots:
            create_quant_plot('qqplot', vz_real, fName, dim, iii, r, rOpt, theta,␣
↪thetaOpt)

        r_str = f"{r: 7.5f}" if type(r) == float else str(r)
        theta_str = f"{theta: 7.5f}" if type(theta) == float else str(theta)
        print(f'\t r = {r_str}, rOpt = {rOpt:7.5f}, theta = {theta_str}, thetaOpt =
↪{thetaOpt:7.5f}\n')

    return [theta, rOptAll, thOptAll, fName]
```

5.18.1 Example 1: Exponential of Cosine

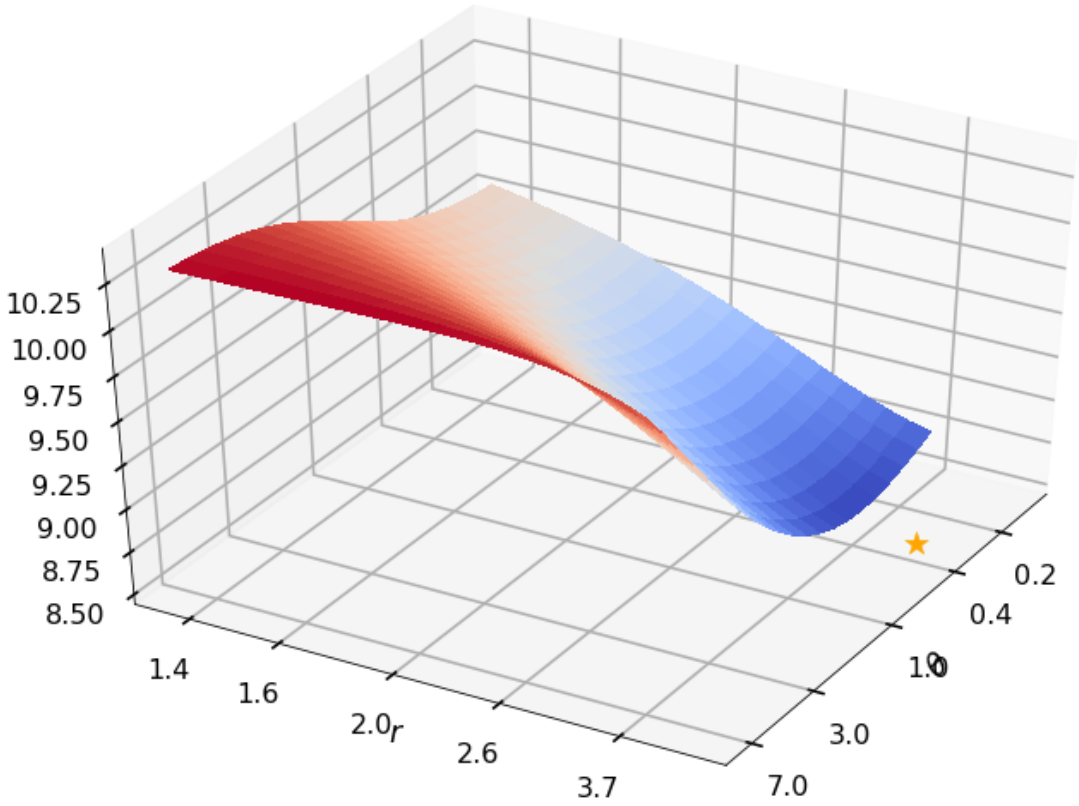
```
fwh = 1
dim = 3
npts = 2 ** 6
nRep = 20
nPlot = 2
[_, rOptAll, thOptAll, fName] = \
    gaussian_diagnostics_engine(fwh, dim, npts, None, None, nRep, nPlot)

## Plot Exponential Cosine example
figH = plt.figure()
plt.scatter(rOptAll, thOptAll, s=20, color='blue')
# axis([4 6 0.1 10])
# set(gca,'yscale','log')
plt.title(f'$d = {dim}, n = {npts}$')
plt.xlabel('Inferred $r$')
plt.ylabel('Inferred $\\theta$')
# print(f'{fName}-rthInfer-n-{npts}-d-{dim}')
figH.savefig(f'{fName}-rthInfer-n-{npts}-d-{dim}.jpg')
```

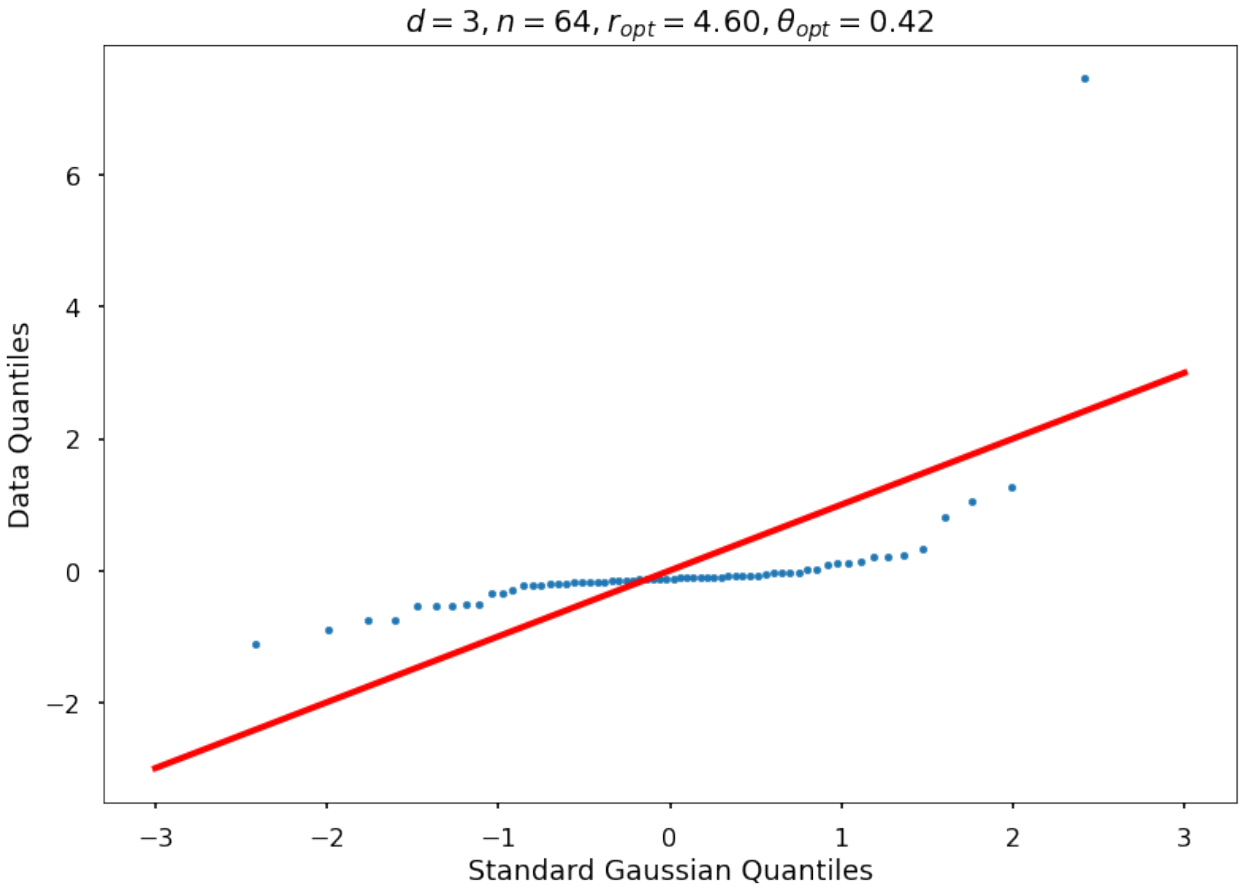
```
0: thetaOptAppx=0.36788, rOptAppx=3.71828, objMinAppx=8.62452, objMin=8.55148
     r = None, rOpt = 4.60174, theta = None, thetaOpt = 0.41845

1: thetaOptAppx=0.36788, rOptAppx=3.71828, objMinAppx=8.45167, objMin=8.30125
     r = None, rOpt = 5.04805, theta = None, thetaOpt = 0.41943

2: thetaOptAppx=0.44933, rOptAppx=3.71828, objMinAppx=8.94756, objMin=8.86498
```

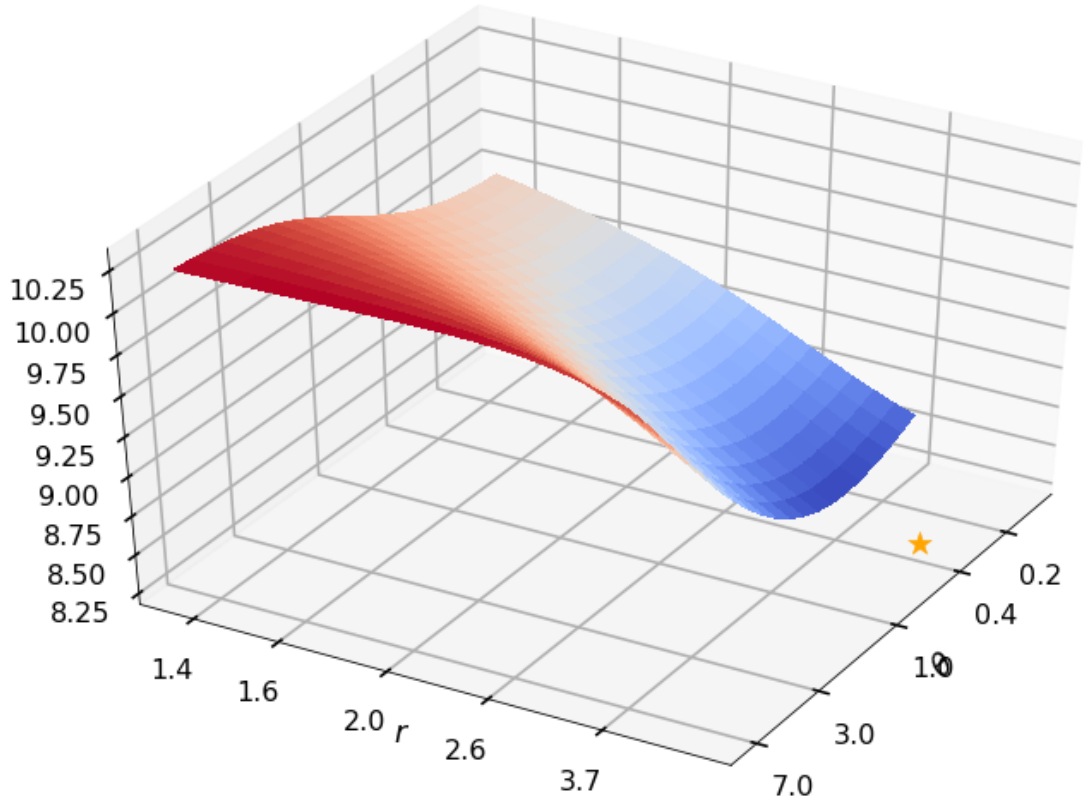
(continues on next page)

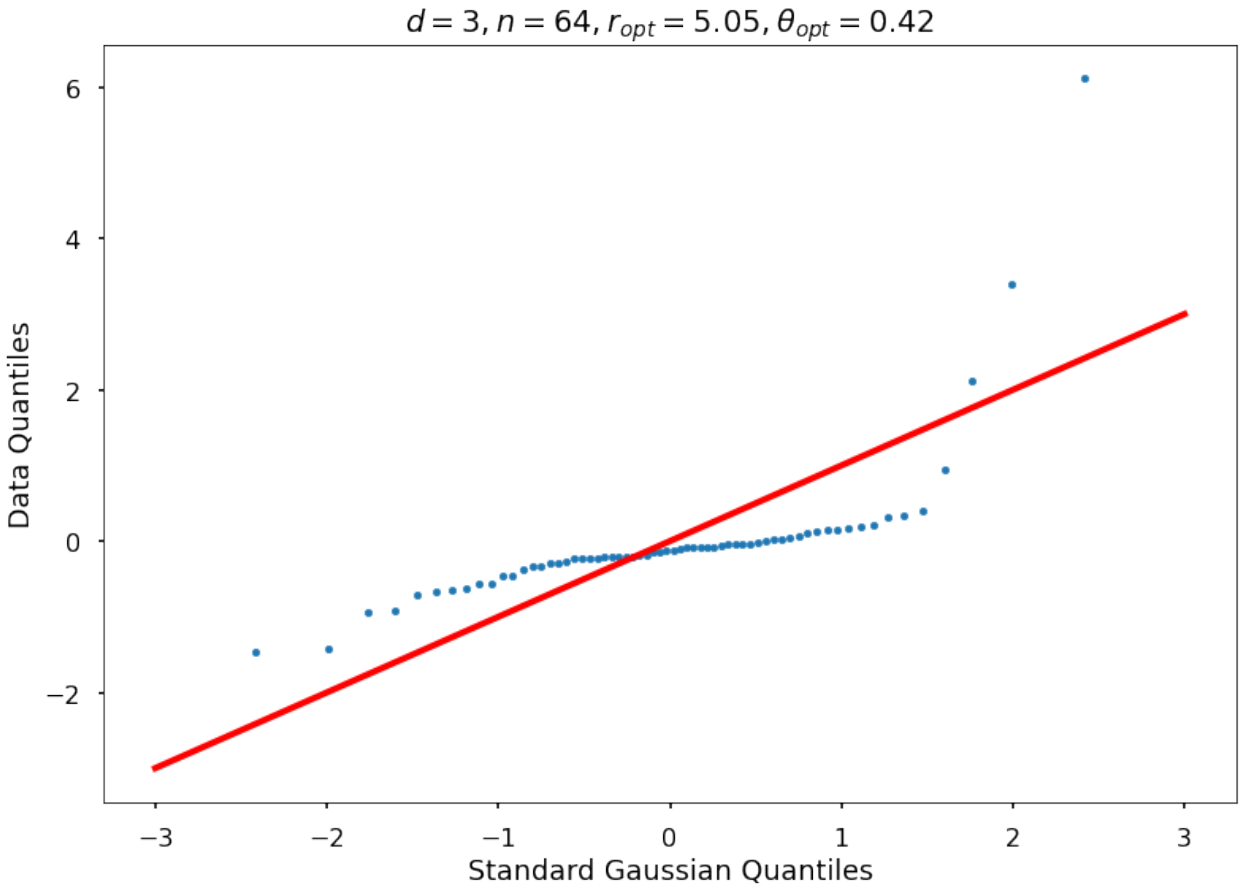
(continued from previous page)

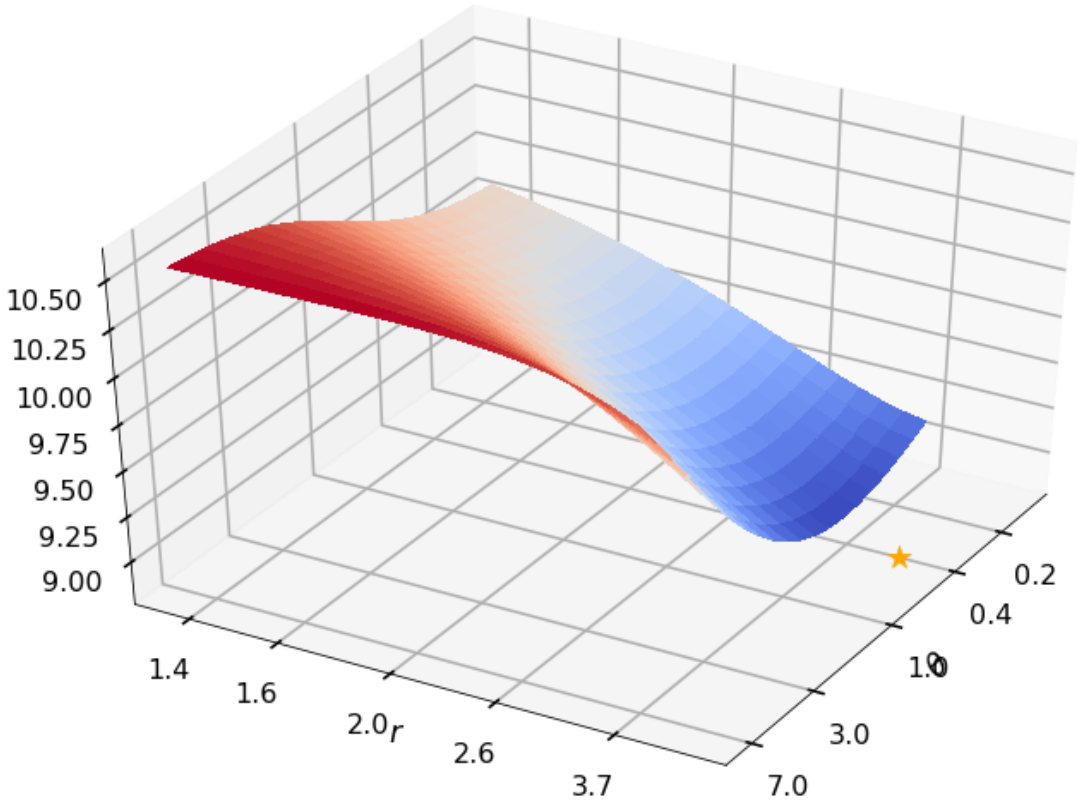
```
     r = None, rOpt = 4.70102, theta = None, thetaOpt = 0.53527

3: thetaOptAppx=0.36788, rOptAppx=3.71828, objMinAppx=8.44618, objMin=8.29799
     r = None, rOpt = 5.04699, theta = None, thetaOpt = 0.39835

4: thetaOptAppx=0.36788, rOptAppx=3.71828, objMinAppx=8.45494, objMin=8.28315
     r = None, rOpt = 5.15814, theta = None, thetaOpt = 0.40447

5: thetaOptAppx=0.36788, rOptAppx=3.71828, objMinAppx=8.57968, objMin=8.47859
     r = None, rOpt = 4.75680, theta = None, thetaOpt = 0.38761

6: thetaOptAppx=0.36788, rOptAppx=3.71828, objMinAppx=8.69501, objMin=8.61827
     r = None, rOpt = 4.63978, theta = None, thetaOpt = 0.44889

7: thetaOptAppx=0.44933, rOptAppx=3.71828, objMinAppx=9.00638, objMin=8.91494
     r = None, rOpt = 4.78933, theta = None, thetaOpt = 0.54575

8: thetaOptAppx=0.44933, rOptAppx=3.71828, objMinAppx=9.01994, objMin=8.90784
     r = None, rOpt = 4.97296, theta = None, thetaOpt = 0.55376

9: thetaOptAppx=0.44933, rOptAppx=3.71828, objMinAppx=8.79716, objMin=8.67140
     r = None, rOpt = 4.96866, theta = None, thetaOpt = 0.50381

10: thetaOptAppx=0.36788, rOptAppx=3.71828, objMinAppx=8.79790, objMin=8.65545
     r = None, rOpt = 5.13328, theta = None, thetaOpt = 0.47860

11: thetaOptAppx=0.36788, rOptAppx=3.71828, objMinAppx=8.47457, objMin=8.34622
     r = None, rOpt = 4.90876, theta = None, thetaOpt = 0.42187

12: thetaOptAppx=0.36788, rOptAppx=3.71828, objMinAppx=8.49223, objMin=8.39115
     r = None, rOpt = 4.75830, theta = None, thetaOpt = 0.39949

13: thetaOptAppx=0.36788, rOptAppx=3.71828, objMinAppx=8.61363, objMin=8.51504
     r = None, rOpt = 4.77039, theta = None, thetaOpt = 0.44782

14: thetaOptAppx=0.30119, rOptAppx=3.71828, objMinAppx=8.48461, objMin=8.34370
     r = None, rOpt = 5.03099, theta = None, thetaOpt = 0.35658

15: thetaOptAppx=0.44933, rOptAppx=3.71828, objMinAppx=9.03541, objMin=8.96192
     r = None, rOpt = 4.65646, theta = None, thetaOpt = 0.56225

16: thetaOptAppx=0.30119, rOptAppx=3.71828, objMinAppx=8.24322, objMin=8.08881
     r = None, rOpt = 4.96519, theta = None, thetaOpt = 0.28016

17: thetaOptAppx=0.36788, rOptAppx=3.71828, objMinAppx=8.53428, objMin=8.37148
     r = None, rOpt = 5.12928, theta = None, thetaOpt = 0.43230

18: thetaOptAppx=0.44933, rOptAppx=3.71828, objMinAppx=8.95320, objMin=8.88930
     r = None, rOpt = 4.57031, theta = None, thetaOpt = 0.53913

19: thetaOptAppx=0.30119, rOptAppx=3.71828, objMinAppx=8.42682, objMin=8.27037
     r = None, rOpt = 5.01433, theta = None, thetaOpt = 0.39608
```

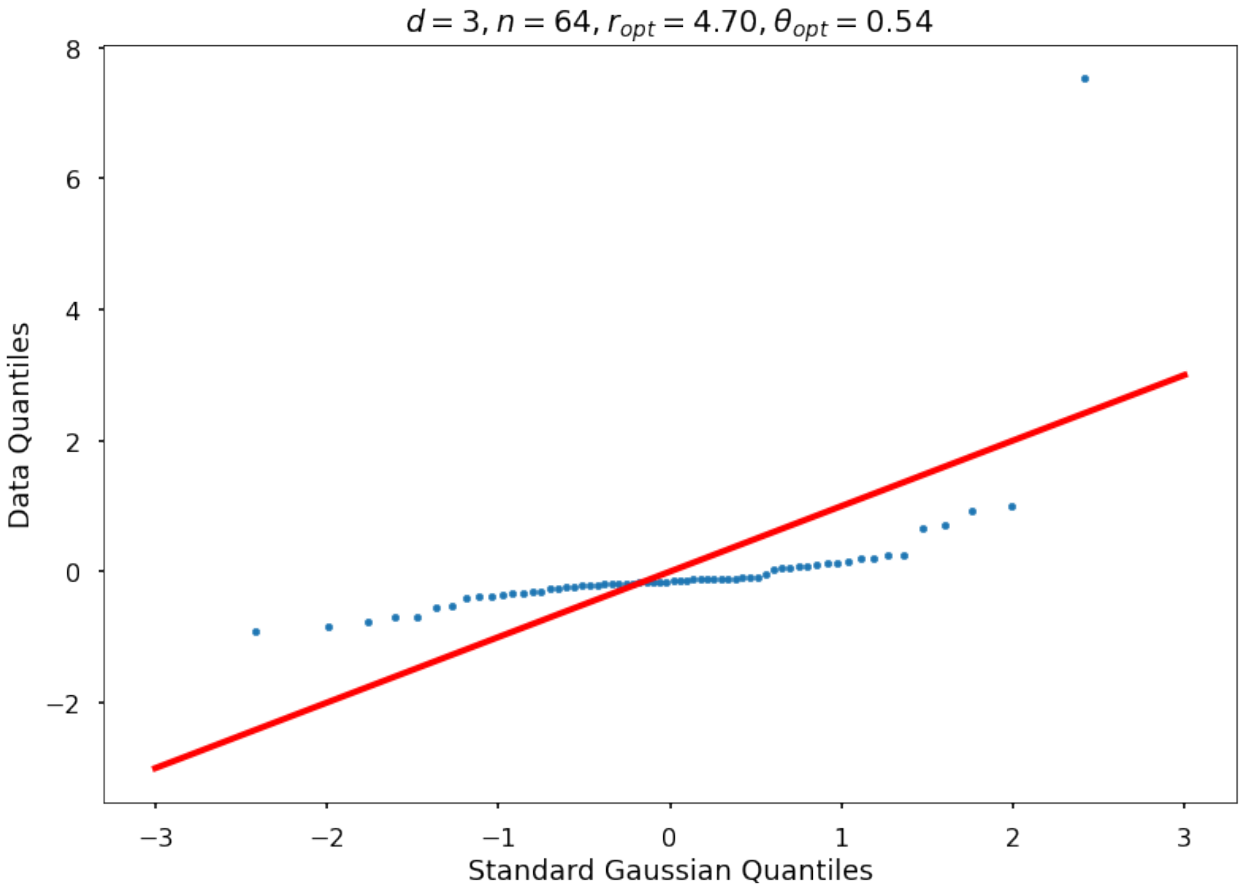

10.25
10.00
9.75
9.50
9.25
9.00
8.75
8.50
1.4
1.6
2.0
r
2.6
3.7
0.2
0.4
3.0
7.0

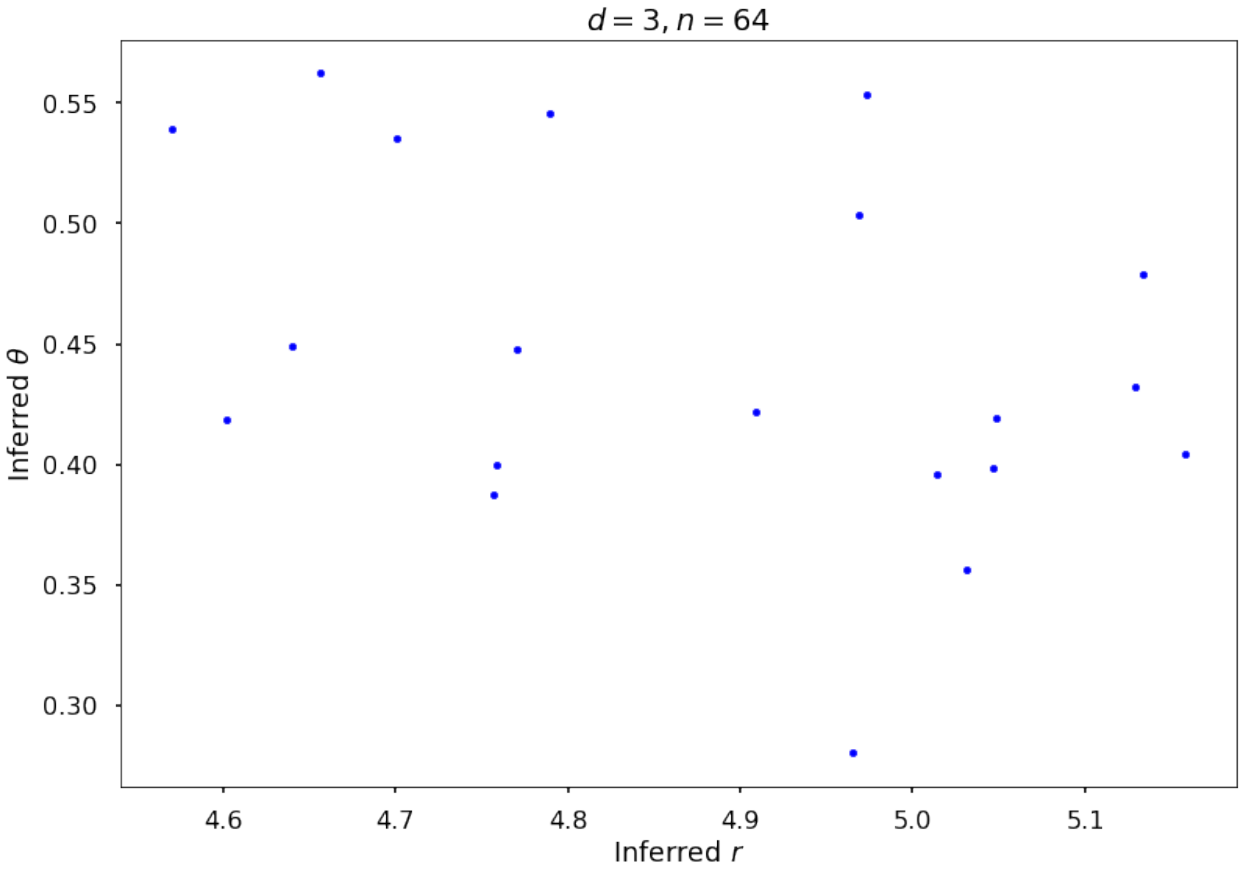

$d = 3, n = 64, r_{opt} = 4.60, \theta_{opt} = 0.42$
Data Quantiles
Standard Gaussian Quantiles
6
4
2
0
−2
−3
−2
−1
0
1
2
3


10.25
10.00
9.75
9.50
9.25
9.00
8.75
8.50
8.25
1.4
1.6
2.0
r
2.6
3.7
0.2
0.4
3.0
7.0


$d = 3, n = 64, r_{opt} = 5.05, \theta_{opt} = 0.42$
Data Quantiles
Standard Gaussian Quantiles
6
4
2
0
−2
−3
−2
−1
0
1
2
3


10.50
10.25
10.00
9.75
9.50
9.25
9.00
1.4
1.6
2.0
r
2.6
3.7
0.2
0.4
3.0
7.0


$d = 3, n = 64, r_{opt} = 4.70, \theta_{opt} = 0.54$
Data Quantiles
Standard Gaussian Quantiles
8
6
4
2
0
−2
−3
−2
−1
0
1
2
3




```
# close all the previous plots to freeup memory
plt.close('all')
```

5.18.2 Example 2: Random function

```
## Tests with random function
rArray = [1.5, 2, 4]
nrArr = len(rArray)
fParArray = [[0.5, 1, 2], [1, 1, 1], [1, 1, 1]]
nfPArr = len(fParArray)
fwh = 3
dim = 2
npts = 2 ** 6
nRep = 5  # reduced from 20 to reduce the plots
nPlot = 2
thetaAll = np.zeros((nrArr, nfPArr))
rOptAll = np.zeros((nrArr, nfPArr, nRep))
thOptAll = np.zeros((nrArr, nfPArr, nRep))
for jjj in range(nrArr):
    for kkk in range(nfPArr):
        thetaAll[jjj, kkk], rOptAll[jjj, kkk, :], thOptAll[jjj, kkk, :], fName = \
            gaussian_diagnostics_engine(fwh, dim, npts, rArray[jjj], fParArray[kkk],␣
↪nRep, nPlot)
```

```
7.287181247404372
7.268099510403598
r =  1.50000, rOpt = 1.08321, theta =  0.25000, thetaOpt = 0.18959

7.439206667642468
7.420522178757865
r =  1.50000, rOpt = 1.08048, theta =  0.25000, thetaOpt = 0.37974

7.237565543945999
7.2348776269121124
r =  1.50000, rOpt = 1.24698, theta =  0.25000, thetaOpt = 1.00000

7.144410538581832
7.143842961246502
r =  1.50000, rOpt = 1.75370, theta =  0.25000, thetaOpt = 2.48385

7.317407633559816
7.317303232422838
r =  1.50000, rOpt = 1.55240, theta =  0.25000, thetaOpt = 0.64486

10.864589626523202
10.864398926256648
r =  1.50000, rOpt = 1.38658, theta =  1.00000, thetaOpt = 6.87095

10.506347397179722
10.483539417962092
r =  1.50000, rOpt = 1.00000, theta =  1.00000, thetaOpt = 1.32199

10.732206996328214
10.7195206863197
r =  1.50000, rOpt = 1.03409, theta =  1.00000, thetaOpt = 0.96591

10.713950538487756
10.706754036079342
r =  1.50000, rOpt = 1.70351, theta =  1.00000, thetaOpt = 12.89957

10.3923460325491
10.367674586632736
r =  1.50000, rOpt = 1.00000, theta =  1.00000, thetaOpt = 0.60214

10.481613959193172
10.481573119044445
r =  1.50000, rOpt = 1.44574, theta =  1.00000, thetaOpt = 1.86278

10.624184268139306
10.587513806429946
r =  1.50000, rOpt = 1.00000, theta =  1.00000, thetaOpt = 0.65292

10.336255790040328
10.32105621339534
r =  1.50000, rOpt = 1.11594, theta =  1.00000, thetaOpt = 1.00000

10.699799283646295
```

(continues on next page)

(continued from previous page)

```
10.69967356700936
r =  1.50000, rOpt = 1.44362, theta =  1.00000, thetaOpt = 2.38455

10.51500018761115
10.506939121521446
r =  1.50000, rOpt = 1.17968, theta =  1.00000, thetaOpt = 8.69595

6.158209638152028
6.157963442419079
r = 2, rOpt = 1.57828, theta =  0.25000, thetaOpt = 0.23715
```

```
c:toolsminiconda3envsqmcpy1libsite-packagesipykernel_launcher.py:66
    RuntimeWarning: More than 20 figures have been opened. Figures created through the␣
↪pyplot interface (matplotlib.pyplot.figure) are retained until explicitly closed and␣
↪may consume too much memory. (To control this warning, see the rcParam figure.max_open_
↪warning).
c:toolsminiconda3envsqmcpy1libsite-packagesipykernel_launcher.py:2
    RuntimeWarning: More than 20 figures have been opened. Figures created through the␣
↪pyplot interface (matplotlib.pyplot.figure) are retained until explicitly closed and␣
↪may consume too much memory. (To control this warning, see the rcParam figure.max_open_
↪warning).
```

```
6.157143147450321
6.156543364291681
r = 2, rOpt = 2.15114, theta =  0.25000, thetaOpt = 2.01164

5.75085055457758
5.750809973133201
r = 2, rOpt = 2.00000, theta =  0.25000, thetaOpt = 0.68772

6.069632328951123
6.069055139303397
r = 2, rOpt = 1.84615, theta =  0.25000, thetaOpt = 0.42398

6.034985224985181
6.034874930815441
r = 2, rOpt = 1.67831, theta =  0.25000, thetaOpt = 0.53403

9.551558949940272
9.551384972239866
r = 2, rOpt = 1.92009, theta =  1.00000, thetaOpt = 1.58611

9.633085074190632
9.632776824536194
r = 2, rOpt = 1.76012, theta =  1.00000, thetaOpt = 1.73525

9.568557894047606
9.568348412892643
r = 2, rOpt = 1.58769, theta =  1.00000, thetaOpt = 0.13924

9.541751585509552
9.541667509495124
```

(continues on next page)

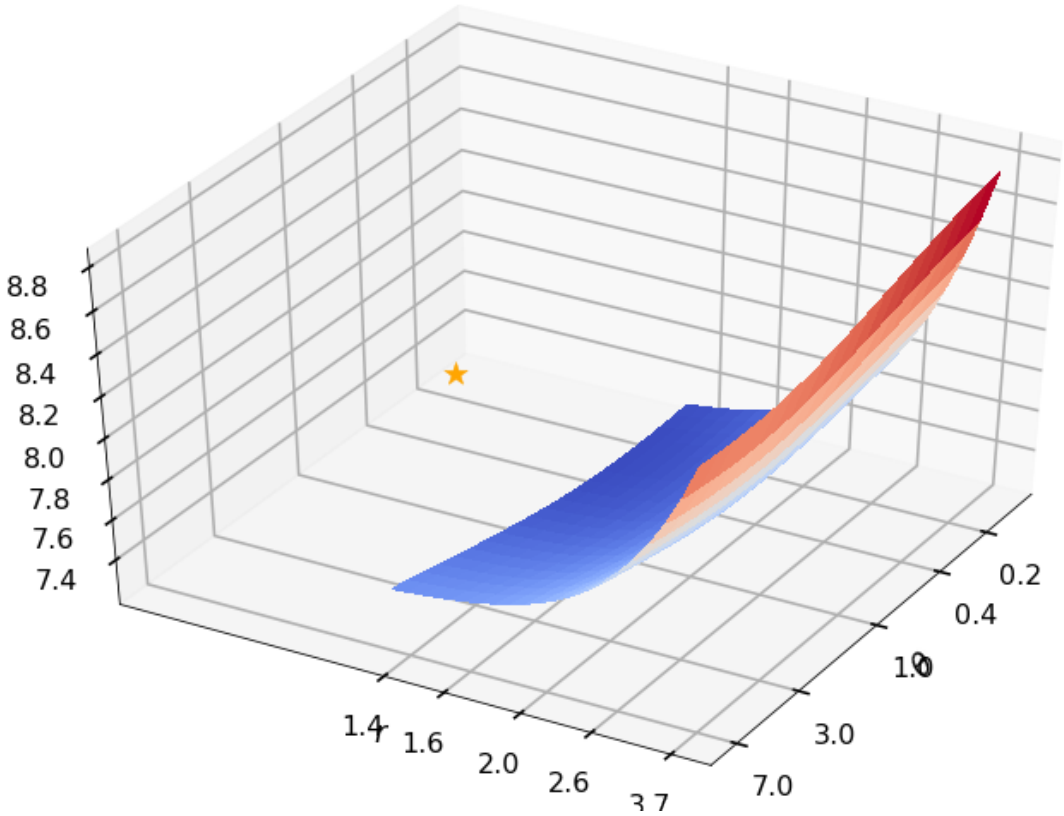
(continued from previous page)

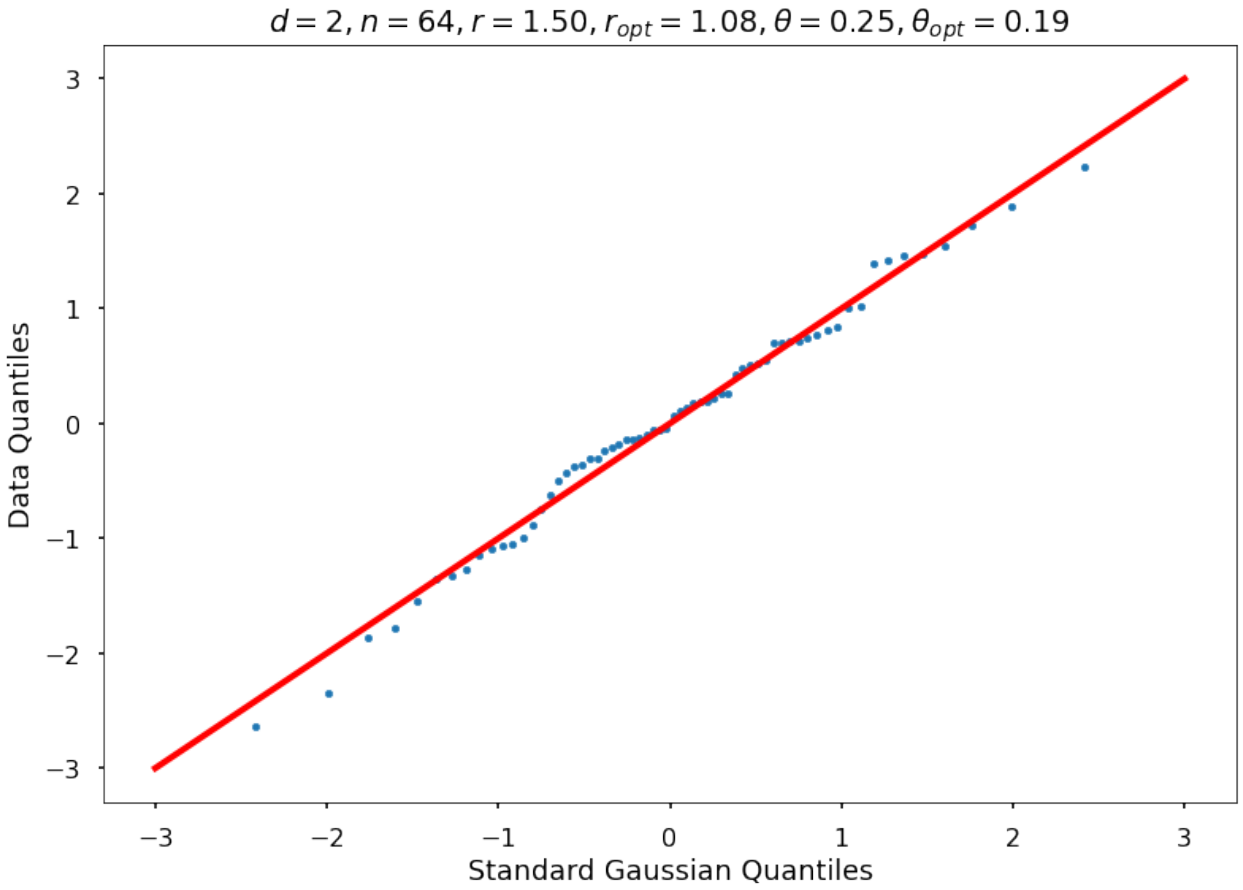
```
r = 2, rOpt = 1.48197, theta =  1.00000, thetaOpt = 1.41333

9.5396175595642
9.539398408753463
r = 2, rOpt = 1.55887, theta =  1.00000, thetaOpt = 0.12623

9.170505000843896
9.162768712949173
r = 2, rOpt = 1.15246, theta =  1.00000, thetaOpt = 1.68521

9.6238915085669
9.62377543126308
r = 2, rOpt = 1.39439, theta =  1.00000, thetaOpt = 0.86724

9.851657112032425
9.850539894156423
r = 2, rOpt = 2.15206, theta =  1.00000, thetaOpt = 3.90645

9.806190014351044
9.805541561888338
r = 2, rOpt = 2.44945, theta =  1.00000, thetaOpt = 2.23506

9.79209159625104
9.791515852224858
r = 2, rOpt = 1.85087, theta =  1.00000, thetaOpt = 1.61353

3.2161538975858517
3.211437887705981
r = 4, rOpt = 3.84952, theta =  0.25000, thetaOpt = 0.56819

3.1312343157485762
3.1292745389997414
r = 4, rOpt = 3.37450, theta =  0.25000, thetaOpt = 0.30053

3.0695614957742343
3.0682627340993567
r = 4, rOpt = 3.39840, theta =  0.25000, thetaOpt = 0.41193

3.027380943889609
3.0266075283943112
r = 4, rOpt = 3.76590, theta =  0.25000, thetaOpt = 0.36430

3.5228968345657194
3.5143715614313487
r = 4, rOpt = 3.89398, theta =  0.25000, thetaOpt = 0.48767

7.379762232781738
7.376813143468294
r = 4, rOpt = 3.60031, theta =  1.00000, thetaOpt = 1.14533

6.641928348506947
6.617895389661783
```

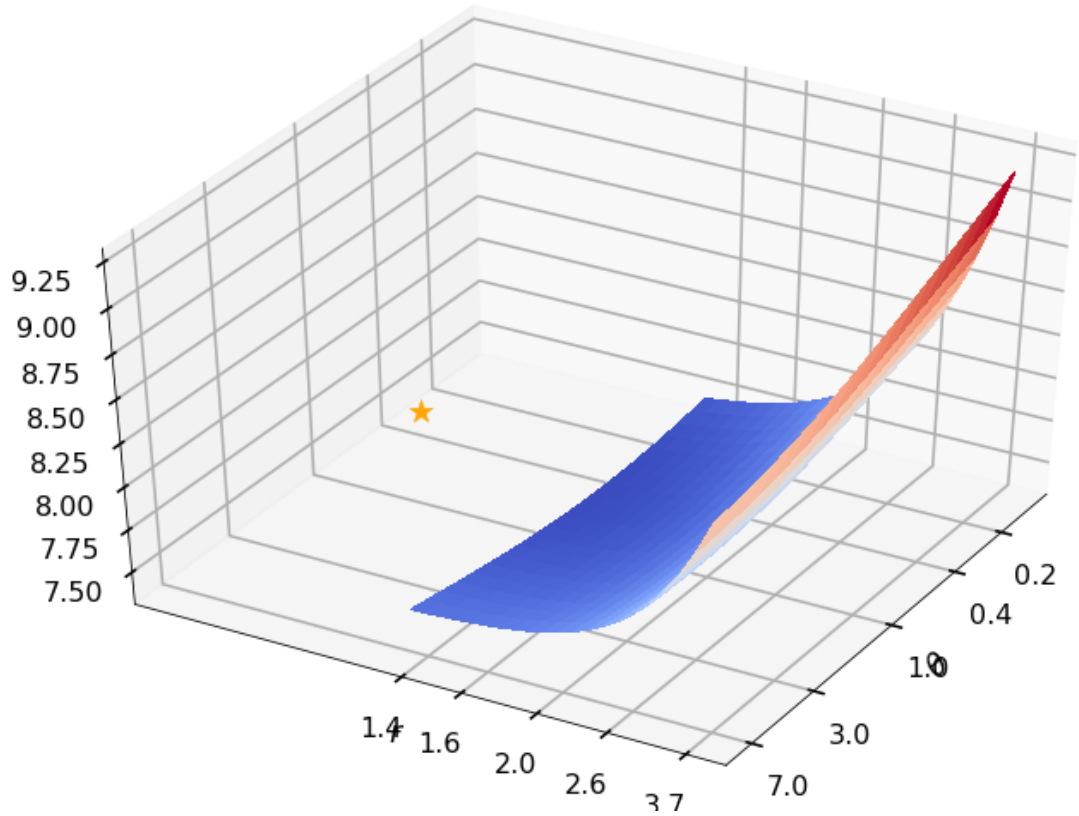
(continues on next page)

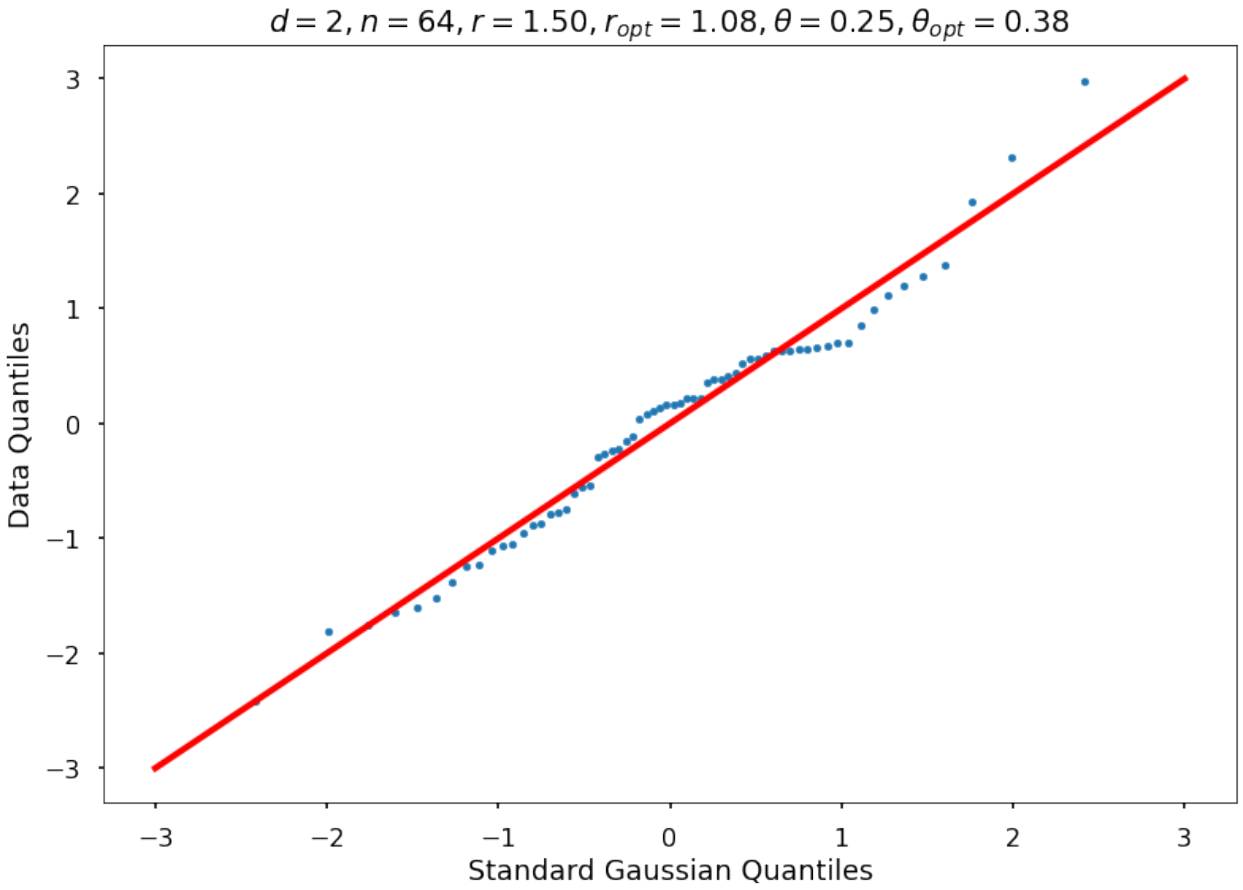
(continued from previous page)

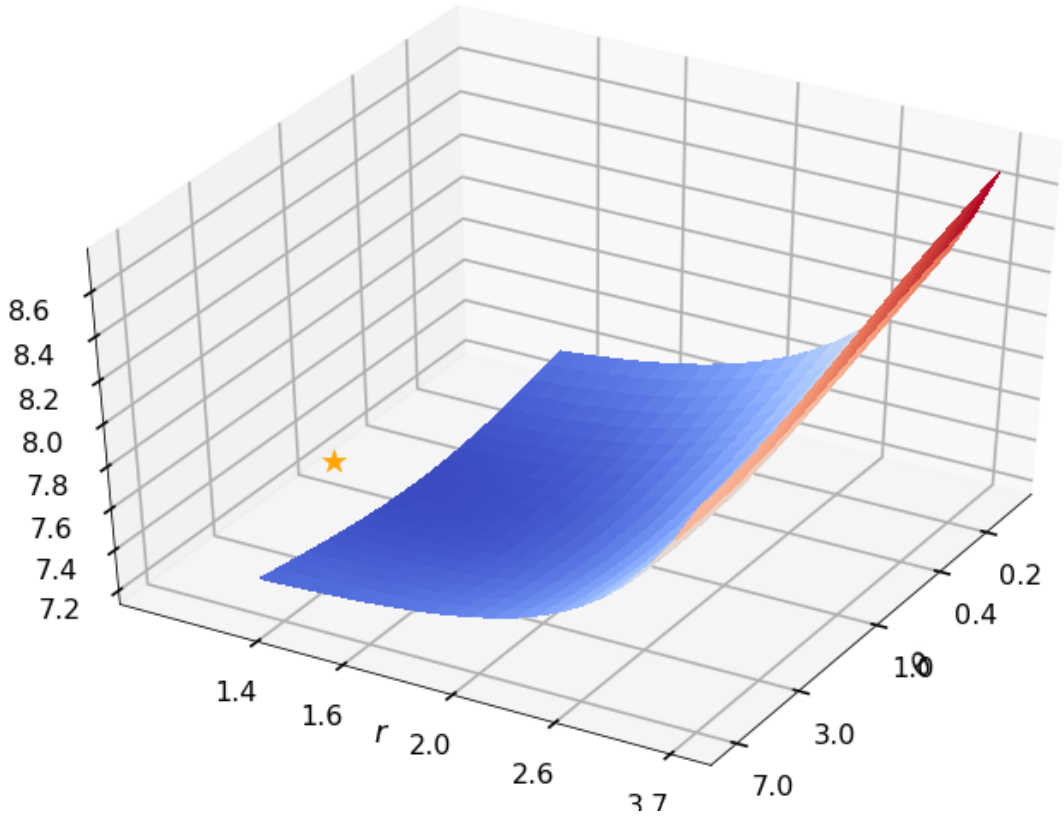
```
r = 4, rOpt = 4.05057, theta =  1.00000, thetaOpt = 4.39747

7.476855960512737
7.476496711605328
r = 4, rOpt = 3.18967, theta =  1.00000, thetaOpt = 1.00000

6.905066745166625
6.731884736747105
r = 4, rOpt = 4.55762, theta =  1.00000, thetaOpt = 2.69082

6.998288530030136
6.956313755828155
r = 4, rOpt = 4.25120, theta =  1.00000, thetaOpt = 1.18999

6.863019673493637
6.8628173704594655
r = 4, rOpt = 3.43551, theta =  1.00000, thetaOpt = 1.00000

6.942054984148132
6.9414476082169205
r = 4, rOpt = 3.74774, theta =  1.00000, thetaOpt = 1.32587

6.928869540278924
6.928780381246793
r = 4, rOpt = 3.23388, theta =  1.00000, thetaOpt = 1.54465

6.864061134245936
6.84621776419379
r = 4, rOpt = 3.98079, theta =  1.00000, thetaOpt = 1.00000

7.027153534225495
6.917713557698754
r = 4, rOpt = 4.51165, theta =  1.00000, thetaOpt = 2.35318
```

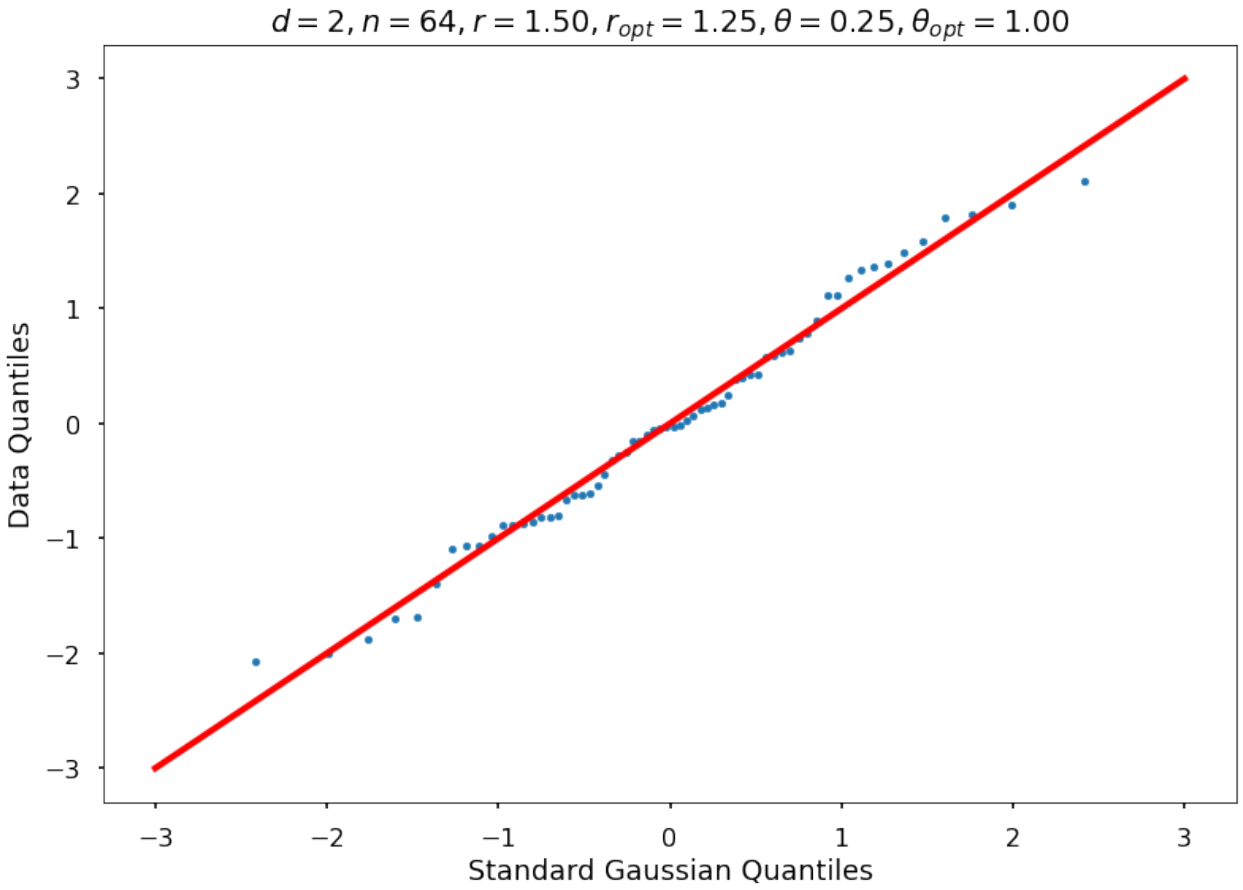

8.8
8.6
8.4
8.2
8.0
7.8
7.6
7.4
1.4
1.6
2.0
2.6
3.7
7.0
3.0
0.4
0.2

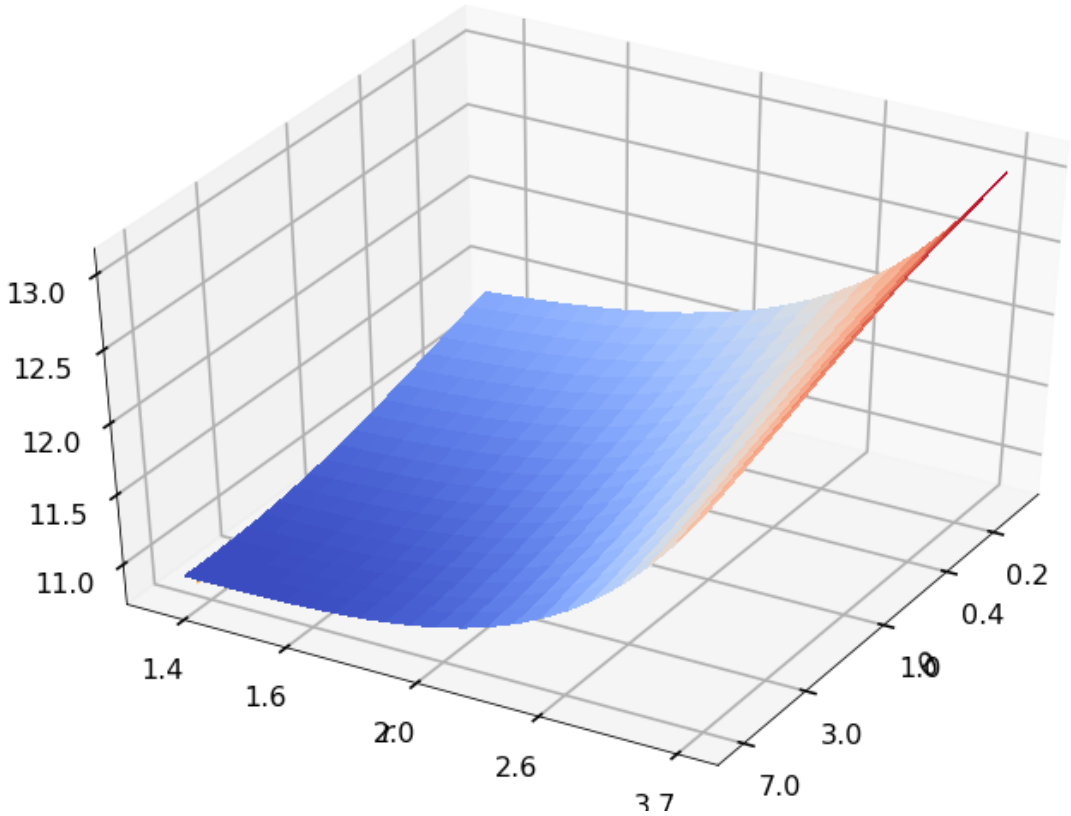

$d = 2, n = 64, r = 1.50, r_{opt} = 1.08, \theta = 0.25, \theta_{opt} = 0.19$
Data Quantiles
Standard Gaussian Quantiles
3
2
1
0
−1
−2
−3
−3
−2
−1
0
1
2
3

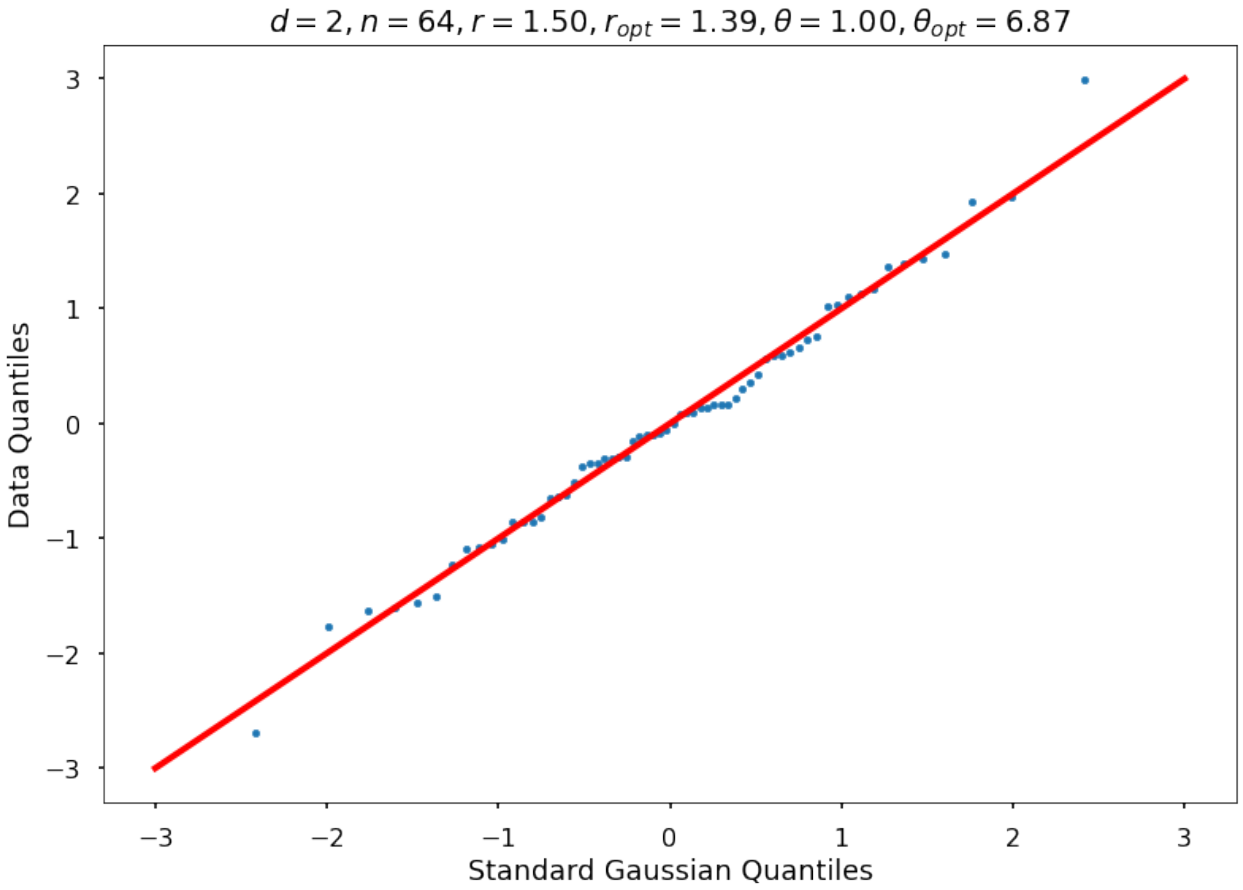

9.25
9.00
8.75
8.50
8.25
8.00
7.75
7.50
1.4
1.6
2.0
2.6
3.7
0.2
0.4
3.0
7.0

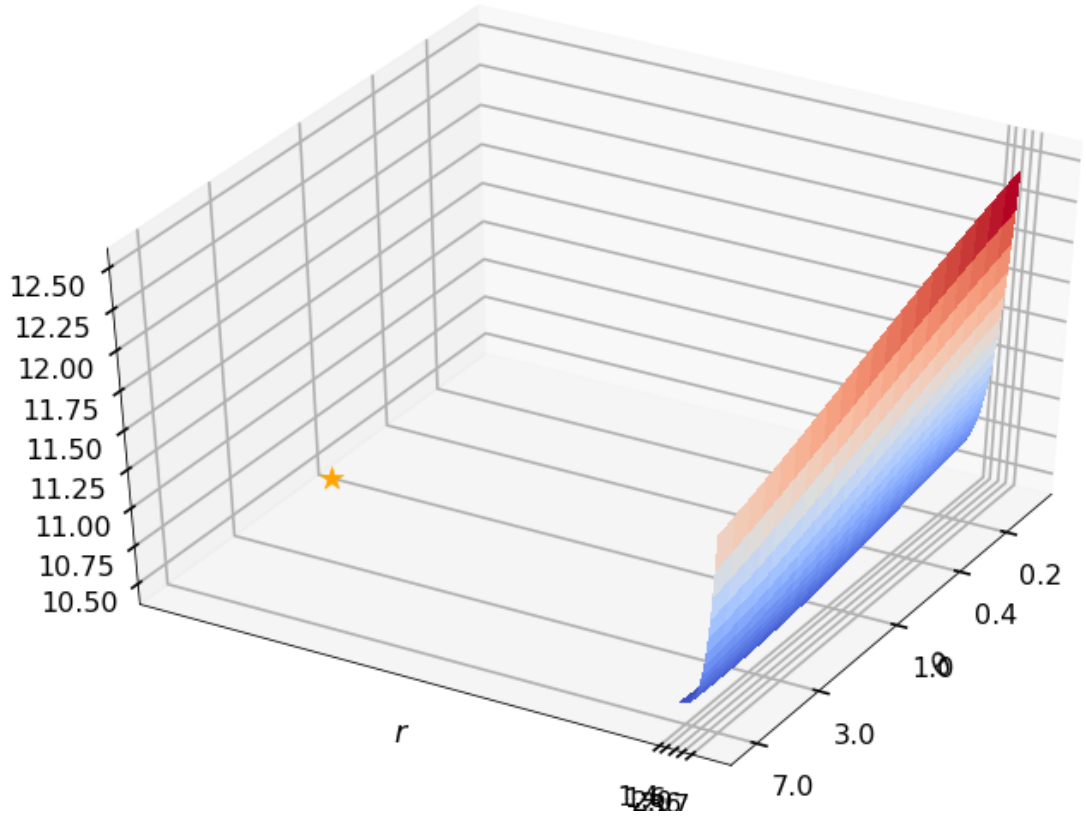

$d = 2, n = 64, r = 1.50, r_{opt} = 1.08, \theta = 0.25, \theta_{opt} = 0.38$
Data Quantiles
Standard Gaussian Quantiles
3
2
1
0
−1
−2
−3
−3
−2
−1
0
1
2
3

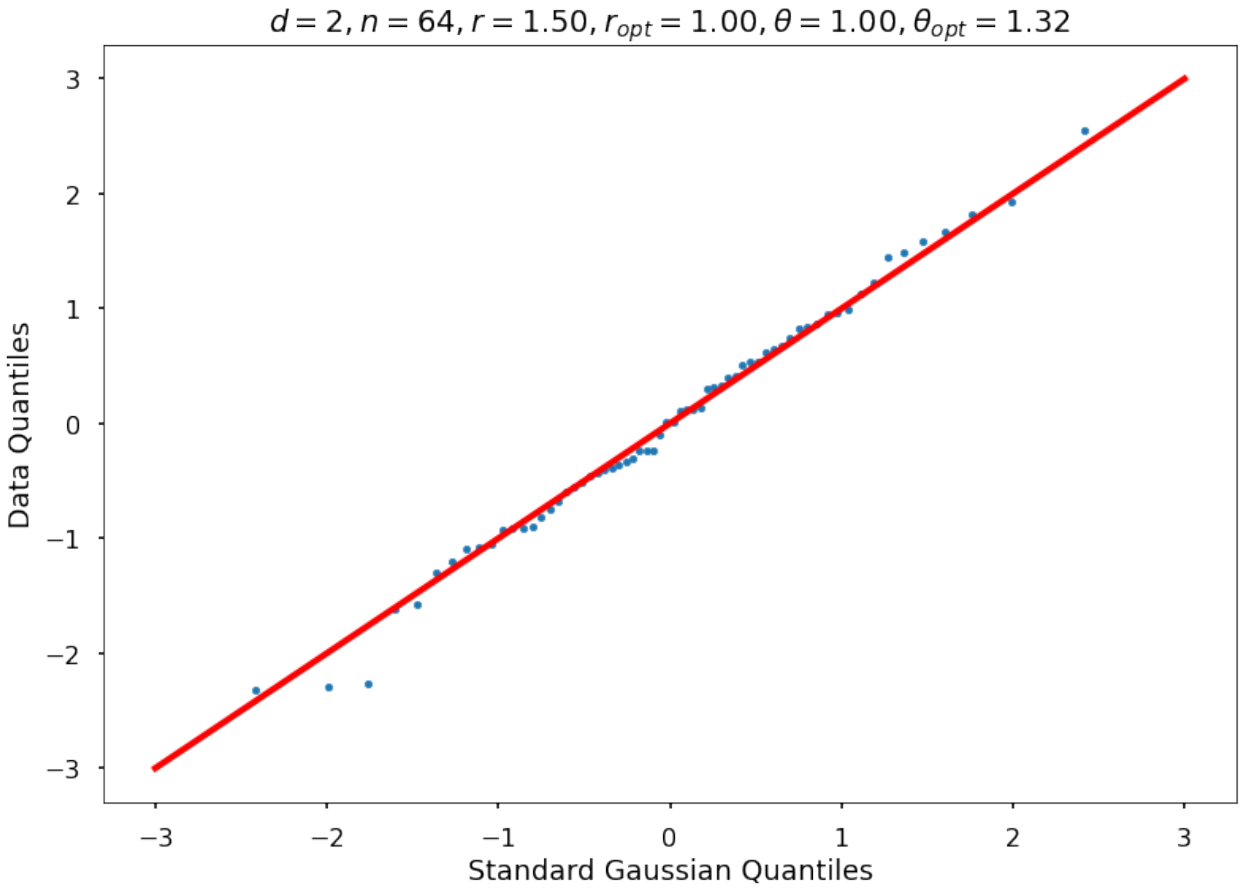

8.6
8.4
8.2
8.0
7.8
7.6
7.4
7.2
1.4
1.6
r
2.0
2.6
3.7
0.2
0.4
3.0
7.0

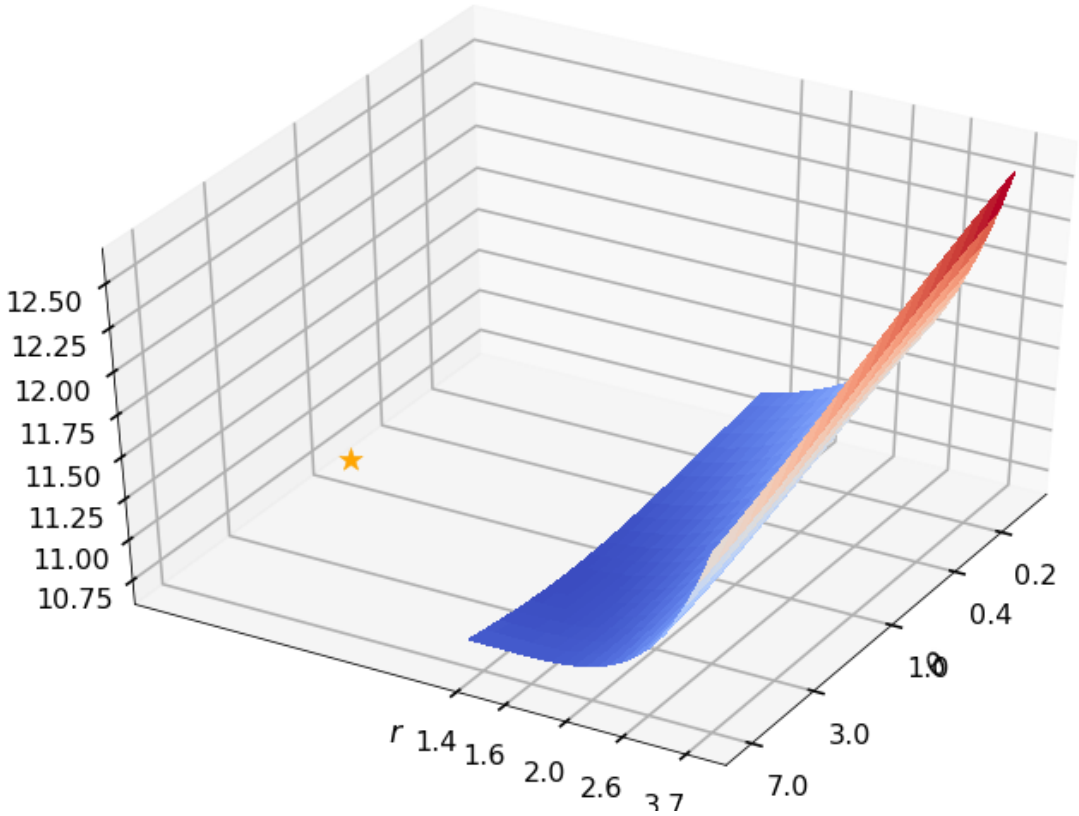

$d = 2, n = 64, r = 1.50, r_{opt} = 1.25, \theta = 0.25, \theta_{opt} = 1.00$
Data Quantiles
Standard Gaussian Quantiles
3
2
1
0
−1
−2
−3
−3
−2
−1
0
1
2
3

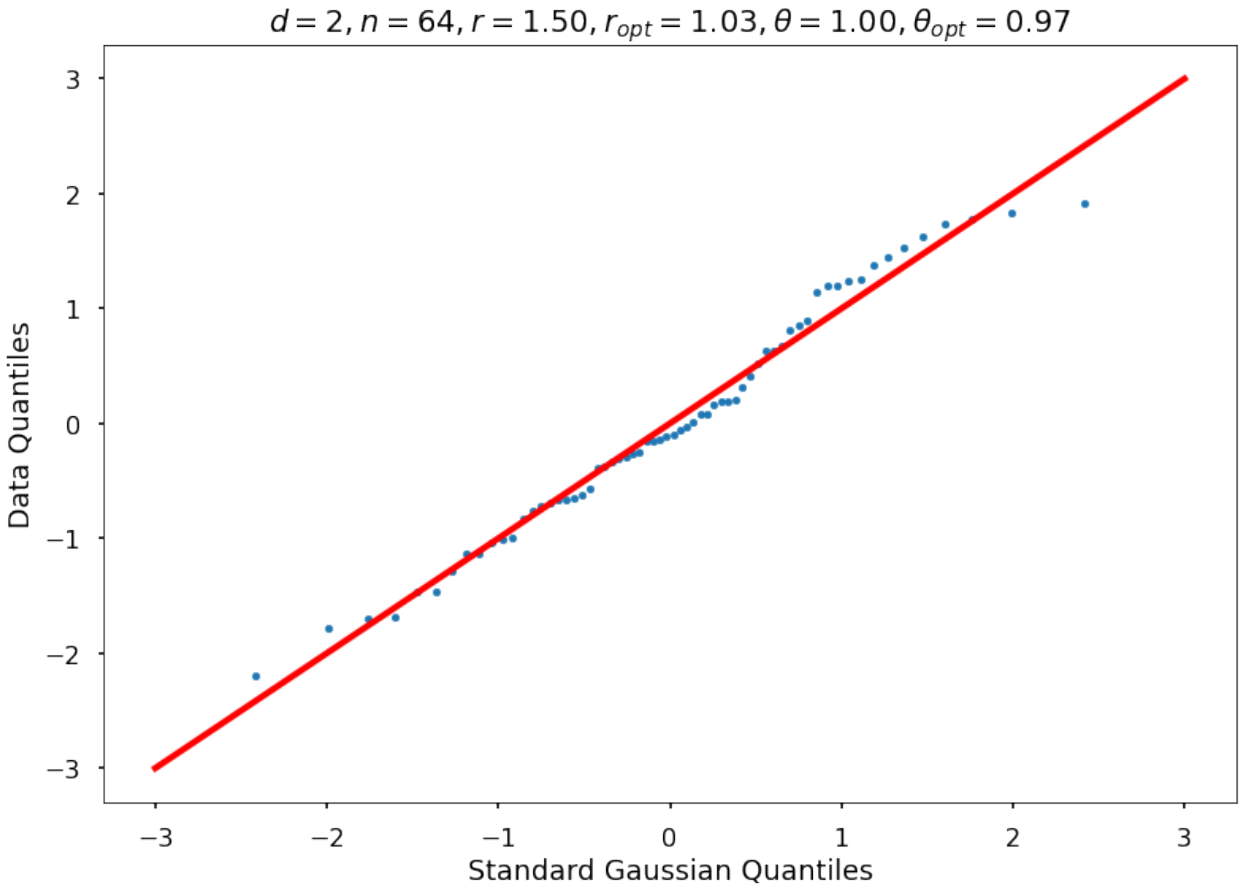

13.0
12.5
12.0
11.5
11.0
1.4
1.6
2.6
3.0
7.0
0.2
0.4

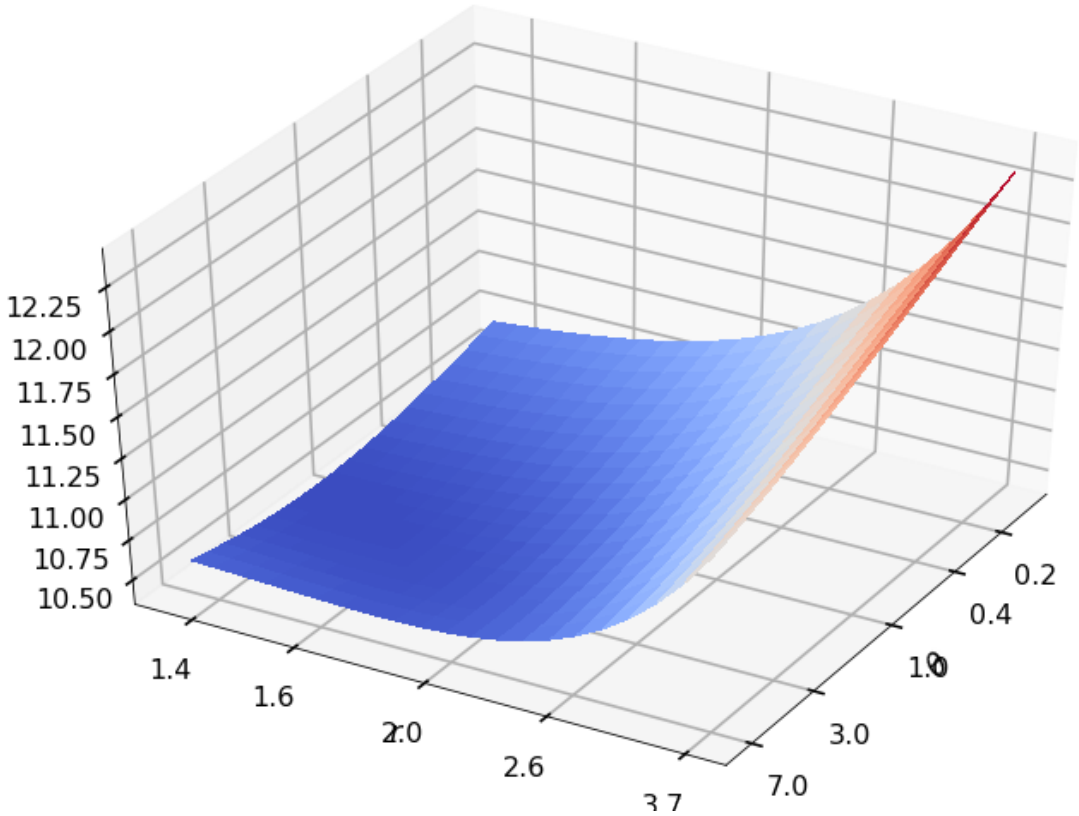

$d = 2, n = 64, r = 1.50, r_{opt} = 1.39, \theta = 1.00, \theta_{opt} = 6.87$
Data Quantiles
Standard Gaussian Quantiles
3
2
1
0
−1
−2
−3
−3
−2
−1
0
1
2
3

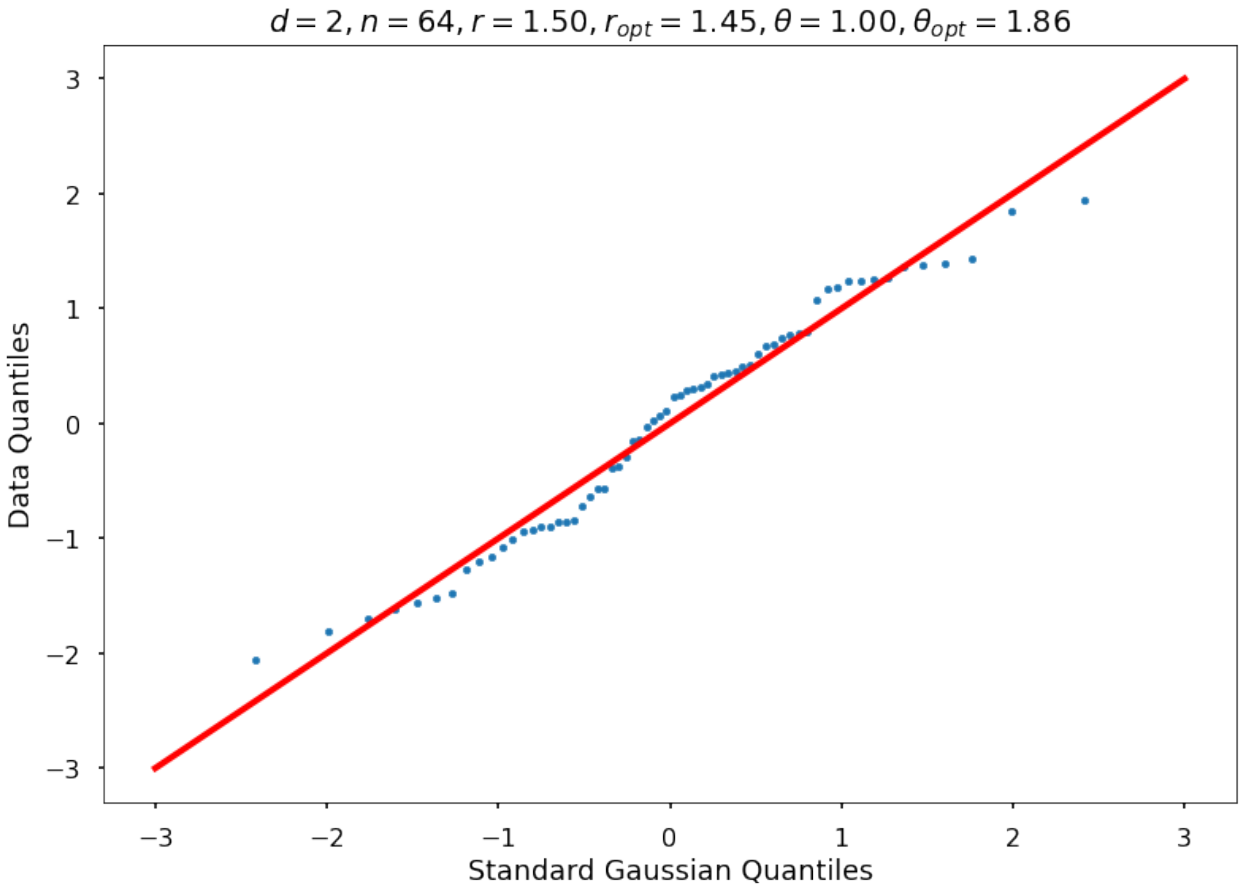

12.50
12.25
12.00
11.75
11.50
11.25
11.00
10.75
10.50
r
0.2
0.4
3.0
7.0

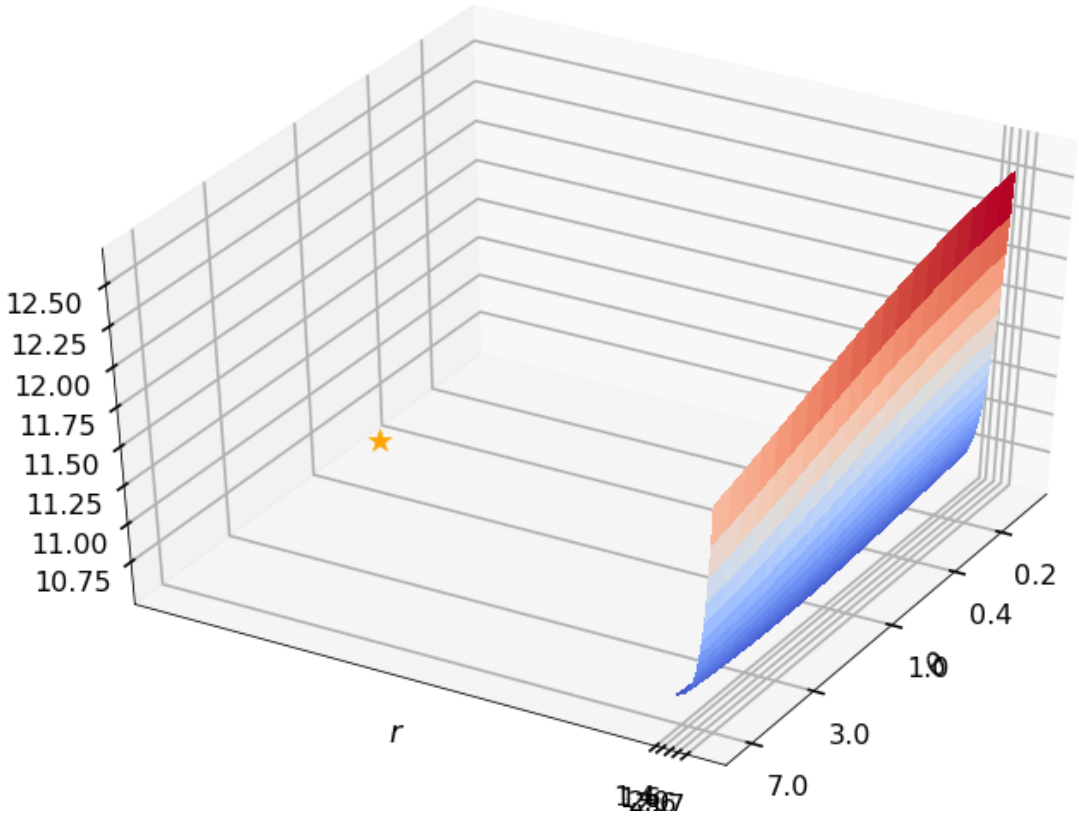

$d = 2, n = 64, r = 1.50, r_{opt} = 1.00, \theta = 1.00, \theta_{opt} = 1.32$
Data Quantiles
Standard Gaussian Quantiles
3
2
1
0
−1
−2
−3
−3
−2
−1
0
1
2
3

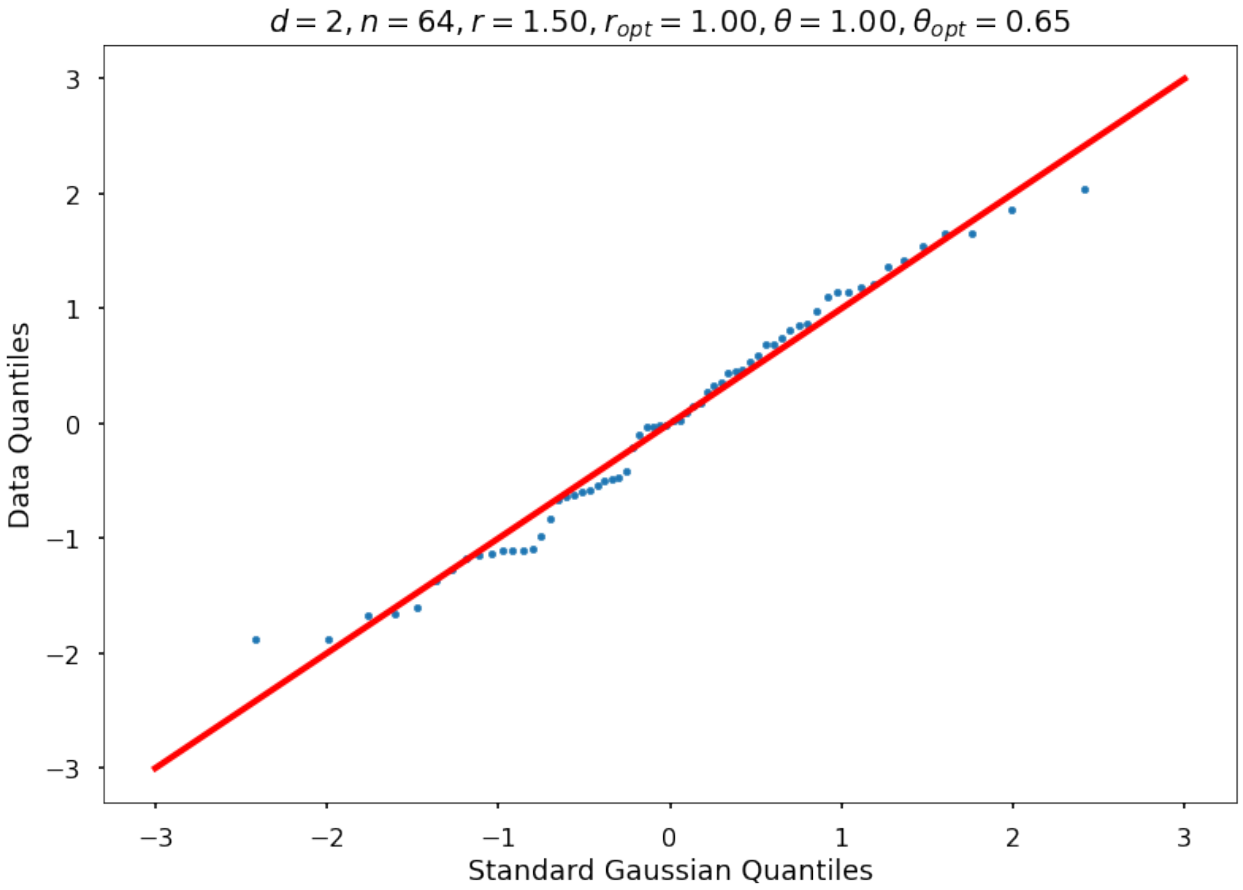

12.50
12.25
12.00
11.75
11.50
11.25
11.00
10.75
r
1.4
1.6
2.0
2.6
3.7
0.2
0.4
3.0
7.0

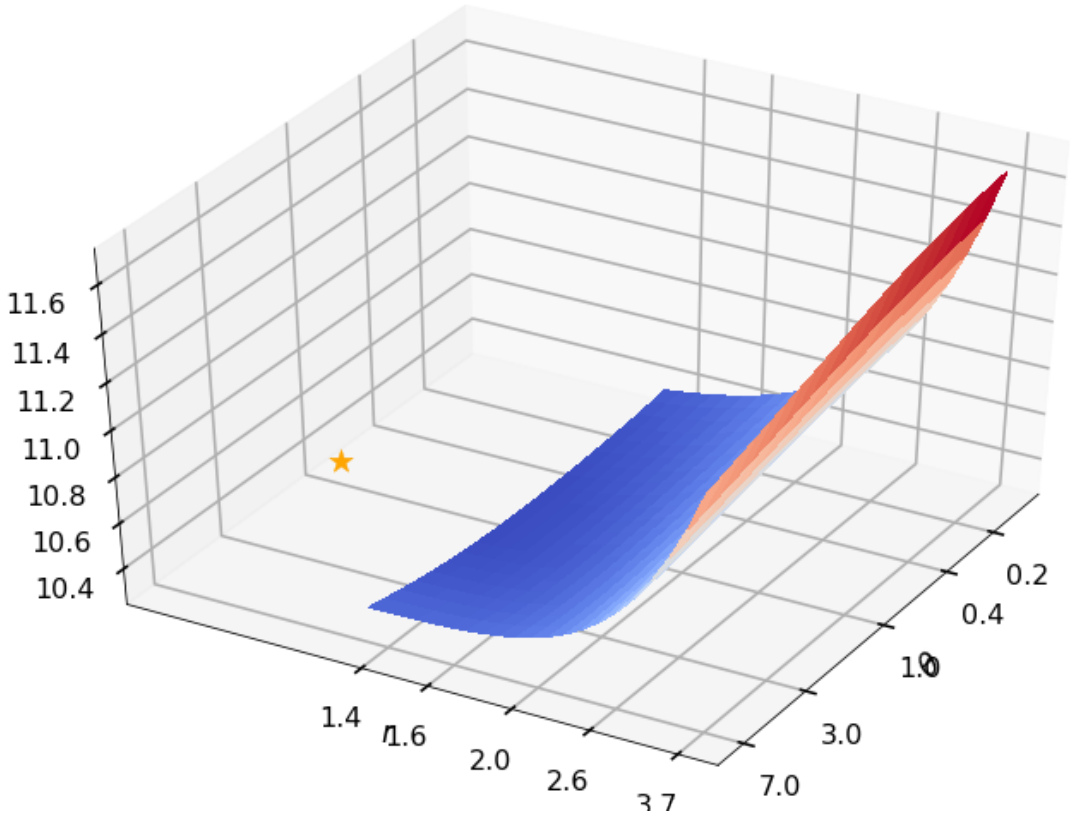

$d = 2, n = 64, r = 1.50, r_{opt} = 1.03, \theta = 1.00, \theta_{opt} = 0.97$
Data Quantiles
Standard Gaussian Quantiles
3
2
1
0
−1
−2
−3
−3
−2
−1
0
1
2
3

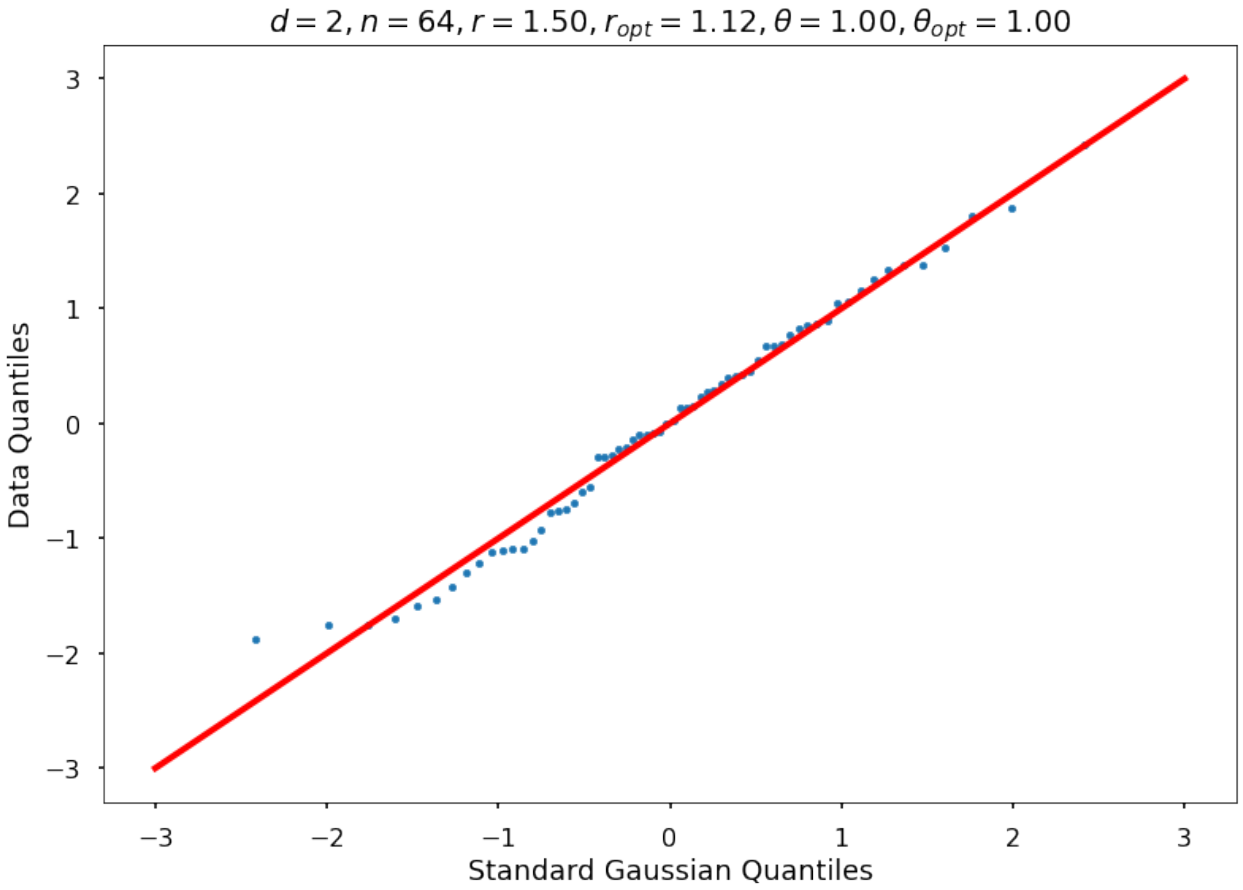

12.25
12.00
11.75
11.50
11.25
11.00
10.75
10.50
1.4
1.6
2.6
0.2
0.4
3.0
7.0

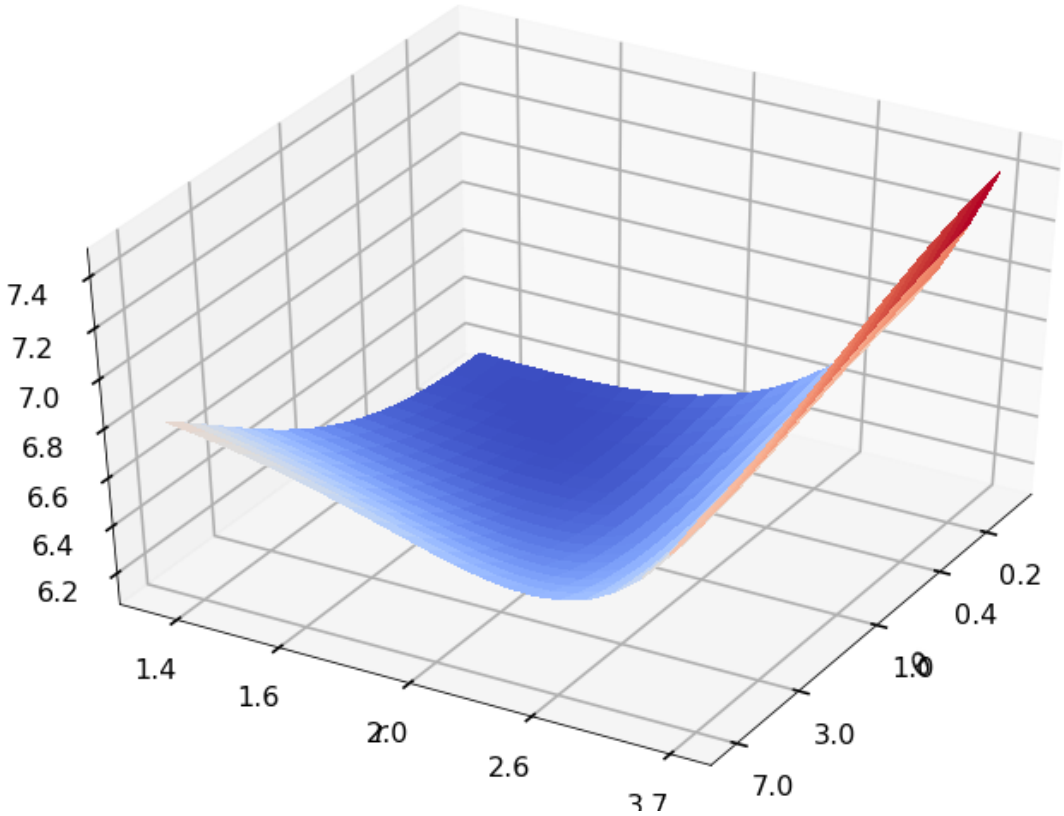

$d = 2, n = 64, r = 1.50, r_{opt} = 1.45, \theta = 1.00, \theta_{opt} = 1.86$
Data Quantiles
Standard Gaussian Quantiles
3
2
1
0
−1
−2
−3
−3
−2
−1
0
1
2
3

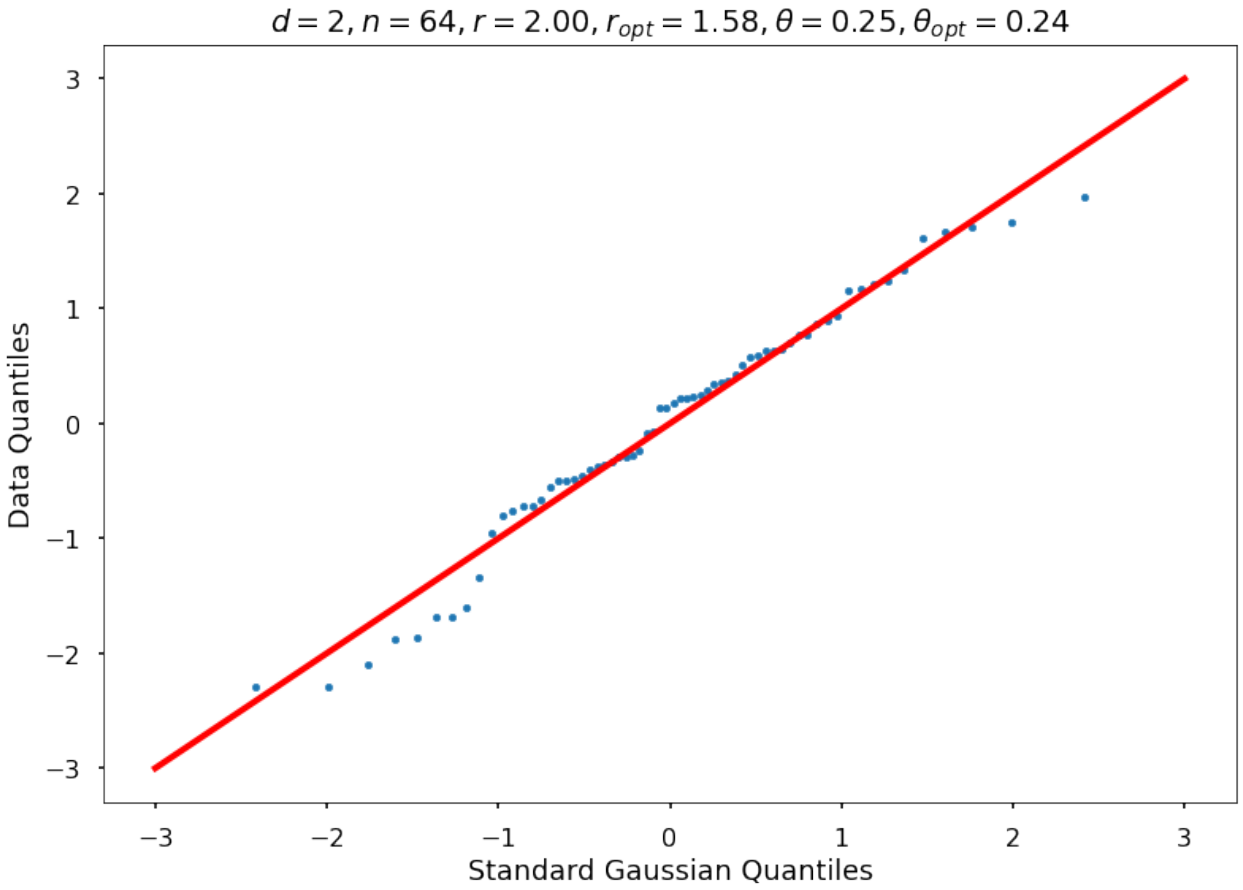

12.50
12.25
12.00
11.75
11.50
11.25
11.00
10.75
r
0.2
0.4
3.0
7.0

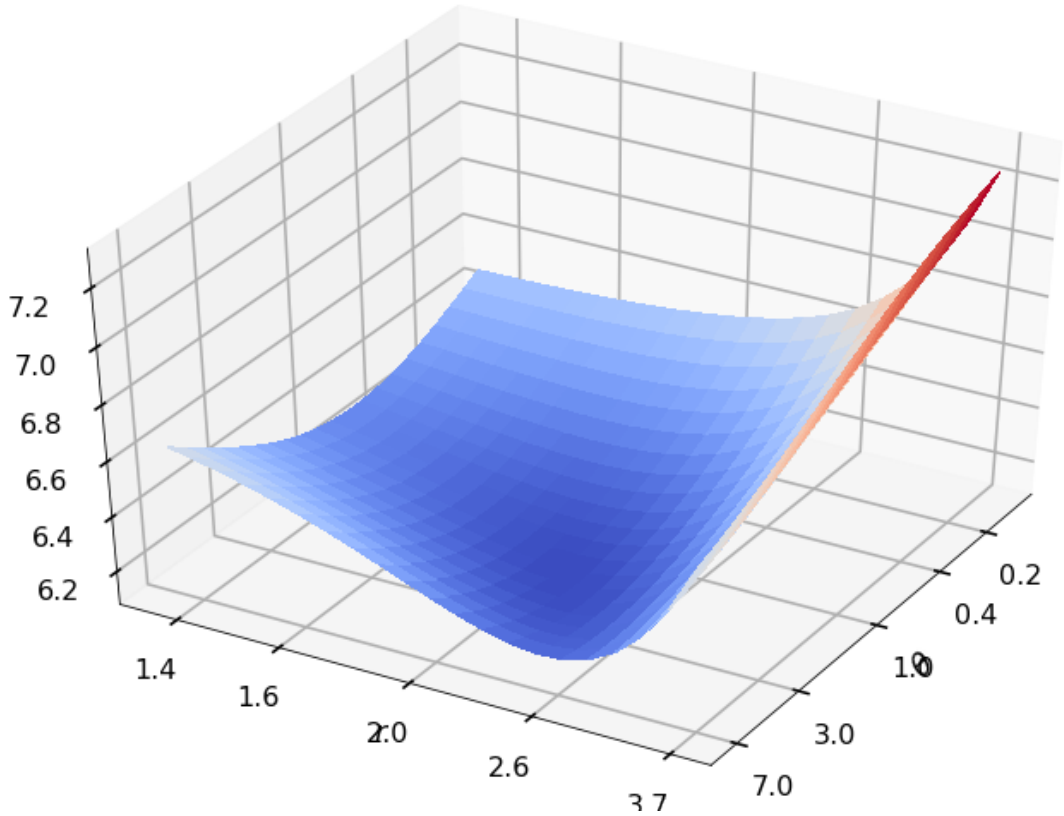

$d = 2, n = 64, r = 1.50, r_{opt} = 1.00, \theta = 1.00, \theta_{opt} = 0.65$
Data Quantiles
Standard Gaussian Quantiles
3
2
1
0
−1
−2
−3
−3
−2
−1
0
1
2
3

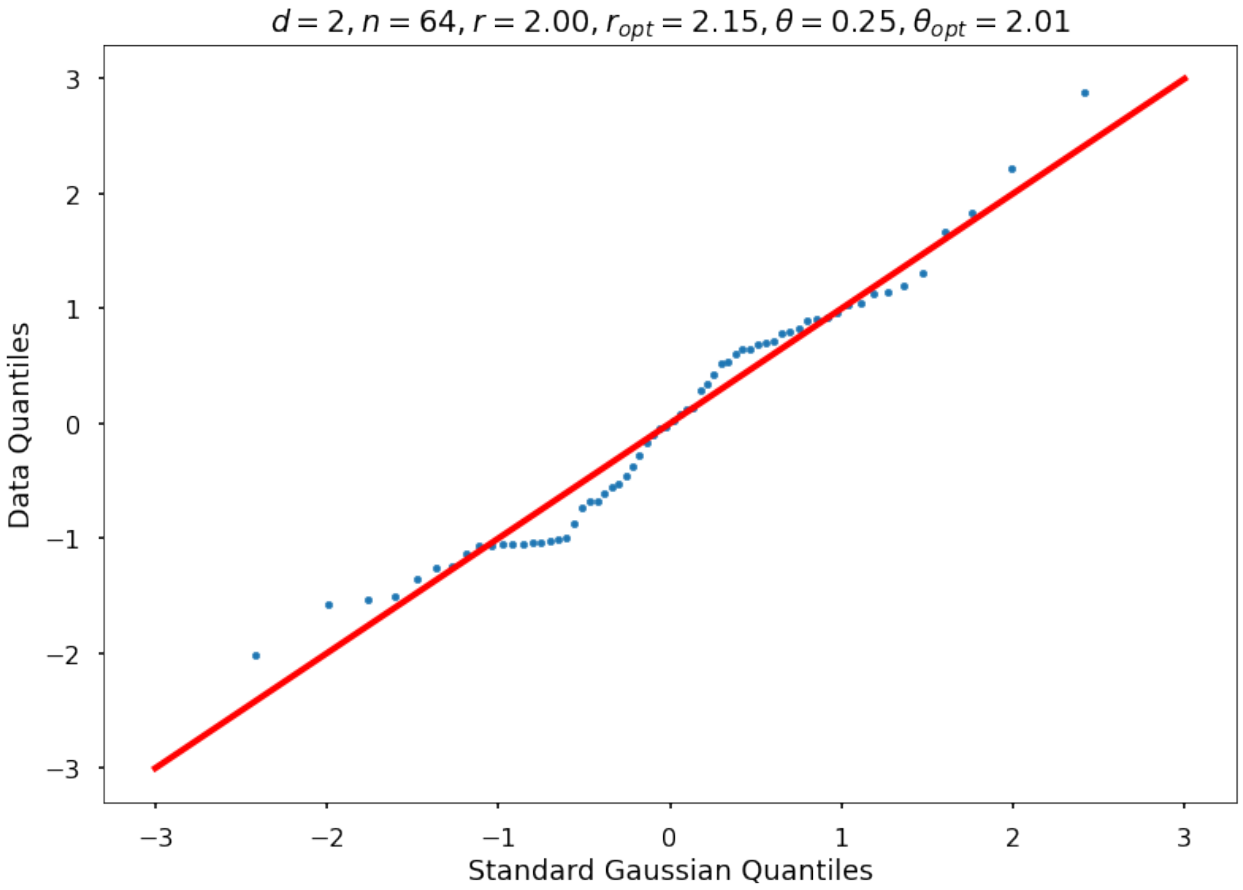

11.6
11.4
11.2
11.0
10.8
10.6
10.4
1.4
1.6
2.0
2.6
3.7
0.2
0.4
3.0
7.0

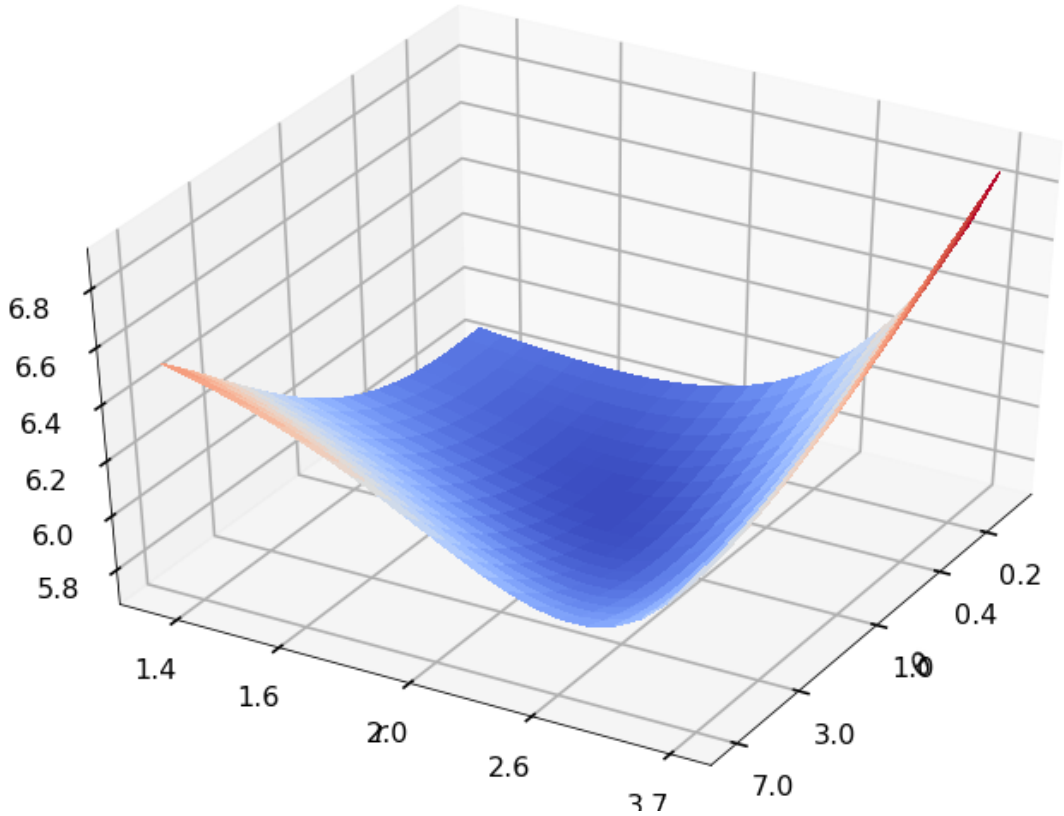

$d = 2, n = 64, r = 1.50, r_{opt} = 1.12, \theta = 1.00, \theta_{opt} = 1.00$
Data Quantiles
Standard Gaussian Quantiles
3
2
1
0
−1
−2
−3
−3
−2
−1
0
1
2
3

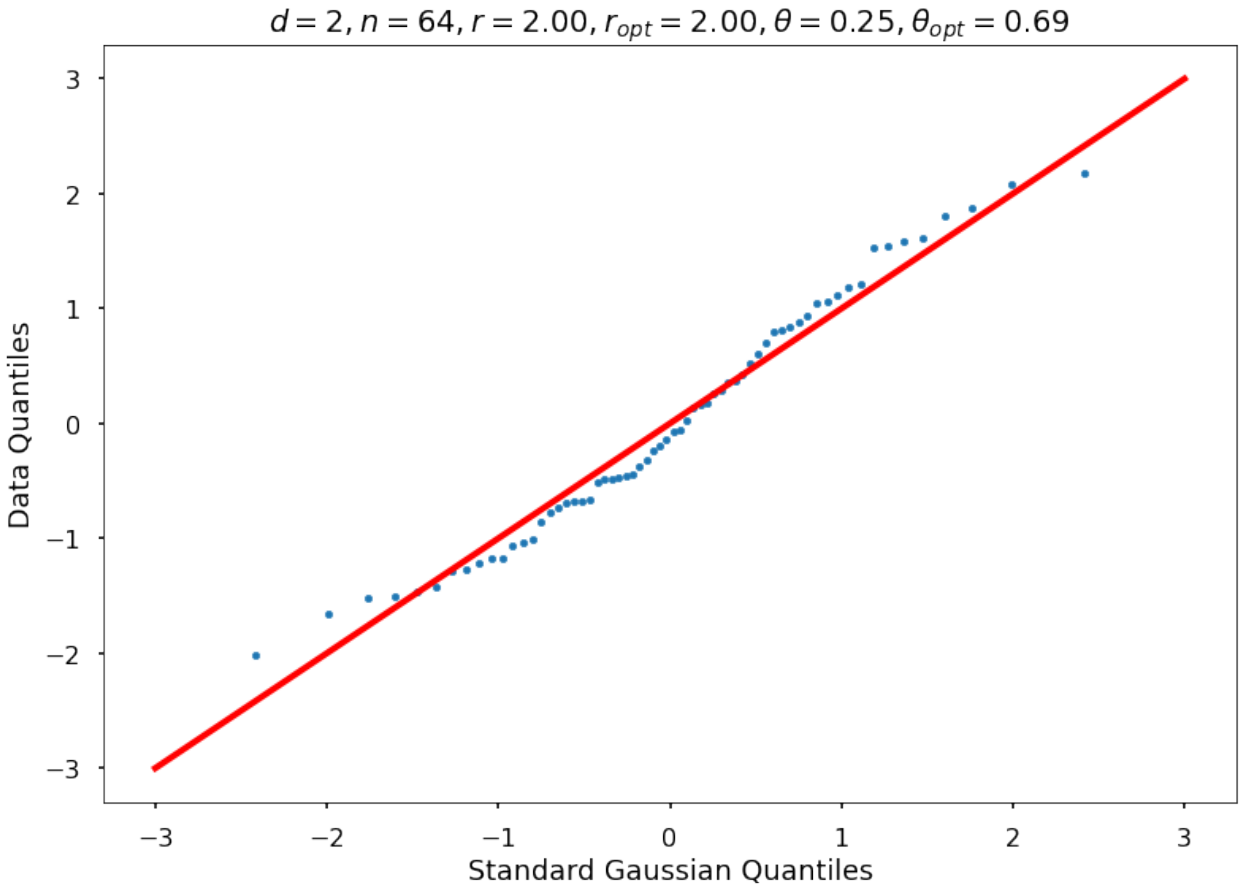

7.4
7.2
7.0
6.8
6.6
6.4
6.2
1.4
1.6
2.6
3.0
7.0
0.2
0.4

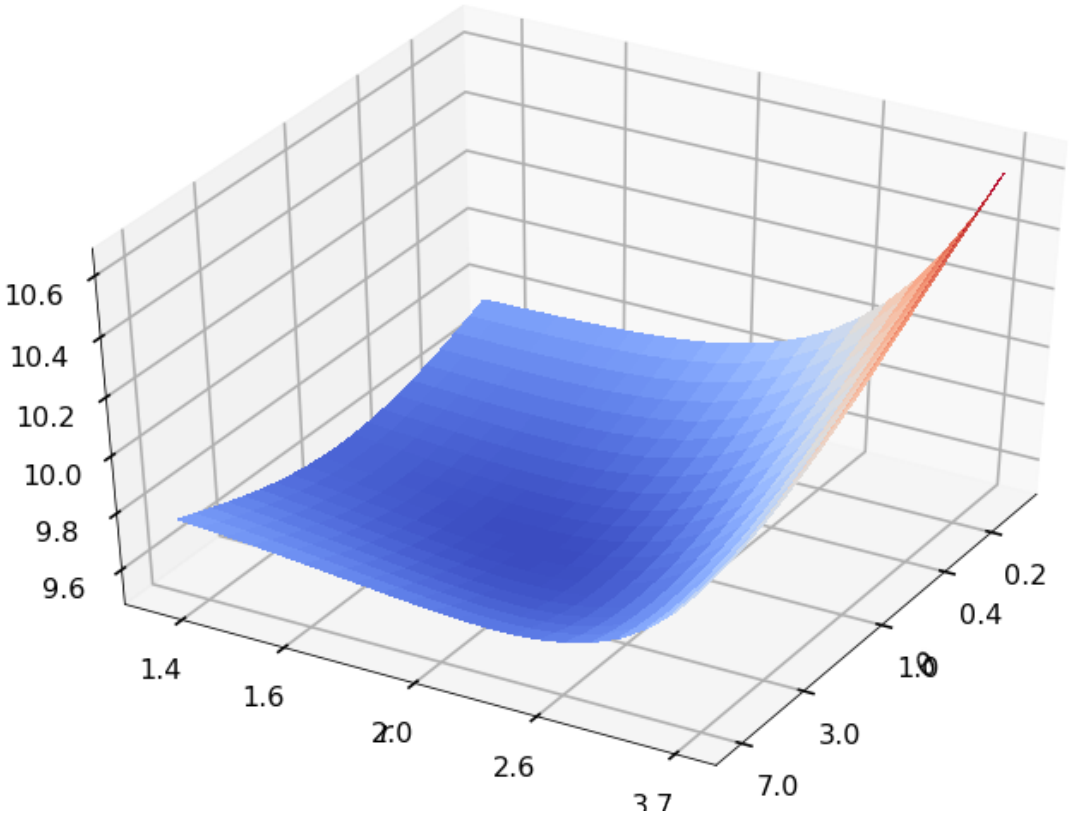

$d = 2, n = 64, r = 2.00, r_{opt} = 1.58, \theta = 0.25, \theta_{opt} = 0.24$
Data Quantiles
Standard Gaussian Quantiles
3
2
1
0
−1
−2
−3
−3
−2
−1
0
1
2
3

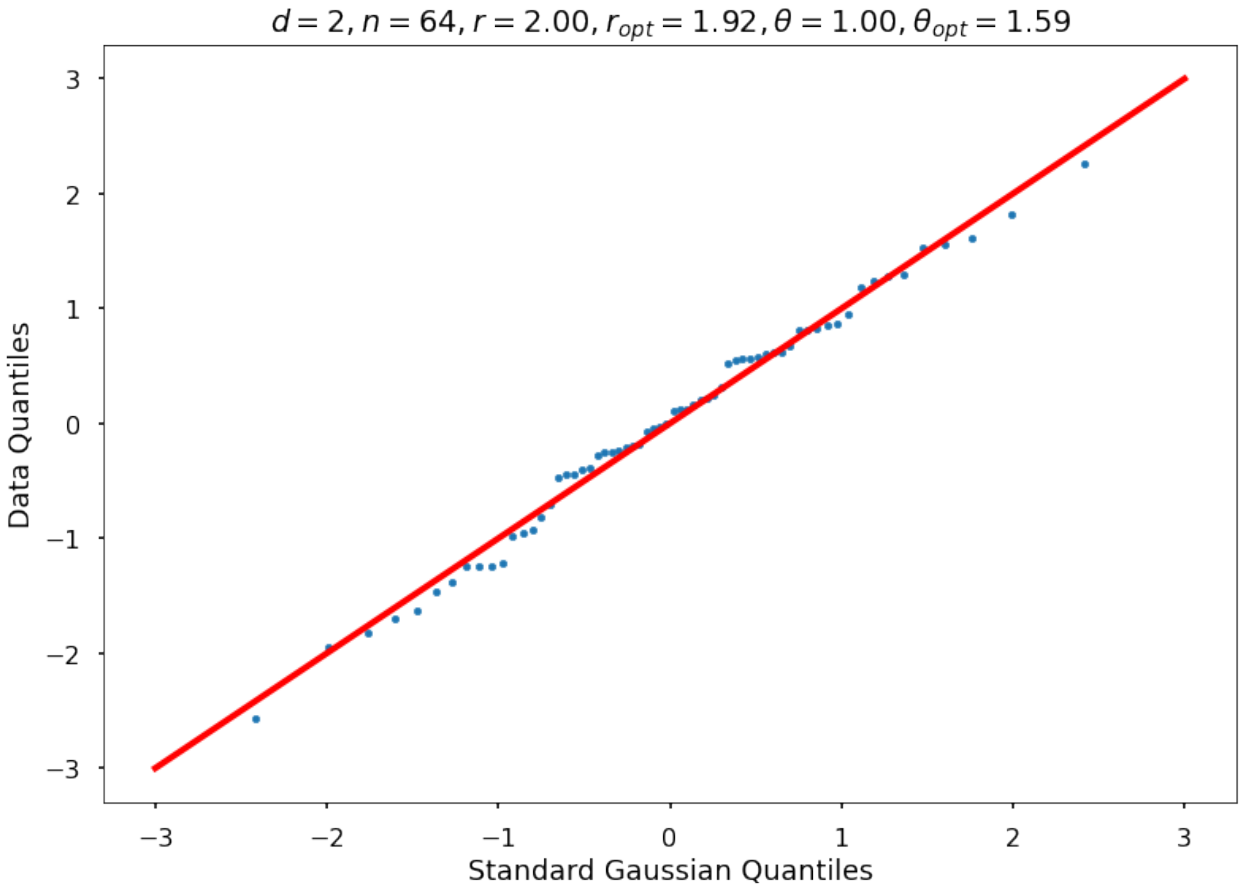

7.2
7.0
6.8
6.6
6.4
6.2
1.4
1.6
2.0
2.6
3.7
0.2
0.4
3.0
7.0

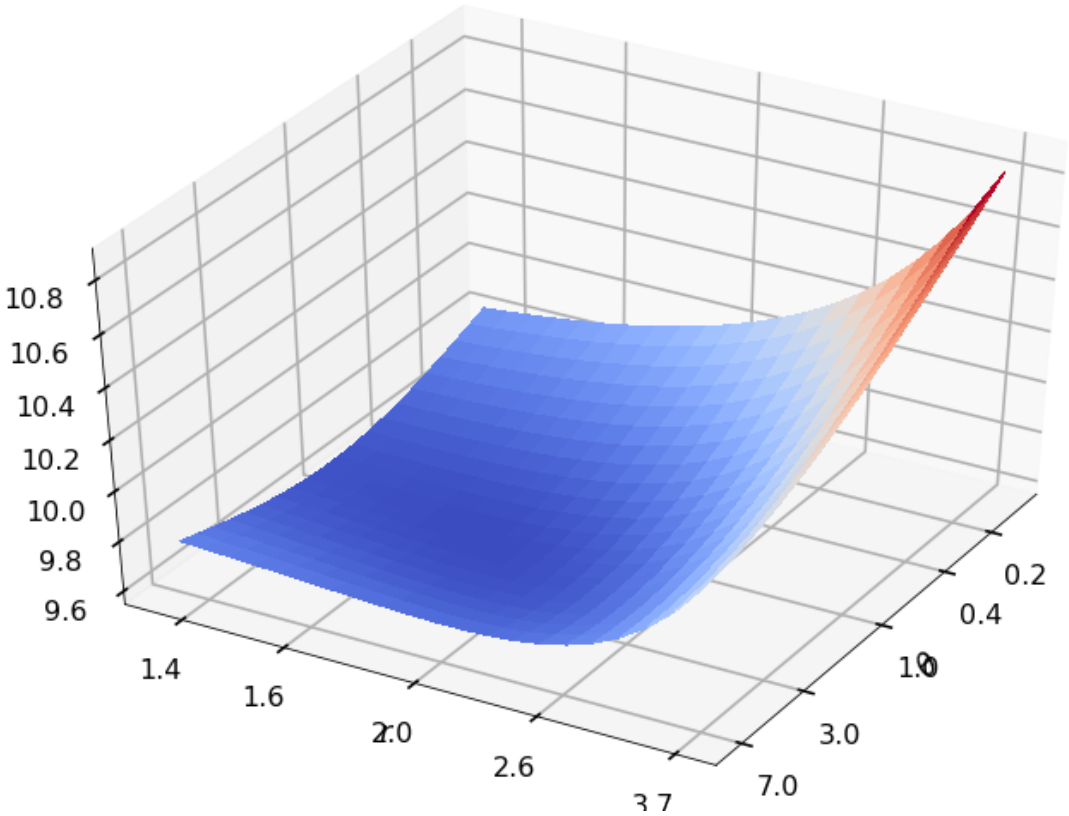

$d = 2, n = 64, r = 2.00, r_{opt} = 2.15, \theta = 0.25, \theta_{opt} = 2.01$
Data Quantiles
Standard Gaussian Quantiles
3
2
1
0
−1
−2
−3
−3
−2
−1
0
1
2
3

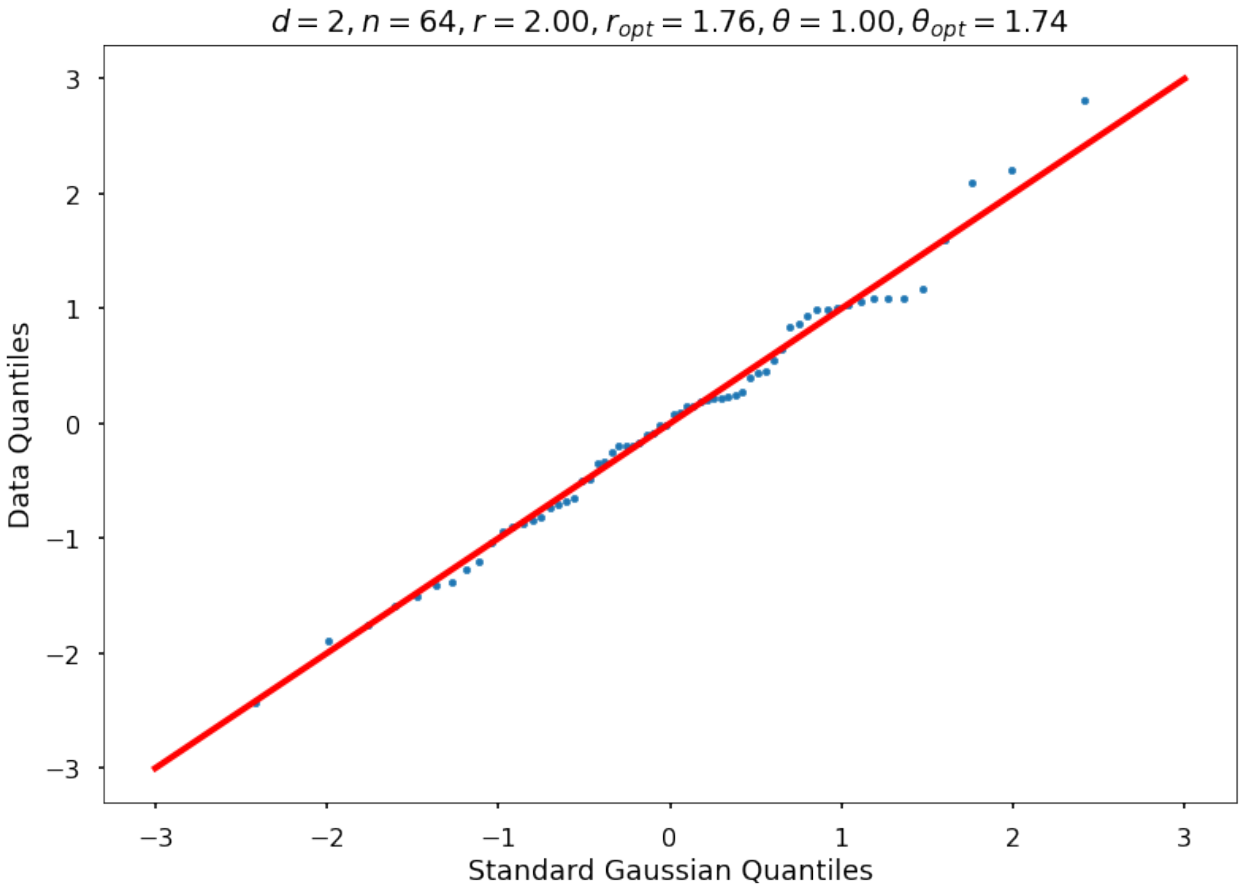

6.8
6.6
6.4
6.2
6.0
5.8
1.4
1.6
2.6
0.2
0.4
3.0
7.0

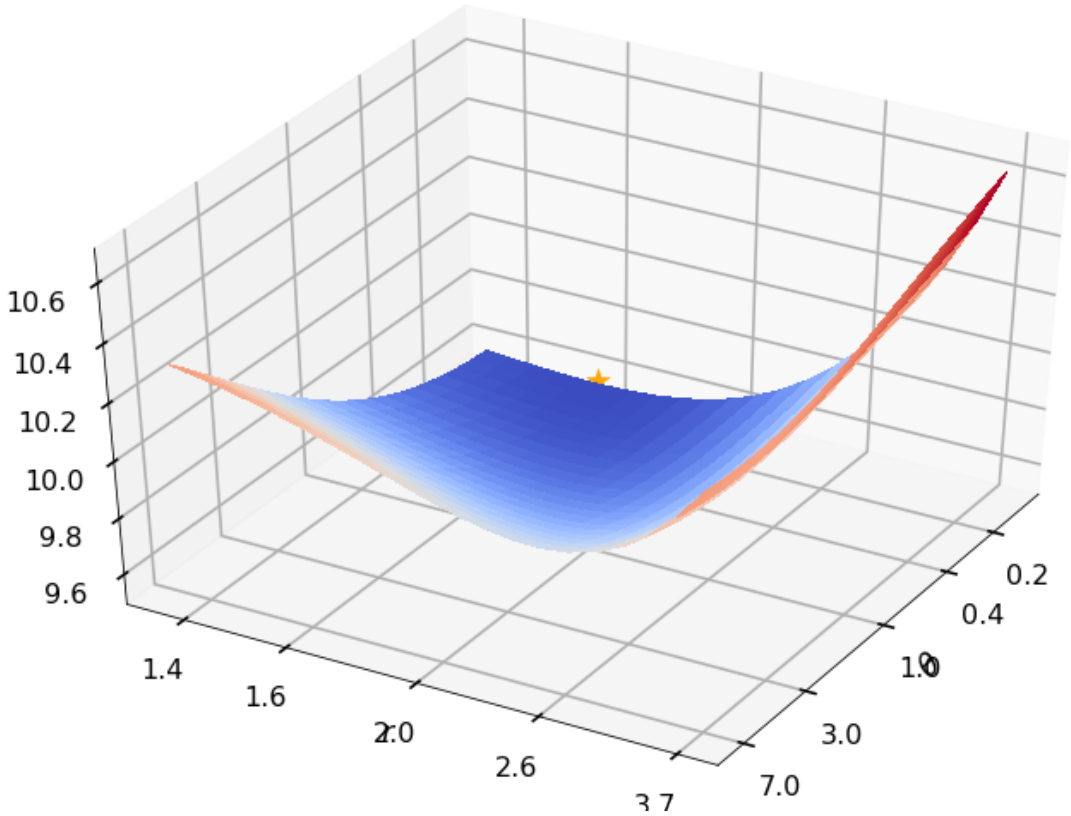

$d = 2, n = 64, r = 2.00, r_{opt} = 2.00, \theta = 0.25, \theta_{opt} = 0.69$
Data Quantiles
Standard Gaussian Quantiles
3
2
1
0
−1
−2
−3
−3
−2
−1
0
1
2
3

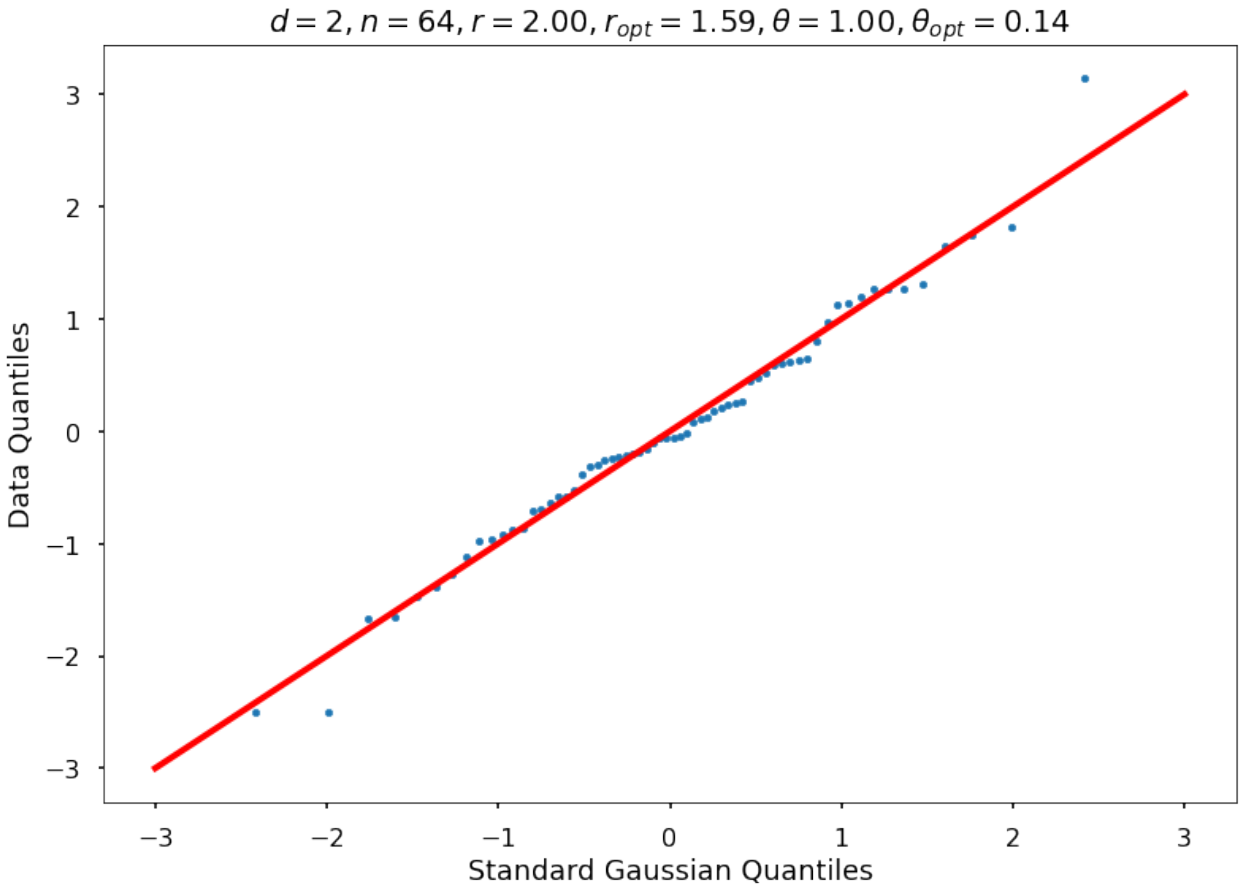

10.6
10.4
10.2
10.0
9.8
9.6
1.4
1.6
2.6
3.0
7.0
0.2
0.4

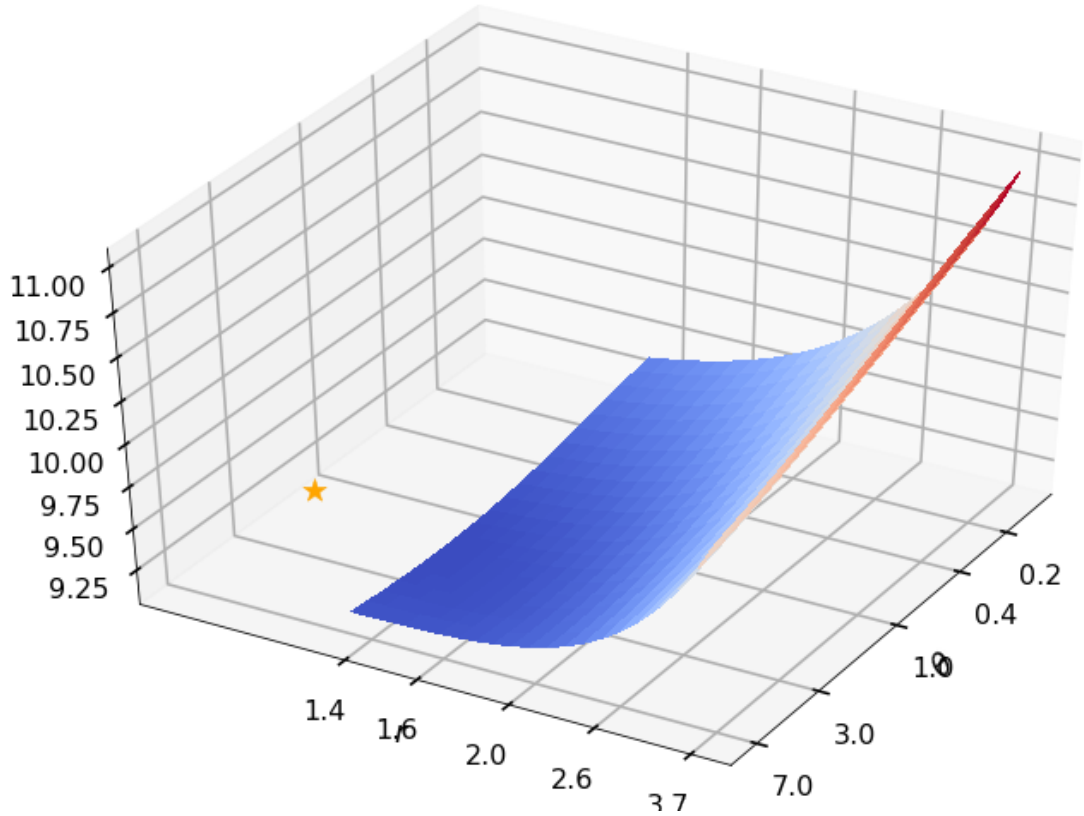

$d = 2, n = 64, r = 2.00, r_{opt} = 1.92, \theta = 1.00, \theta_{opt} = 1.59$
Data Quantiles
Standard Gaussian Quantiles
3
2
1
0
−1
−2
−3
−3
−2
−1
0
1
2
3

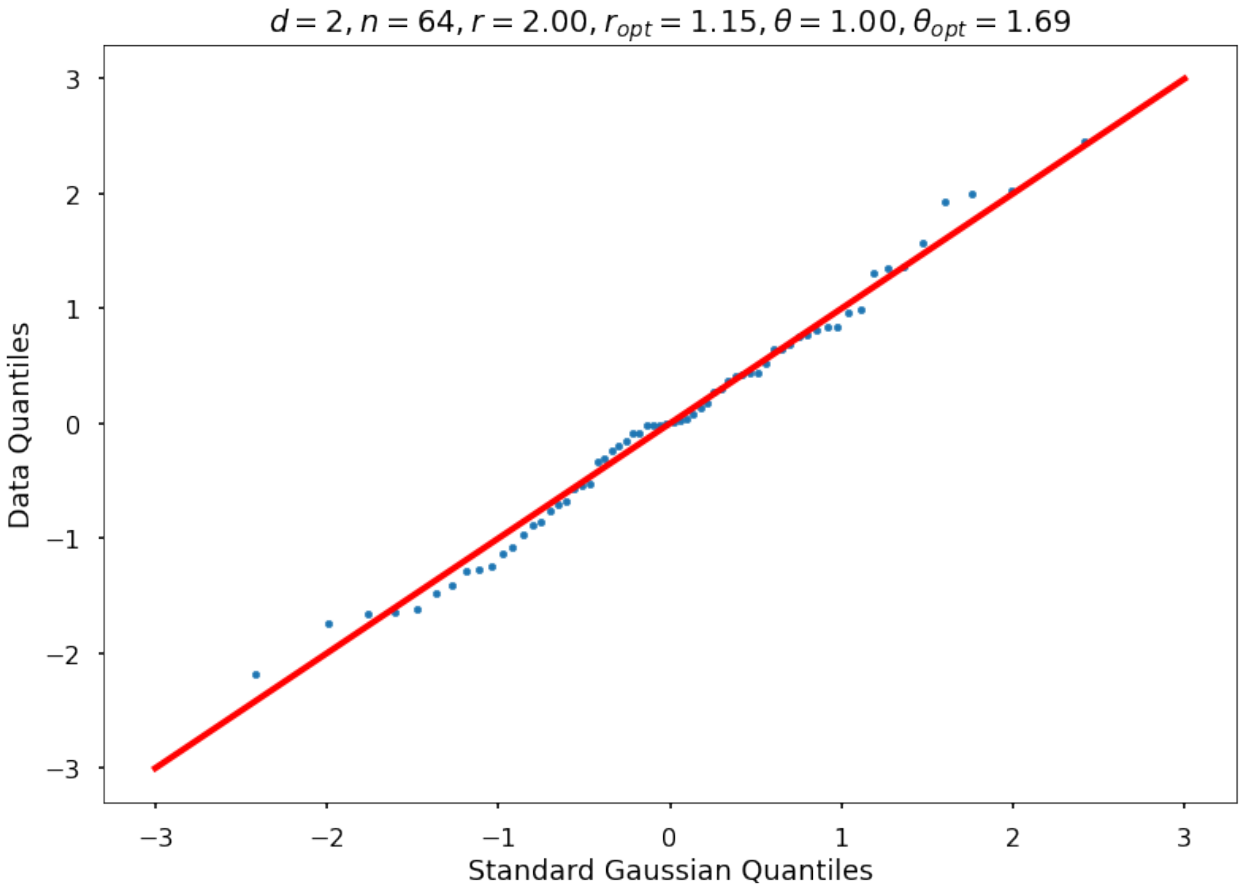

10.8
10.6
10.4
10.2
10.0
9.8
9.6
1.4
1.6
2.6
0.2
0.4
3.0
7.0

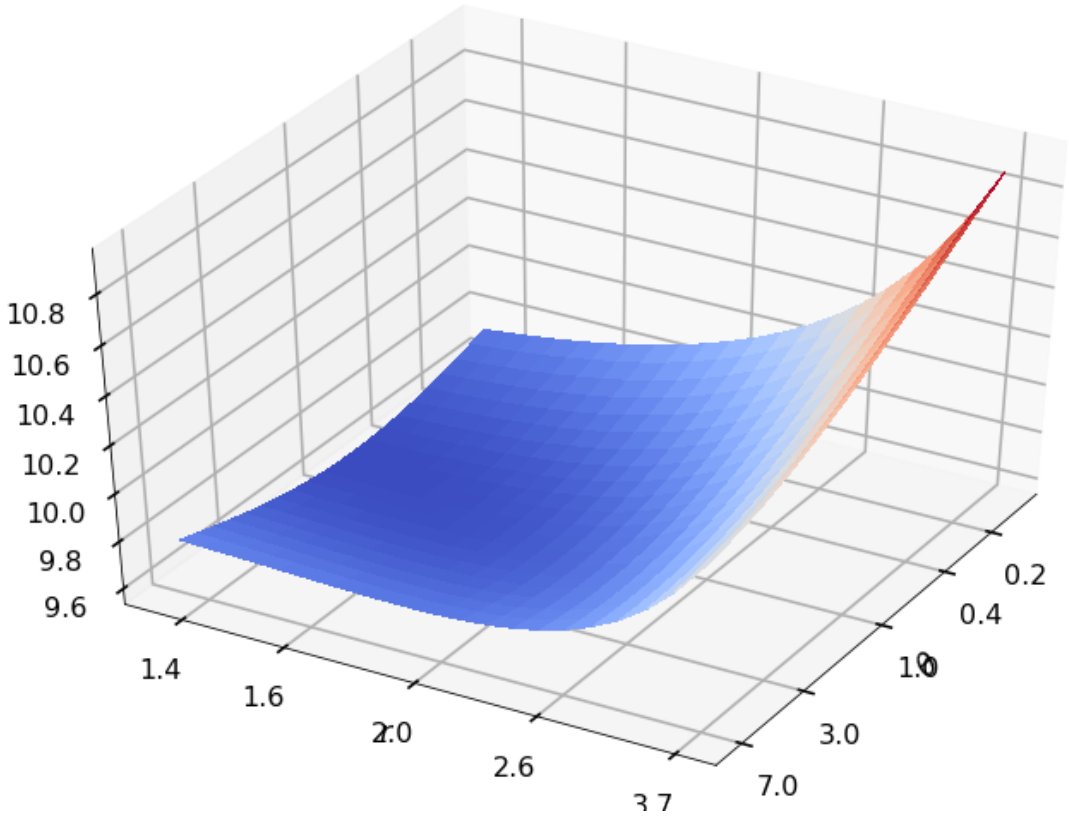

$d = 2, n = 64, r = 2.00, r_{opt} = 1.76, \theta = 1.00, \theta_{opt} = 1.74$
Data Quantiles
Standard Gaussian Quantiles
3
2
1
0
−1
−2
−3
−3
−2
−1
0
1
2
3

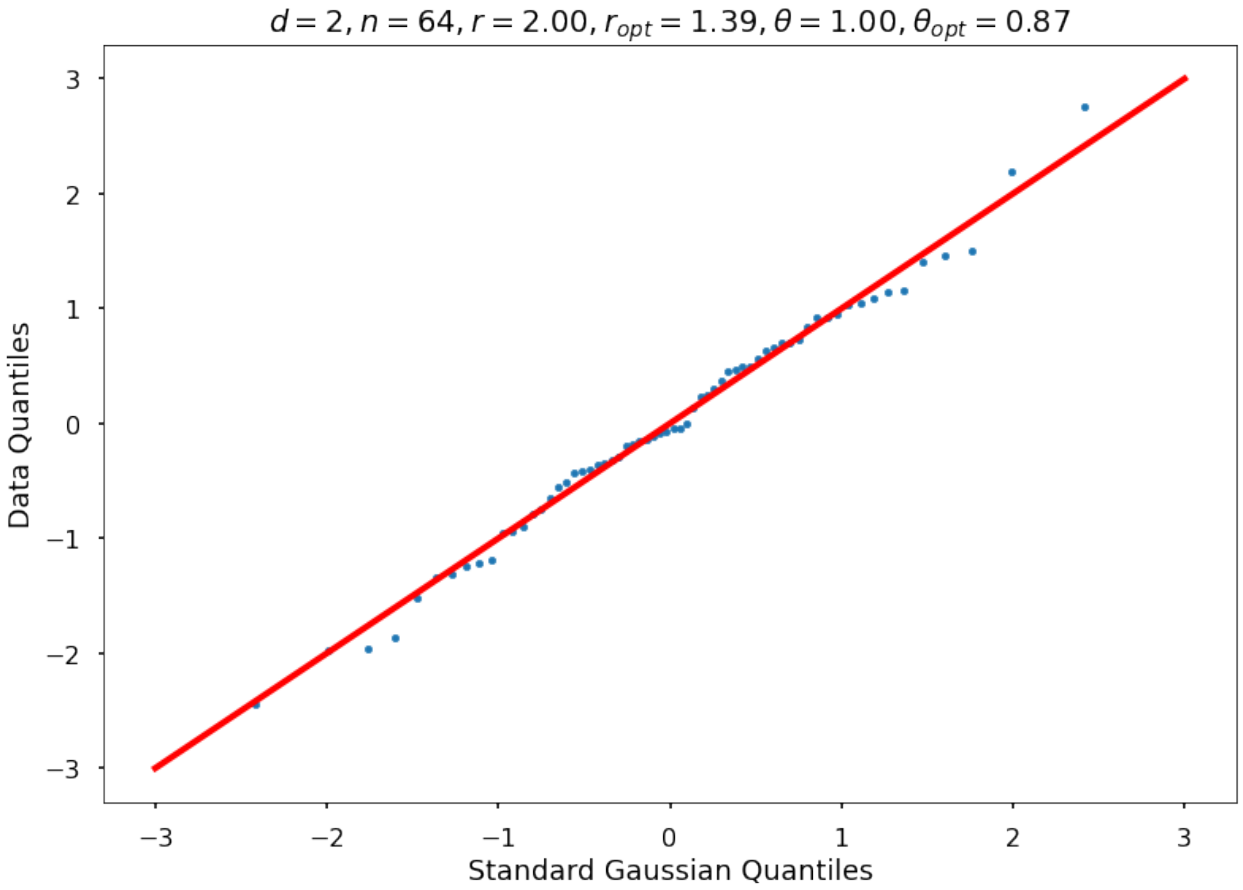

10.6
10.4
10.2
10.0
9.8
9.6
1.4
1.6
2.6
3.0
7.0
0.2
0.4

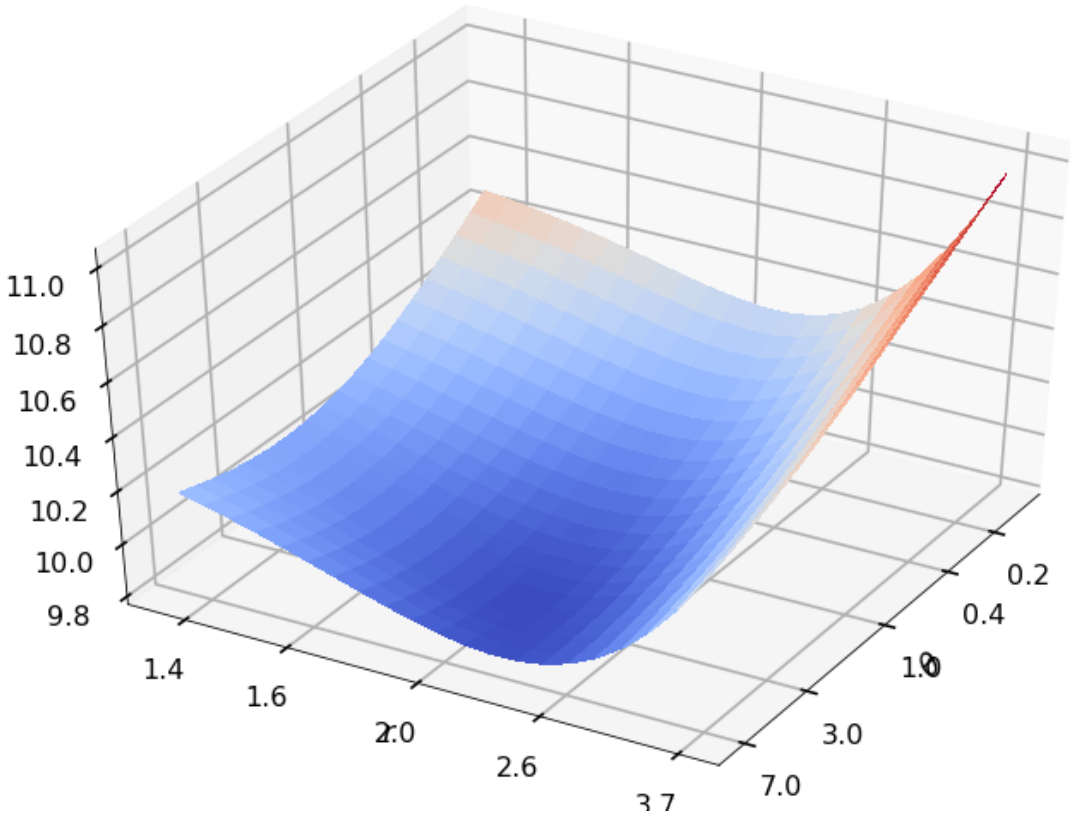

$d = 2, n = 64, r = 2.00, r_{opt} = 1.59, \theta = 1.00, \theta_{opt} = 0.14$
Data Quantiles
Standard Gaussian Quantiles
3
2
1
0
−1
−2
−3
−3
−2
−1
0
1
2
3

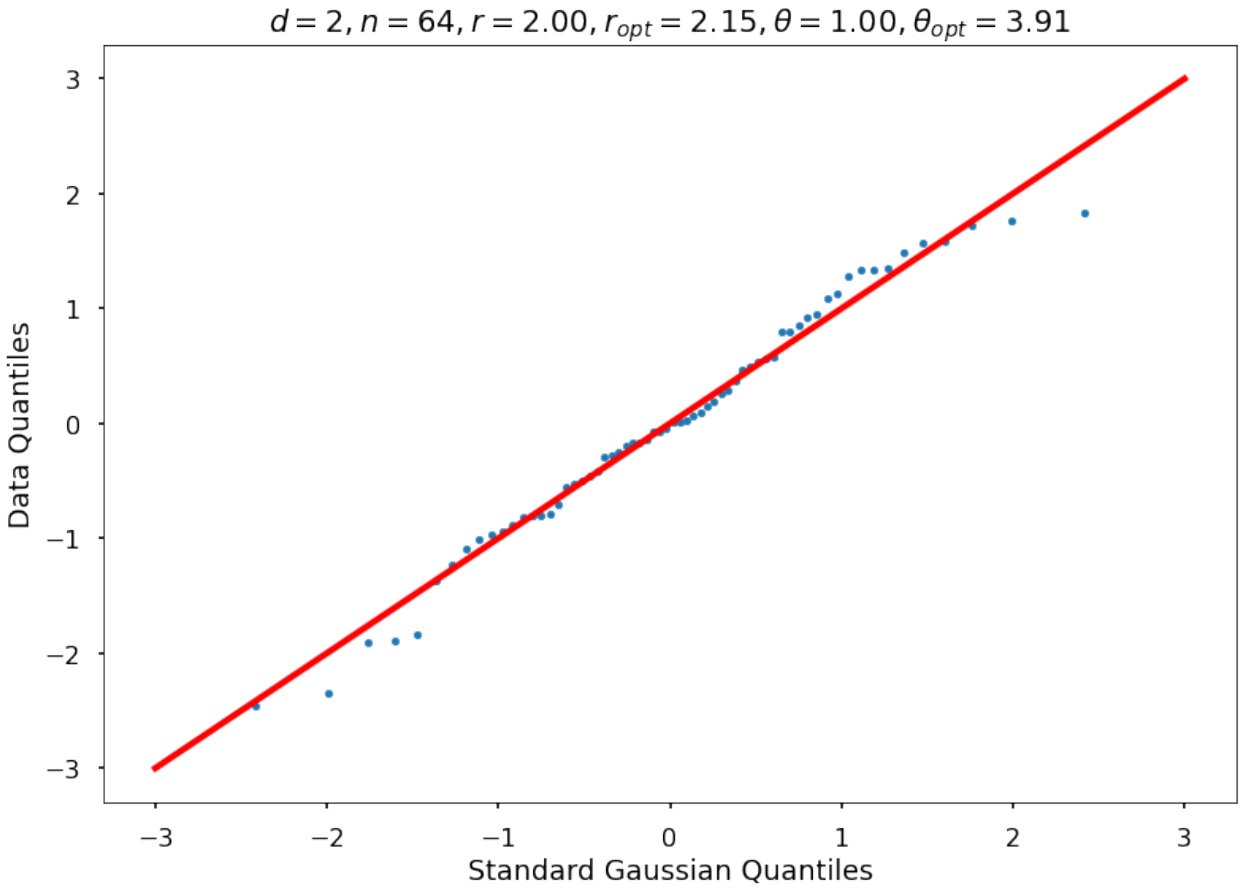

11.00
10.75
10.50
10.25
10.00
9.75
9.50
9.25
1.4
1.6
2.0
2.6
3.7
3.0
7.0
0.2
0.4

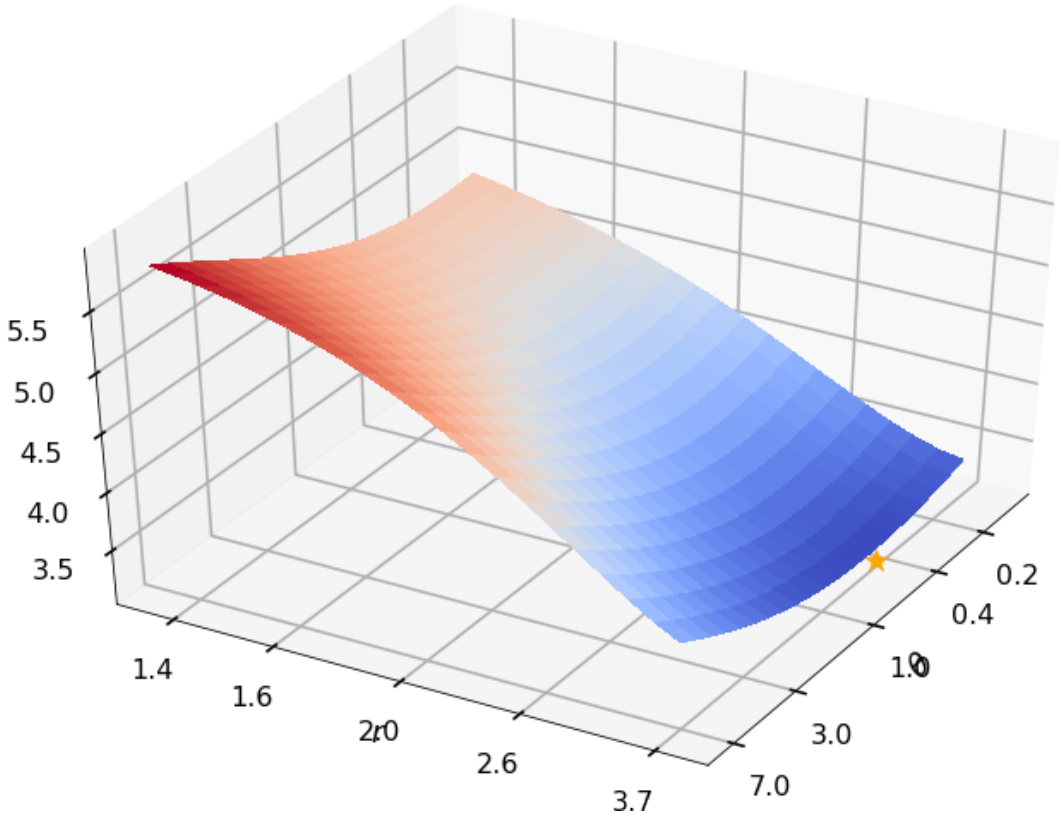

$d = 2, n = 64, r = 2.00, r_{opt} = 1.15, \theta = 1.00, \theta_{opt} = 1.69$
Data Quantiles
Standard Gaussian Quantiles
3
2
1
0
−1
−2
−3
−3
−2
−1
0
1
2
3

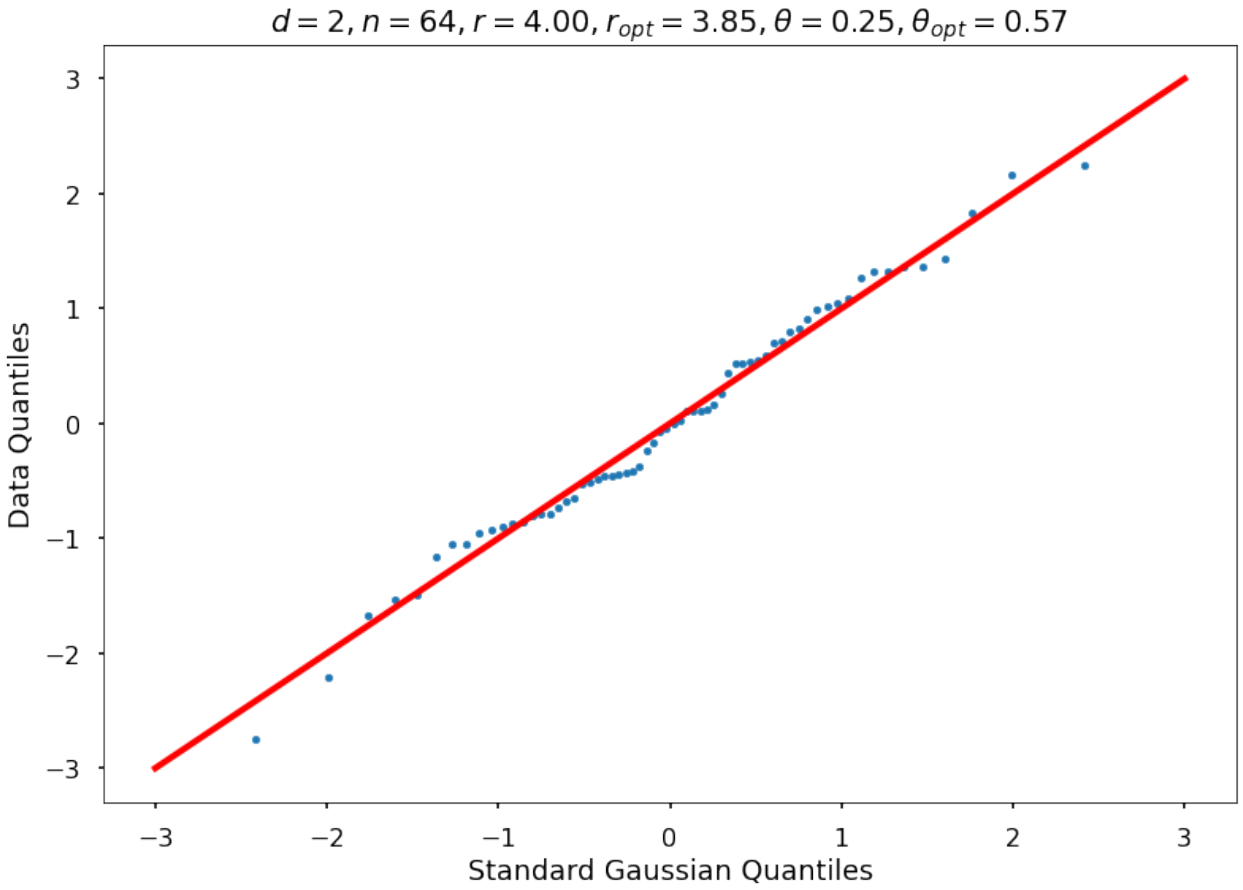

10.8
10.6
10.4
10.2
10.0
9.8
9.6
1.4
1.6
2.6
0.2
0.4
3.0
7.0

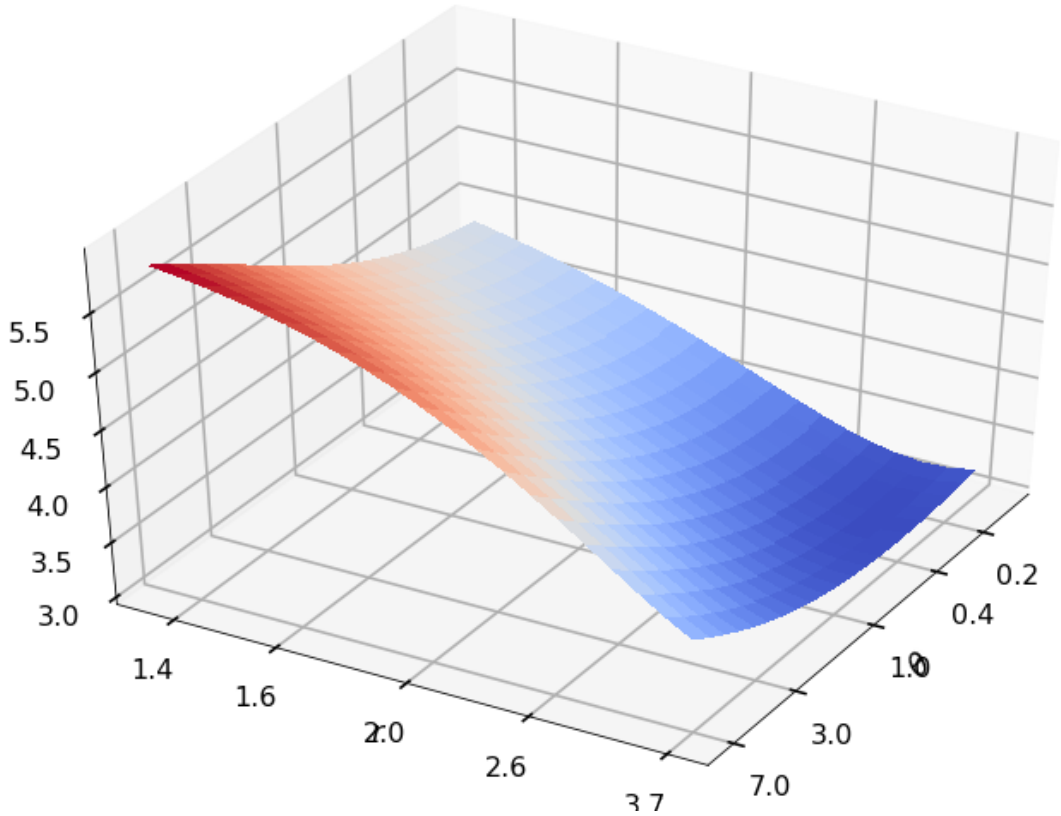

$d = 2, n = 64, r = 2.00, r_{opt} = 1.39, \theta = 1.00, \theta_{opt} = 0.87$
Data Quantiles
Standard Gaussian Quantiles
3
2
1
0
−1
−2
−3
−3
−2
−1
0
1
2
3

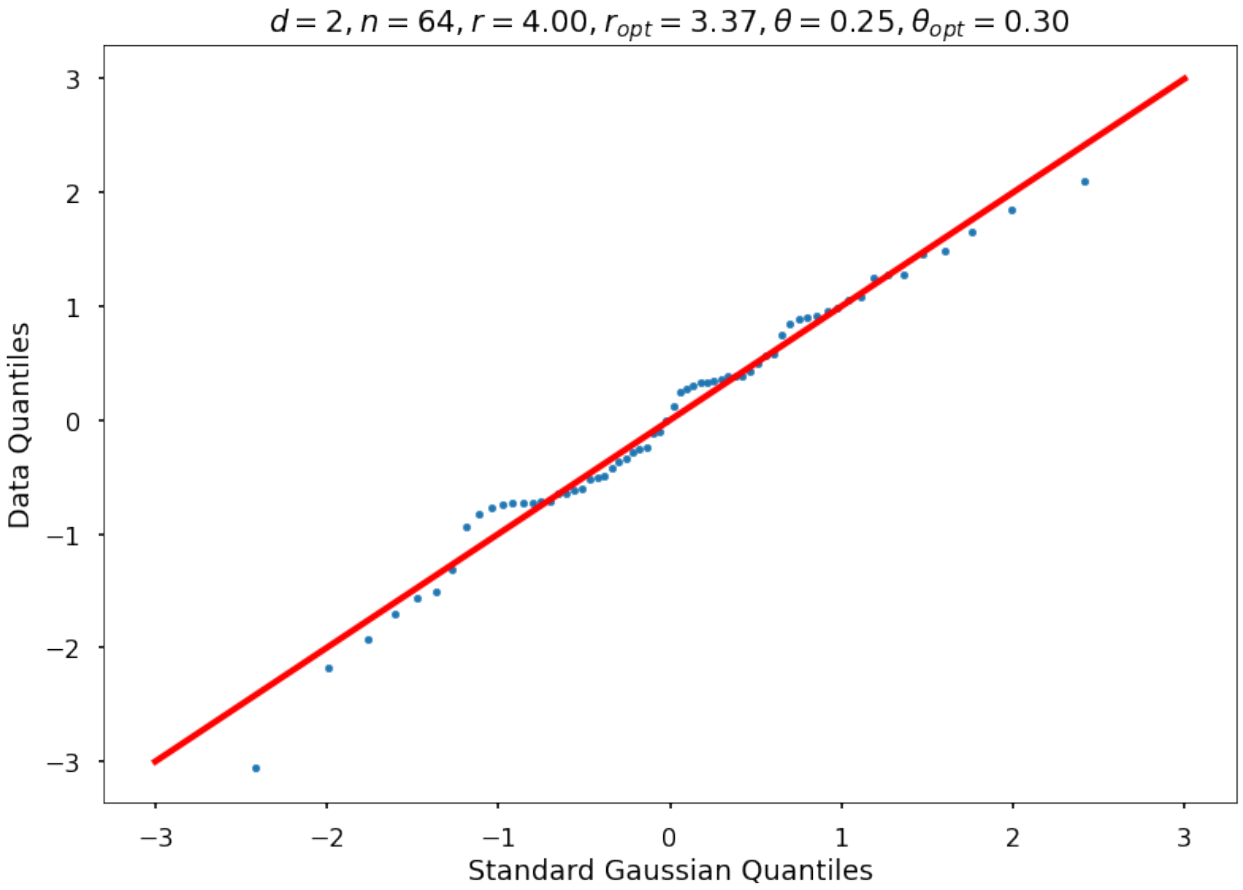

11.0
10.8
10.6
10.4
10.2
10.0
9.8
1.4
1.6
2.6
0.2
0.4
3.0
7.0

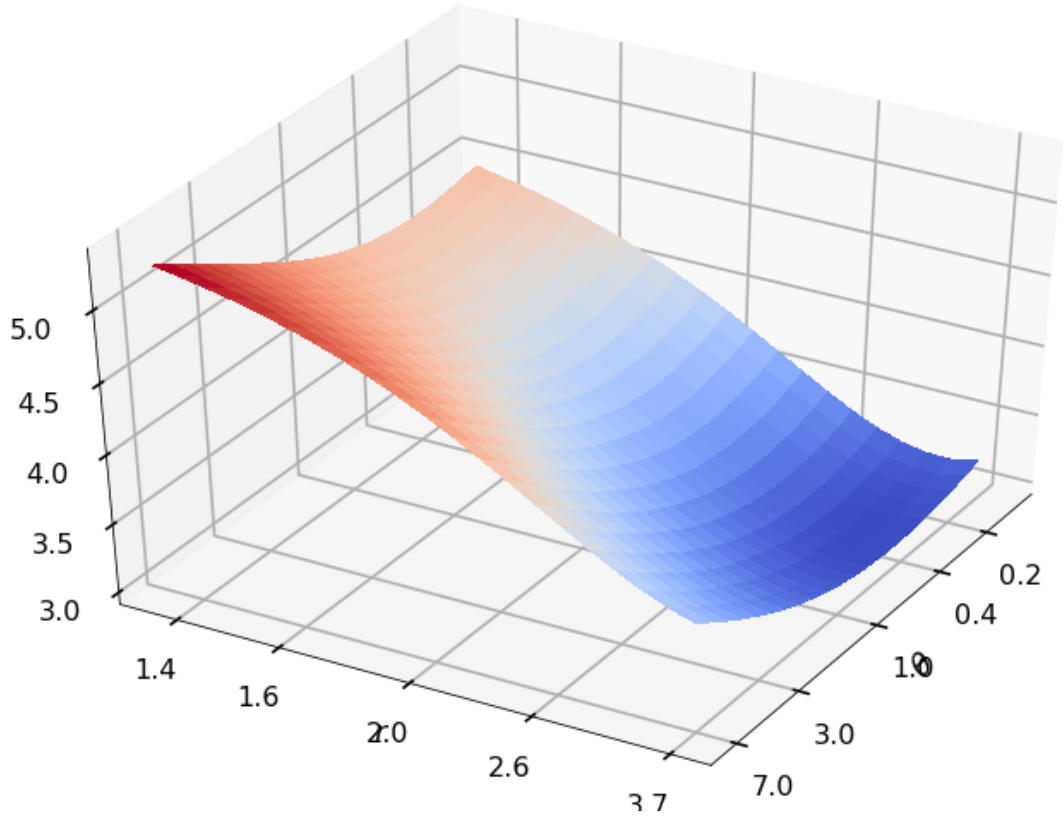

$d = 2, n = 64, r = 2.00, r_{opt} = 2.15, \theta = 1.00, \theta_{opt} = 3.91$
Data Quantiles
Standard Gaussian Quantiles
3
2
1
0
−1
−2
−3
−3
−2
−1
0
1
2
3

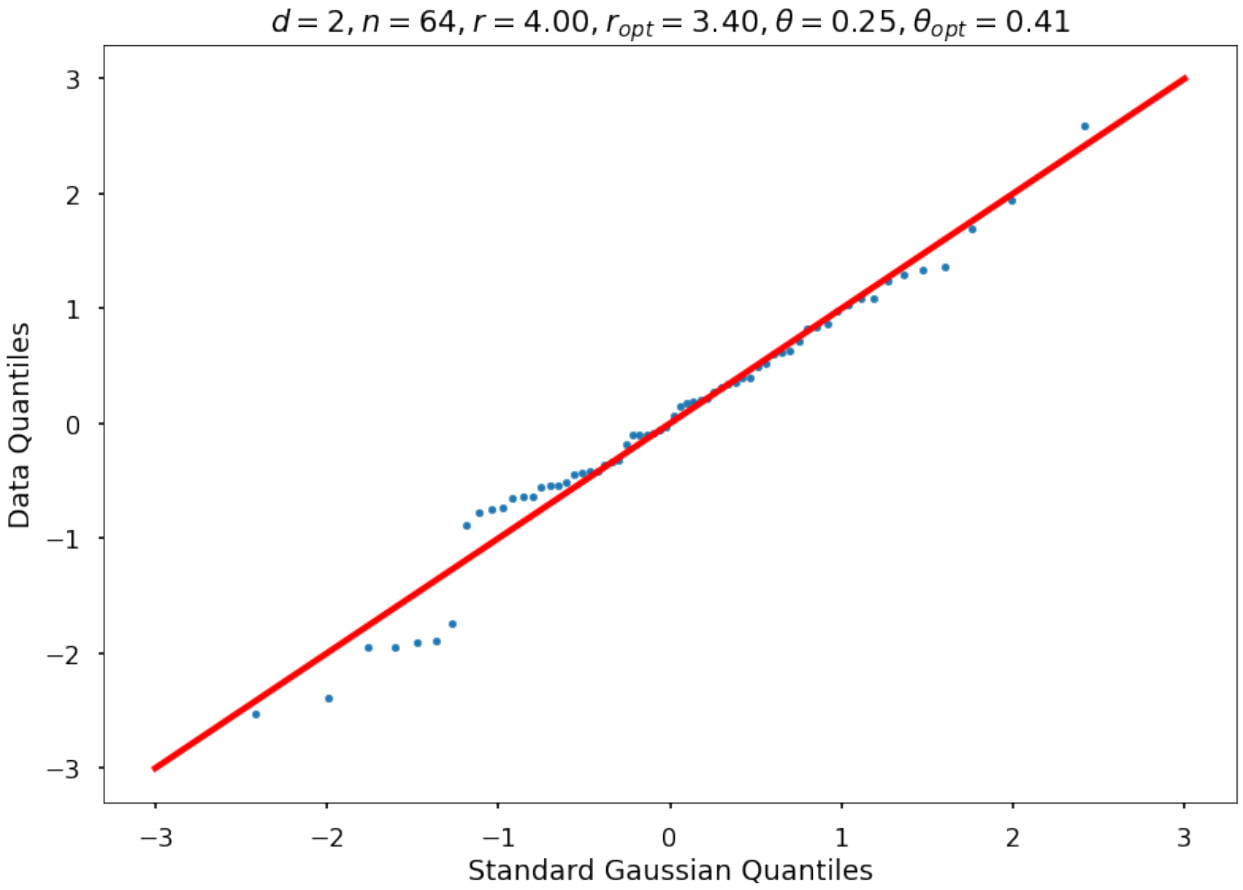

5.5
5.0
4.5
4.0
3.5
1.4
1.6
2.0
2.6
3.7
0.2
0.4
3.0
7.0

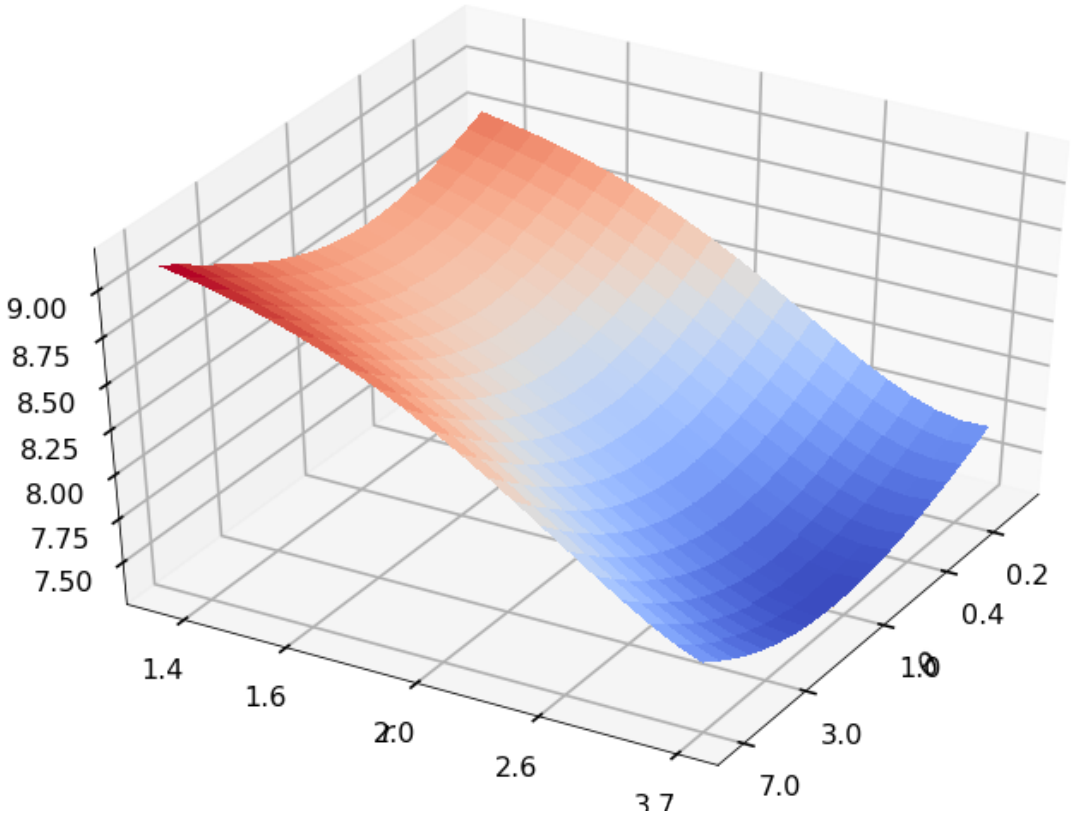

$d = 2, n = 64, r = 4.00, r_{opt} = 3.85, \theta = 0.25, \theta_{opt} = 0.57$
Data Quantiles
Standard Gaussian Quantiles
3
2
1
0
−1
−2
−3
−3
−2
−1
0
1
2
3

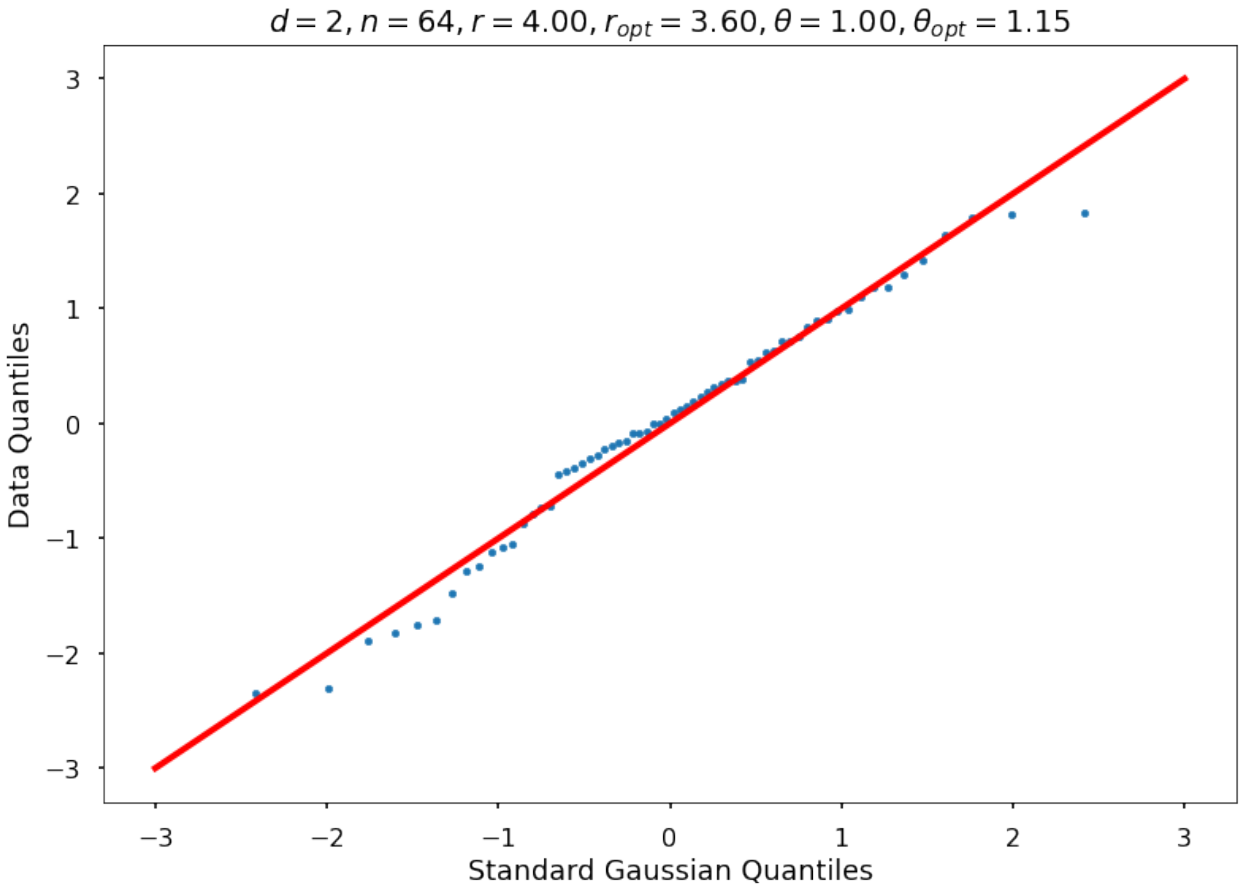

5.5
5.0
4.5
4.0
3.5
3.0
1.4
1.6
2.6
0.2
0.4
3.0
7.0

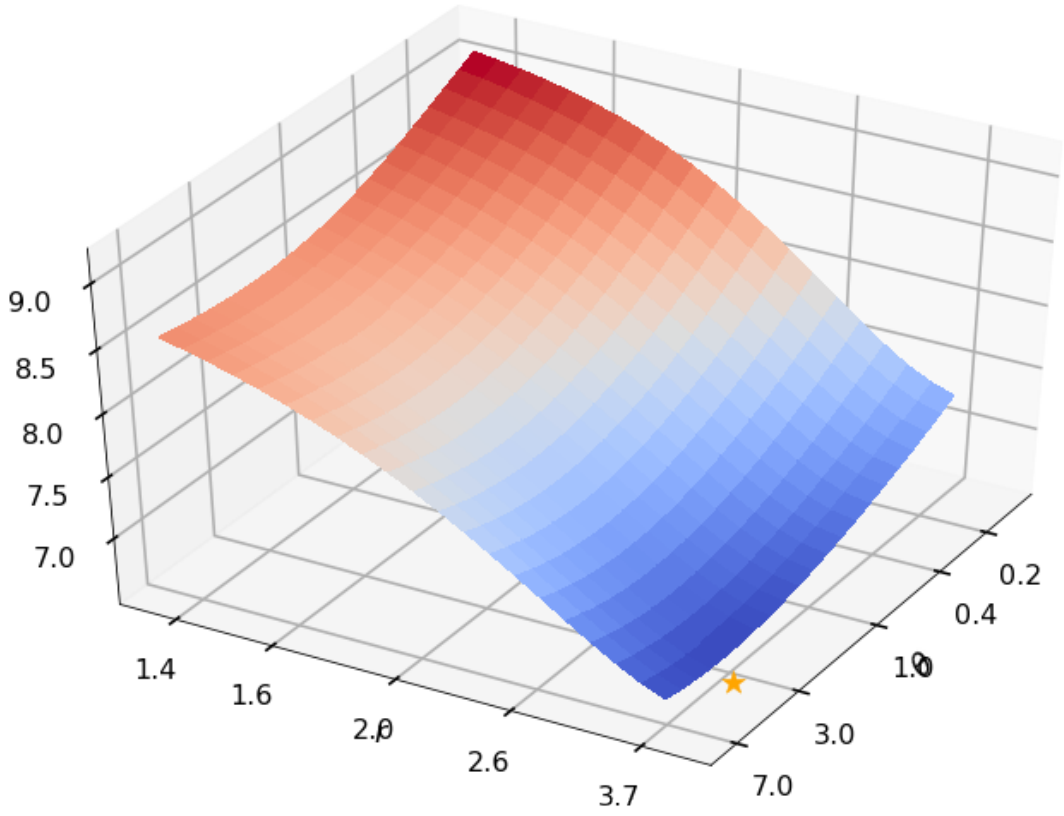

$d = 2, n = 64, r = 4.00, r_{opt} = 3.37, \theta = 0.25, \theta_{opt} = 0.30$
Data Quantiles
Standard Gaussian Quantiles
3
2
1
0
−1
−2
−3
−3
−2
−1
0
1
2
3

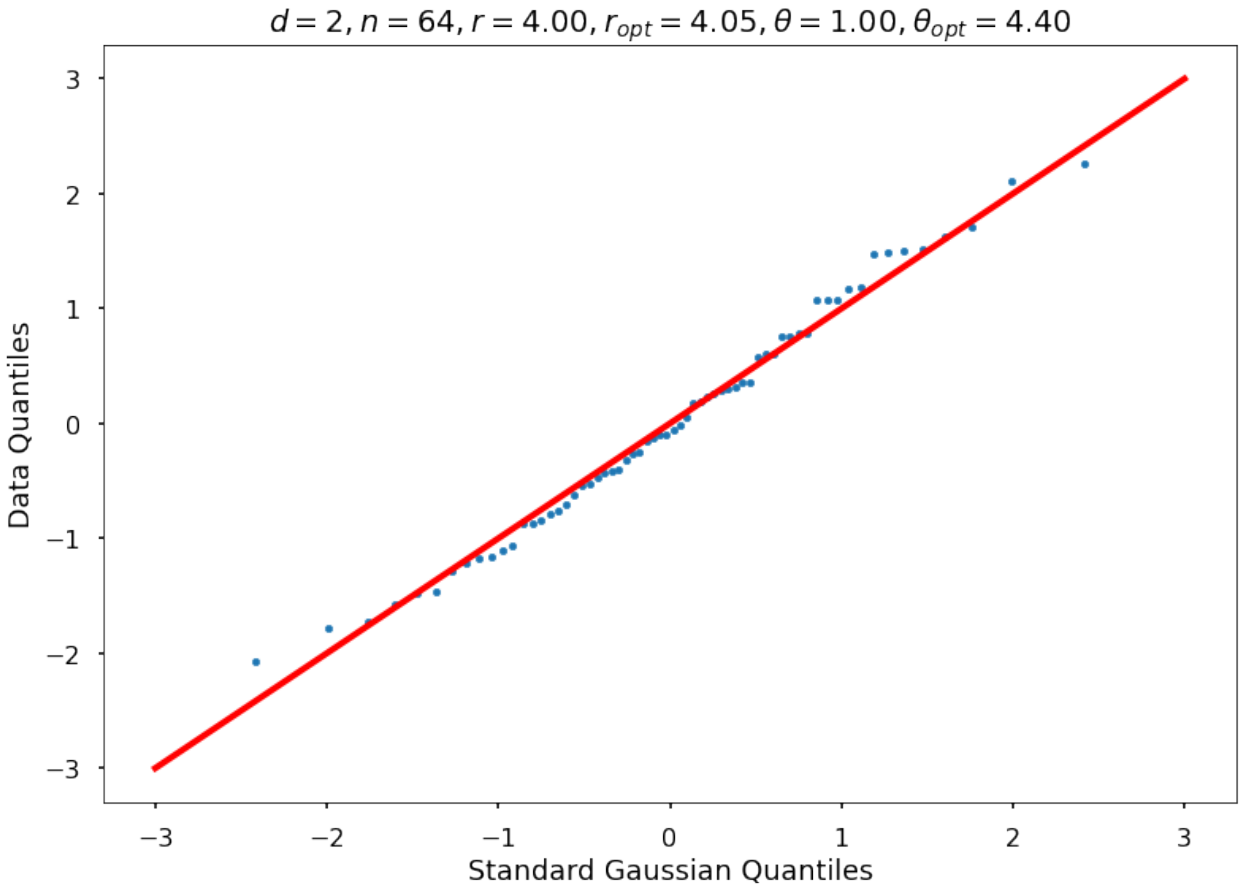

5.0
4.5
4.0
3.5
3.0
1.4
1.6
2.6
3.0
7.0
0.2
0.4

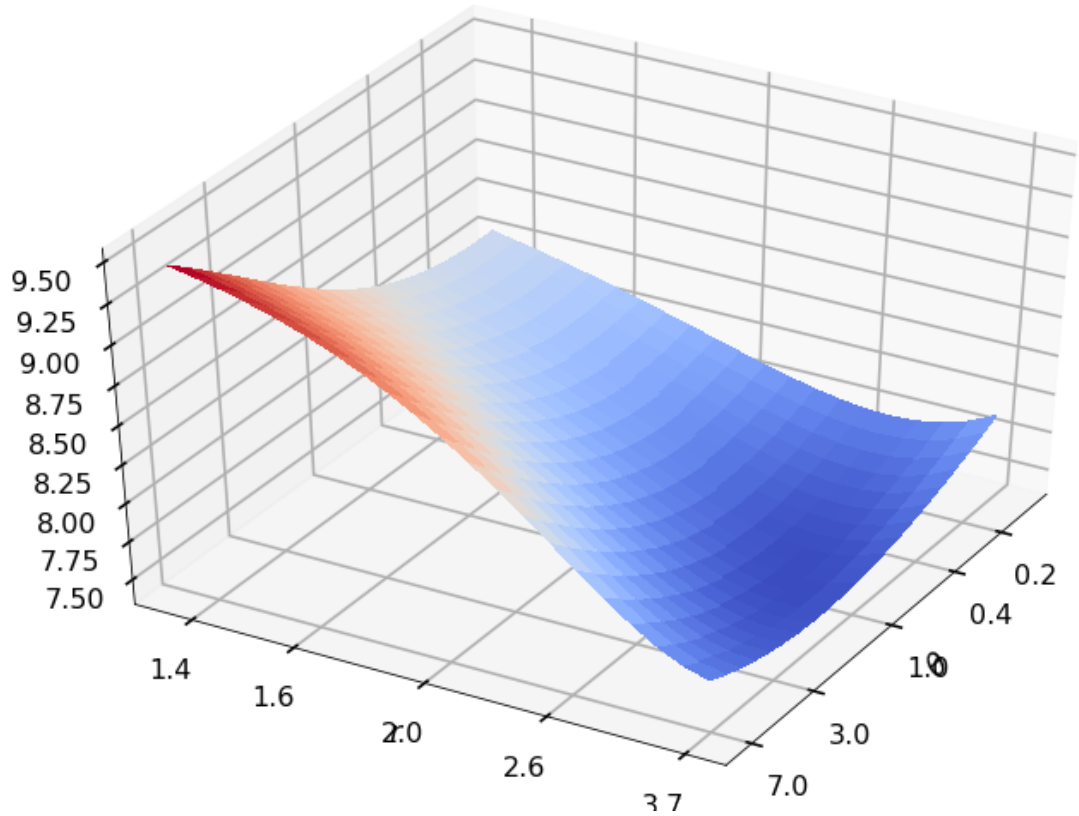

$d = 2, n = 64, r = 4.00, r_{opt} = 3.40, \theta = 0.25, \theta_{opt} = 0.41$
Data Quantiles
Standard Gaussian Quantiles
3
2
1
0
−1
−2
−3
−3
−2
−1
0
1
2
3

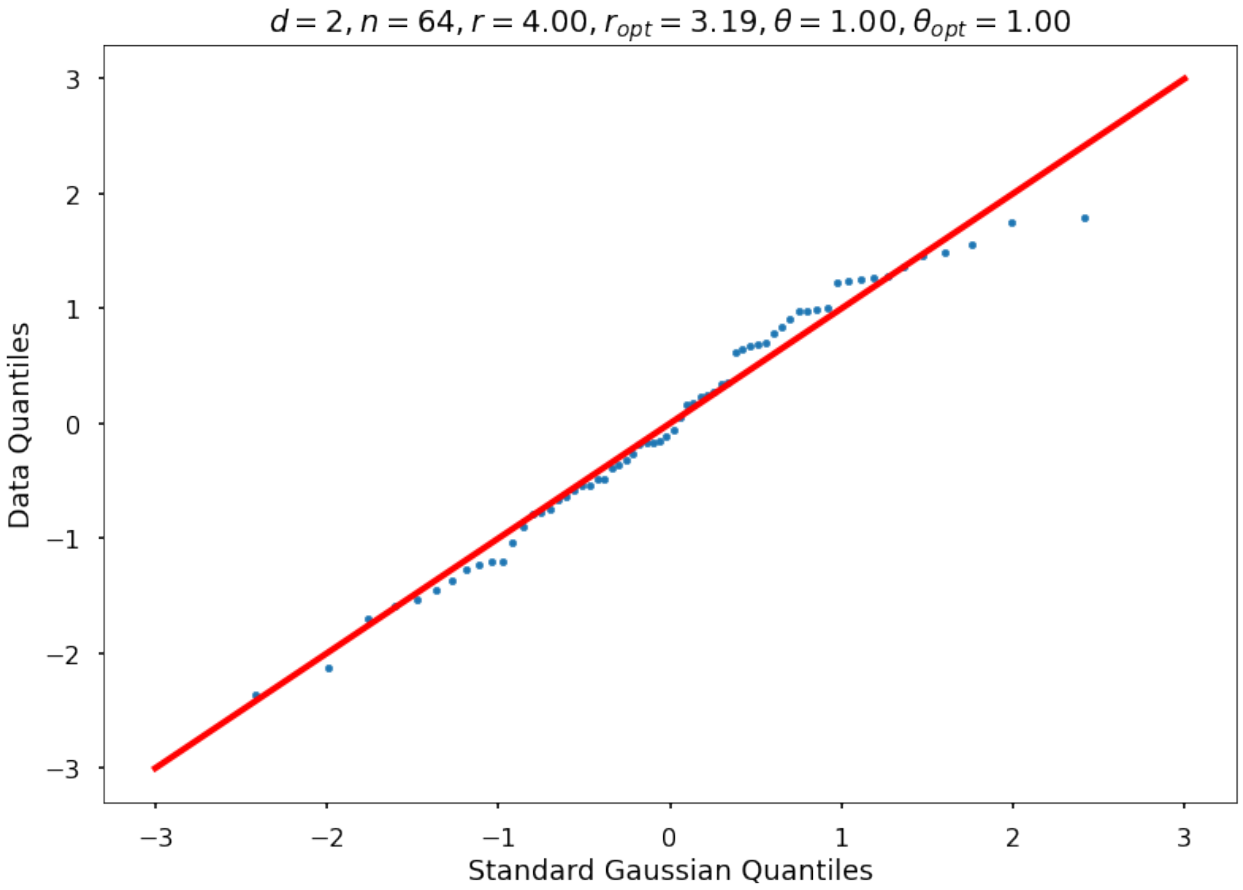

9.00
8.75
8.50
8.25
8.00
7.75
7.50
1.4
1.6
2.6
3.0
7.0
0.2
0.4

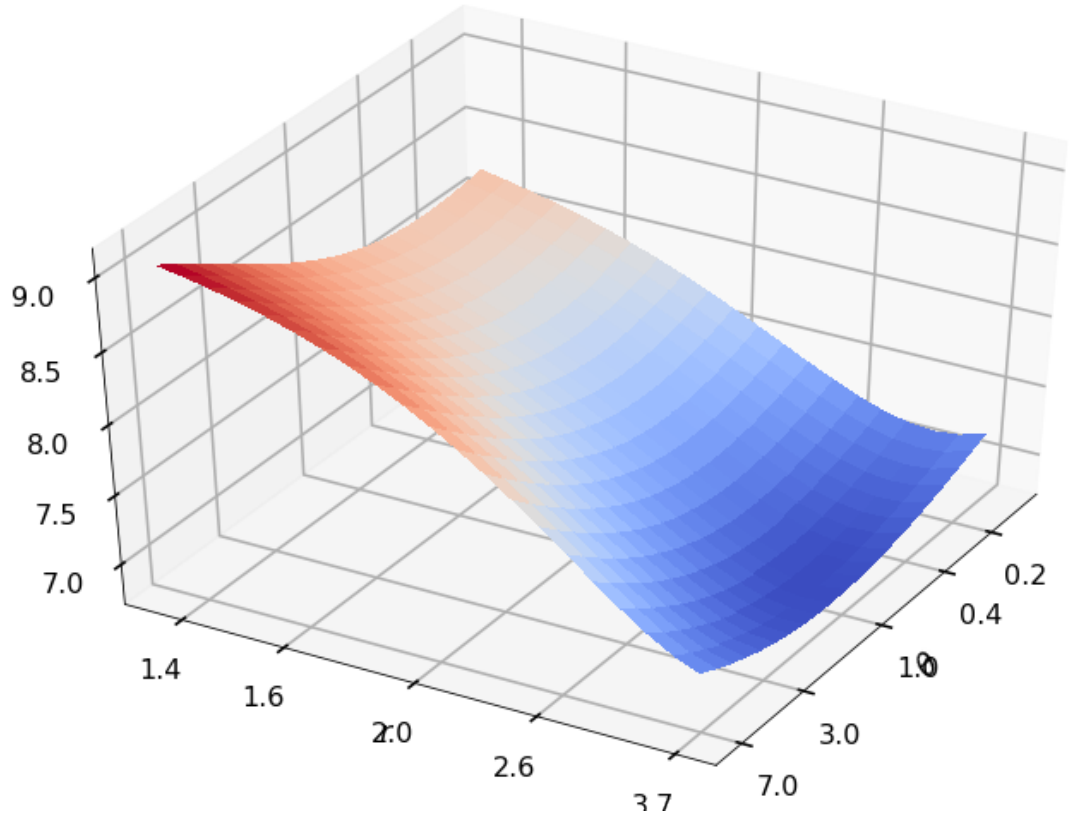

$d = 2, n = 64, r = 4.00, r_{opt} = 3.60, \theta = 1.00, \theta_{opt} = 1.15$
Data Quantiles
Standard Gaussian Quantiles
3
2
1
0
−1
−2
−3
−3
−2
−1
0
1
2
3

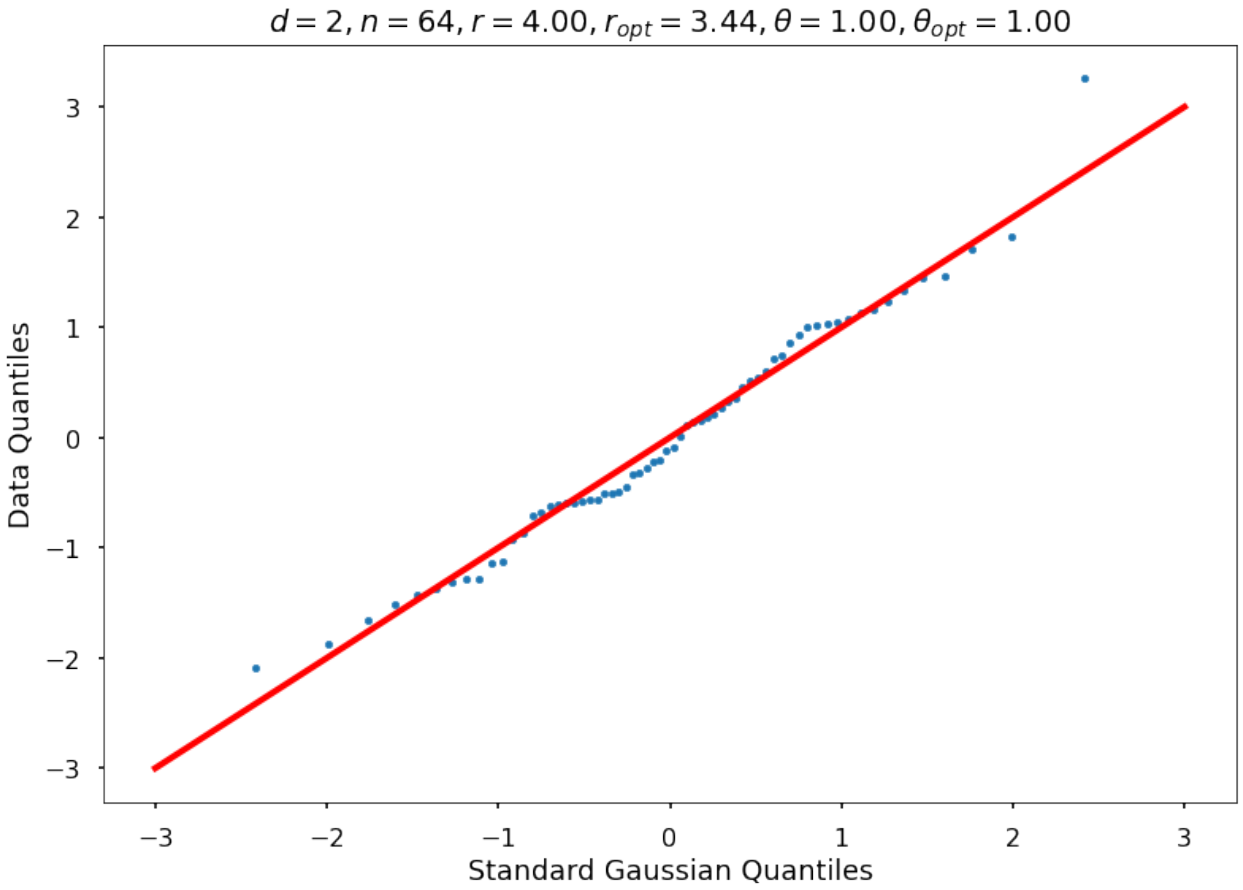

9.0
8.5
8.0
7.5
7.0
1.4
1.6
2.0
2.6
3.7
0.2
0.4
3.0
7.0

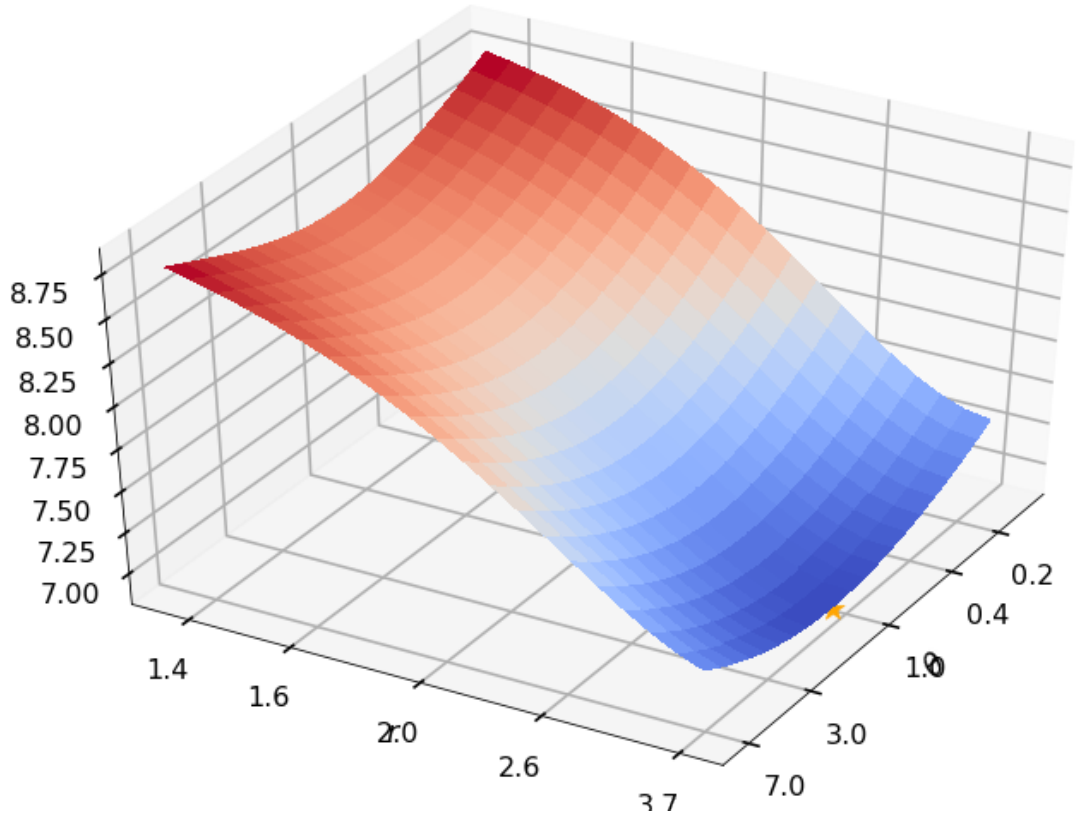

$d = 2, n = 64, r = 4.00, r_{opt} = 4.05, \theta = 1.00, \theta_{opt} = 4.40$
Data Quantiles
Standard Gaussian Quantiles
3
2
1
0
−1
−2
−3
−3
−2
−1
0
1
2
3

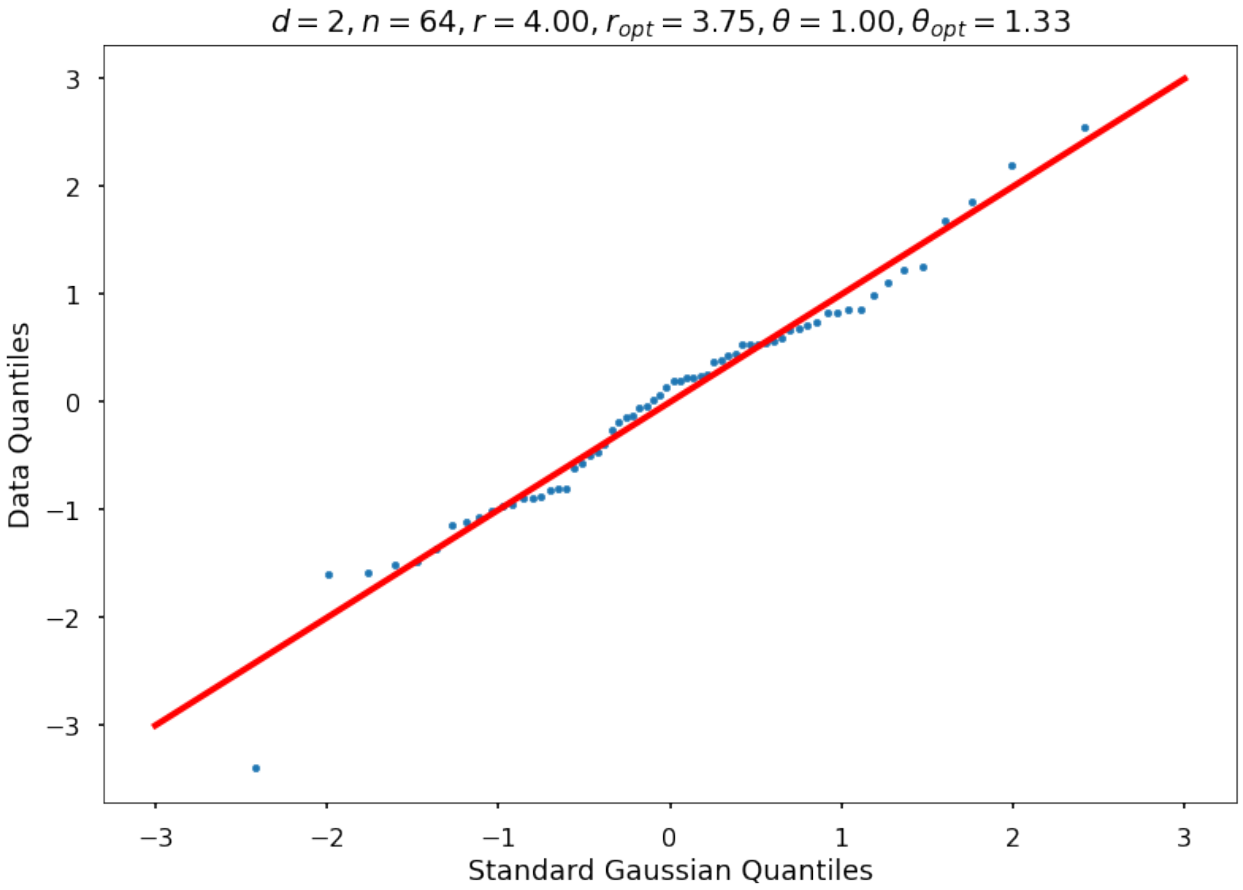

9.50
9.25
9.00
8.75
8.50
8.25
8.00
7.75
7.50
1.4
1.6
2.6
0.2
0.4
3.0
7.0

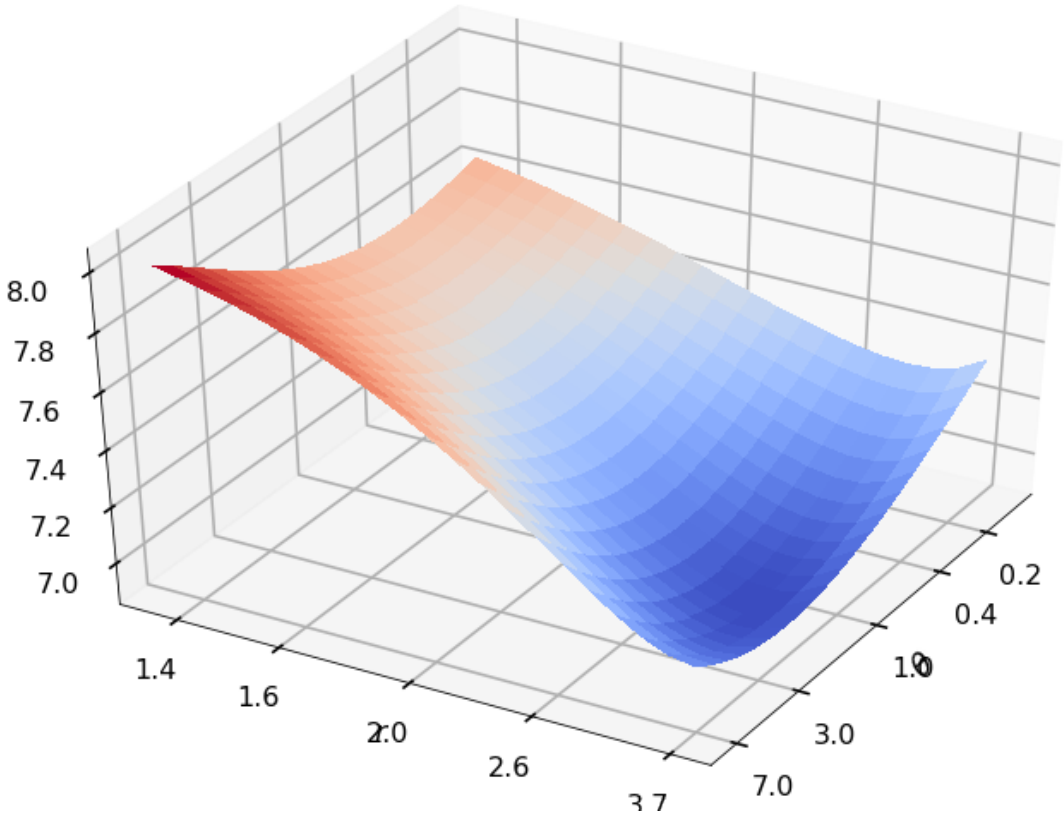

$d = 2, n = 64, r = 4.00, r_{opt} = 3.19, \theta = 1.00, \theta_{opt} = 1.00$
Data Quantiles
Standard Gaussian Quantiles
3
2
1
0
−1
−2
−3
−3
−2
−1
0
1
2
3

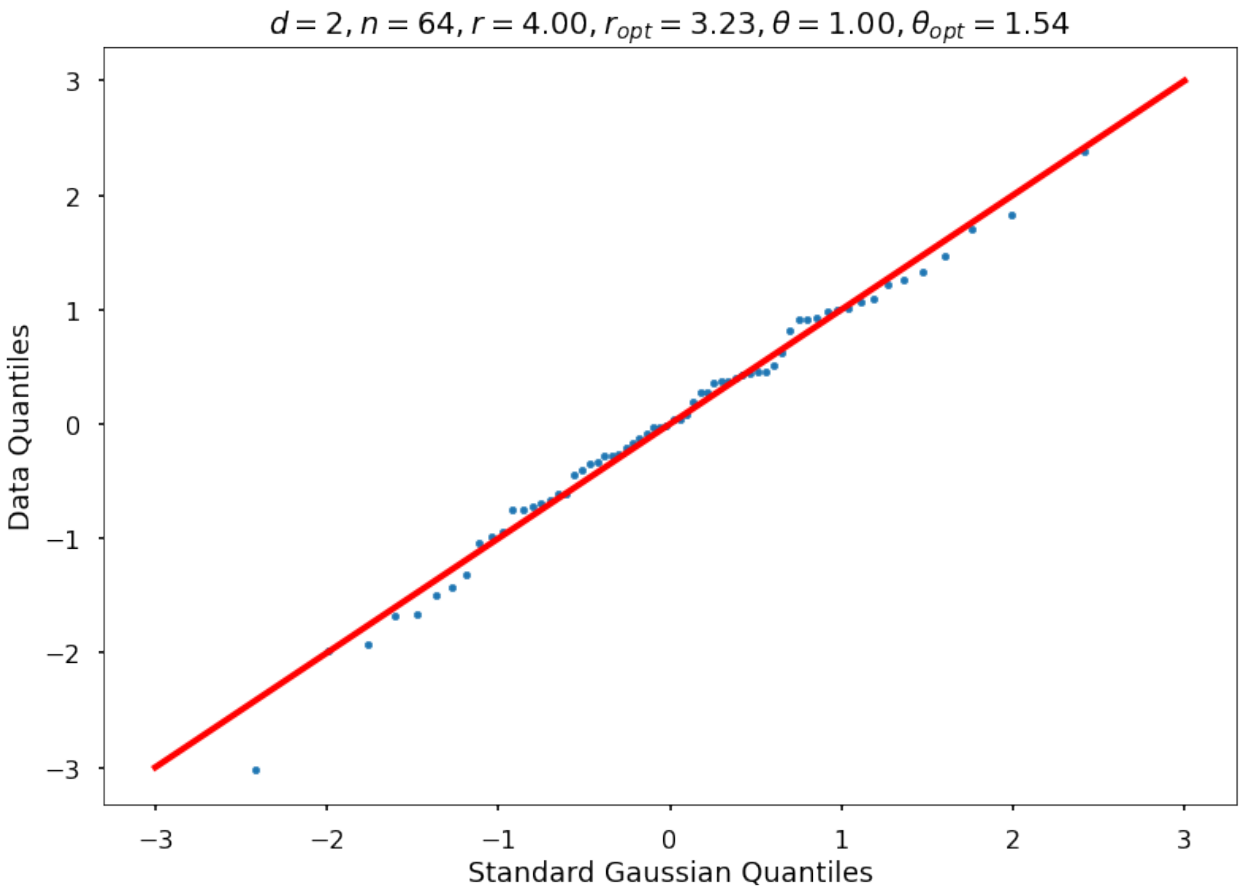

9.0
8.5
8.0
7.5
7.0
1.4
1.6
2.0
2.6
3.7
0.2
0.4
3.0
7.0


$d = 2, n = 64, r = 4.00, r_{opt} = 3.44, \theta = 1.00, \theta_{opt} = 1.00$
Data Quantiles
Standard Gaussian Quantiles
3
2
1
0
−1
−2
−3
−3
−2
−1
0
1
2
3


8.75
8.50
8.25
8.00
7.75
7.50
7.25
7.00
1.4
1.6
2.6
0.2
0.4
3.0
7.0


$d = 2, n = 64, r = 4.00, r_{opt} = 3.75, \theta = 1.00, \theta_{opt} = 1.33$
Data Quantiles
Standard Gaussian Quantiles
3
2
1
0
−1
−2
−3
−3
−2
−1
0
1
2
3

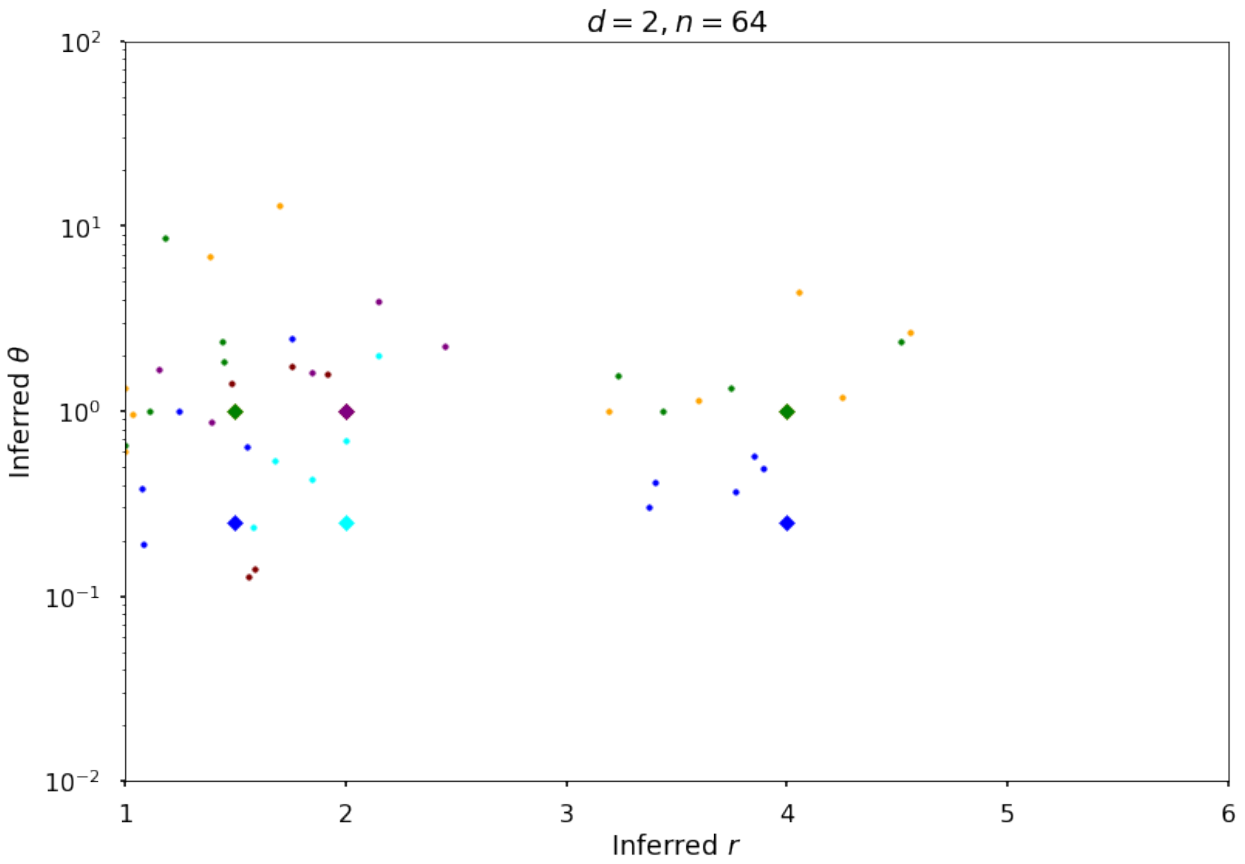

8.0
7.8
7.6
7.4
7.2
7.0
1.4
1.6
2.6
3.0
7.0
0.2
0.4




```
# close all the previous plots to freeup memory
plt.close('all')
```

Plot additional figures for random function

```
figH, axH = plt.subplots()
colorArray = ['blue', 'orange', 'green', 'cyan', 'maroon', 'purple']
nColArray = len(colorArray)
for jjj in range(nrArr):
    for kkk in range(nfPArr):
        clrInd = np.mod(nfPArr * (jjj) + kkk, nColArray)
        clr = colorArray[clrInd]
        axH.scatter(rOptAll[jjj, kkk, :].reshape((nRep, 1)), thOptAll[jjj, kkk, :].
↪reshape((nRep, 1)),
                    s=50, c=clr, marker='.')
        axH.scatter(rArray[jjj], thetaAll[jjj, kkk], s=50, c=clr, marker='D')

axH.set(xlim=[1, 6], ylim=[0.01, 100])
axH.set_yscale('log')
axH.set_title(f'$d = {dim}, n = {npts}$')
axH.set_xlabel('Inferred $r$')
axH.set_ylabel('Inferred $\\theta$')
figH.savefig(f'{fName}-rthInfer-n-{npts}-d-{dim}.jpg')
```




```
# close all the previous plots to freeup memory
plt.close('all')
```

5.18.3 Example 3a: Keister integrand: npts = 64

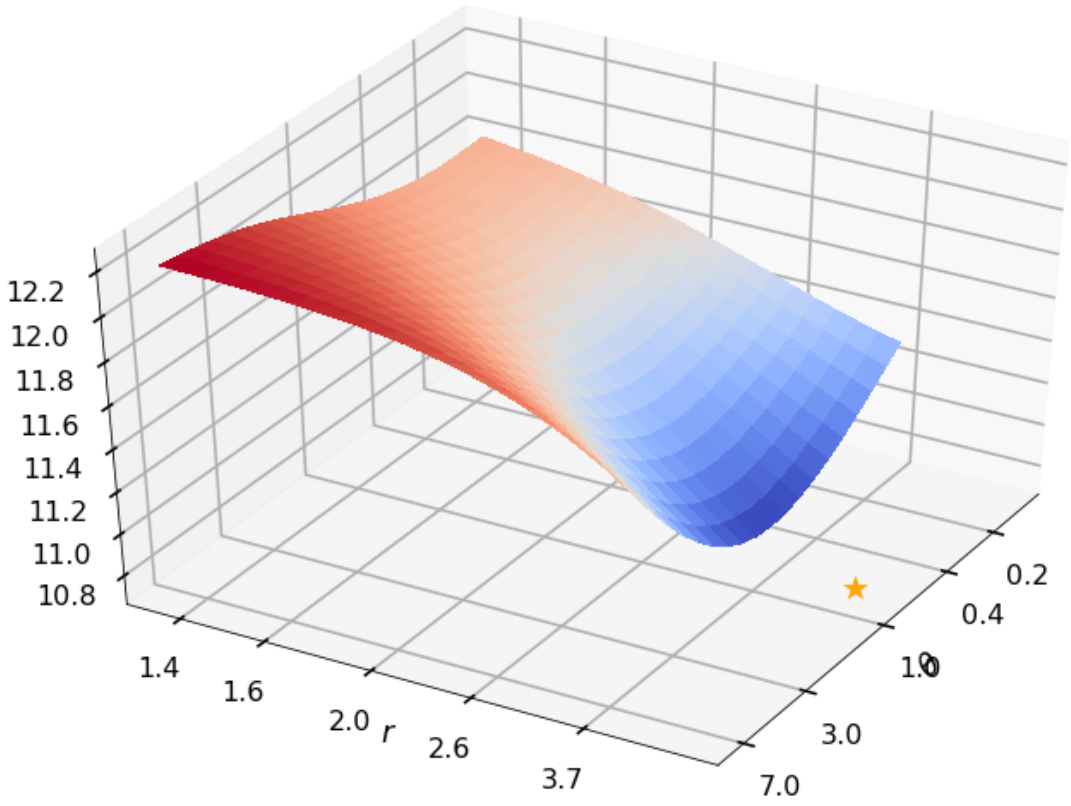
```
## Keister example
fwh = 2
dim = 3
npts = 2 ** 6
nRep = 20
nPlot = 2
_, rOptAll, thOptAll, fName = gaussian_diagnostics_engine(fwh, dim, npts, None, None,␣
↪nRep, nPlot)

## Plot Keister example
figH = plt.figure()
plt.scatter(rOptAll, thOptAll, s=20, color='blue')
# axis([4 6 0.5 1.5])
# set(gca,'yscale','log')
plt.xlabel('Inferred $r$')
plt.ylabel('Inferred $\\theta$')
plt.title(f'$d = {dim}, n = {npts}$')
figH.savefig(f'{fName}-rthInfer-n-{npts}-d-{dim}.jpg')
```

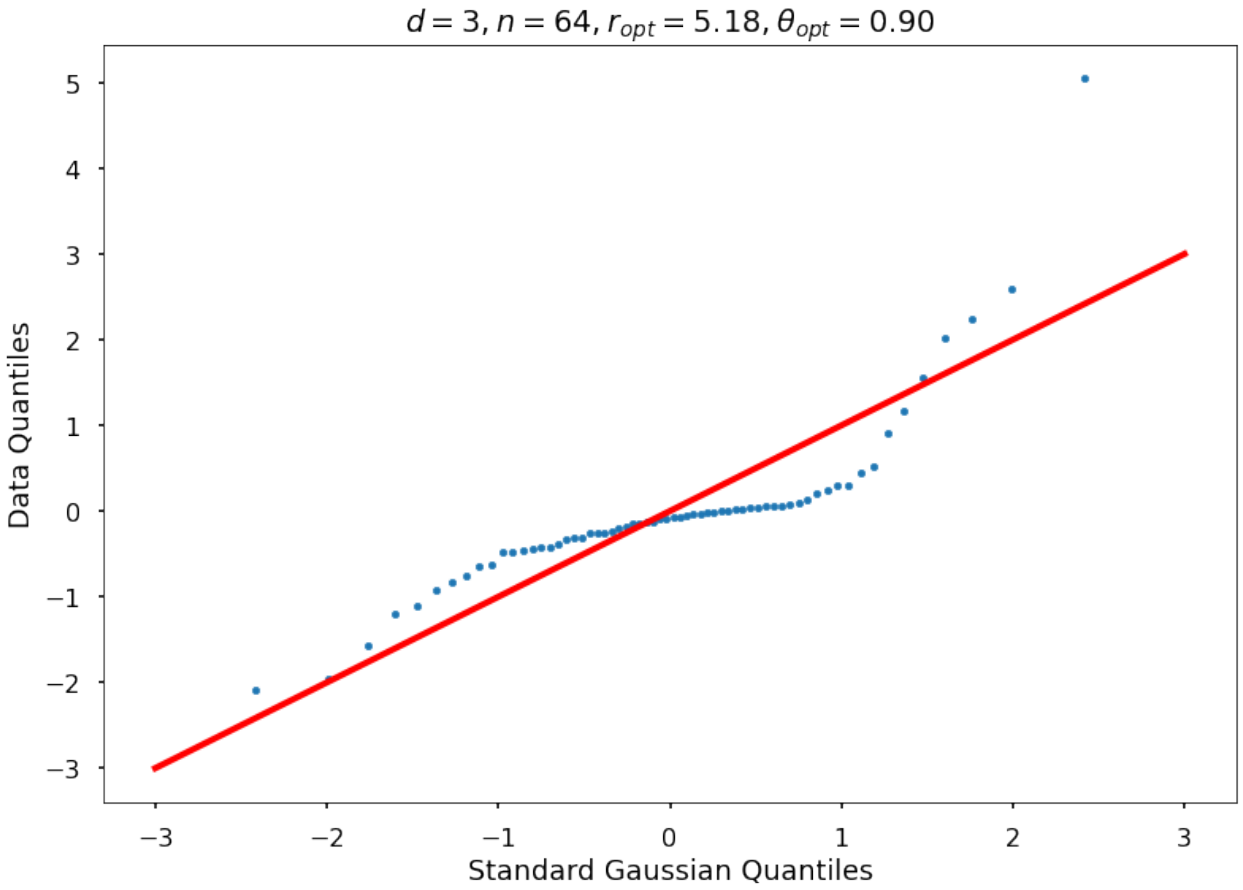
```
10.918373367804714
10.74245342720698
r = None, rOpt = 5.17845, theta = None, thetaOpt = 0.90000

10.717466826946536
10.562731448359418
r = None, rOpt = 5.10797, theta = None, thetaOpt = 0.75631

10.688455595826237
10.539160528442594
r = None, rOpt = 5.02081, theta = None, thetaOpt = 0.73583

10.874548555226458
10.610842354541466
r = None, rOpt = 5.63220, theta = None, thetaOpt = 0.90923

10.740805174948838
10.557754768898818
r = None, rOpt = 5.19574, theta = None, thetaOpt = 0.77893

10.928140146153147
10.720109505682482
r = None, rOpt = 5.43883, theta = None, thetaOpt = 0.81368

10.936776084649589
10.748392943135848
r = None, rOpt = 5.32207, theta = None, thetaOpt = 0.91872

10.866161104206988
10.690655913547836
r = None, rOpt = 5.14400, theta = None, thetaOpt = 0.92759

10.613489865692868
10.45889197376518
r = None, rOpt = 5.08381, theta = None, thetaOpt = 0.61740

10.636181168972138
10.467815857368016
r = None, rOpt = 5.16286, theta = None, thetaOpt = 0.69577

10.728895335365538
10.585065856399083
r = None, rOpt = 5.05675, theta = None, thetaOpt = 0.72234

11.031579085996949
10.889786816011357
r = None, rOpt = 5.05532, theta = None, thetaOpt = 0.91526

10.866073731336623
10.548636876696989
r = None, rOpt = 5.88962, theta = None, thetaOpt = 0.94453

10.898763849532882
```

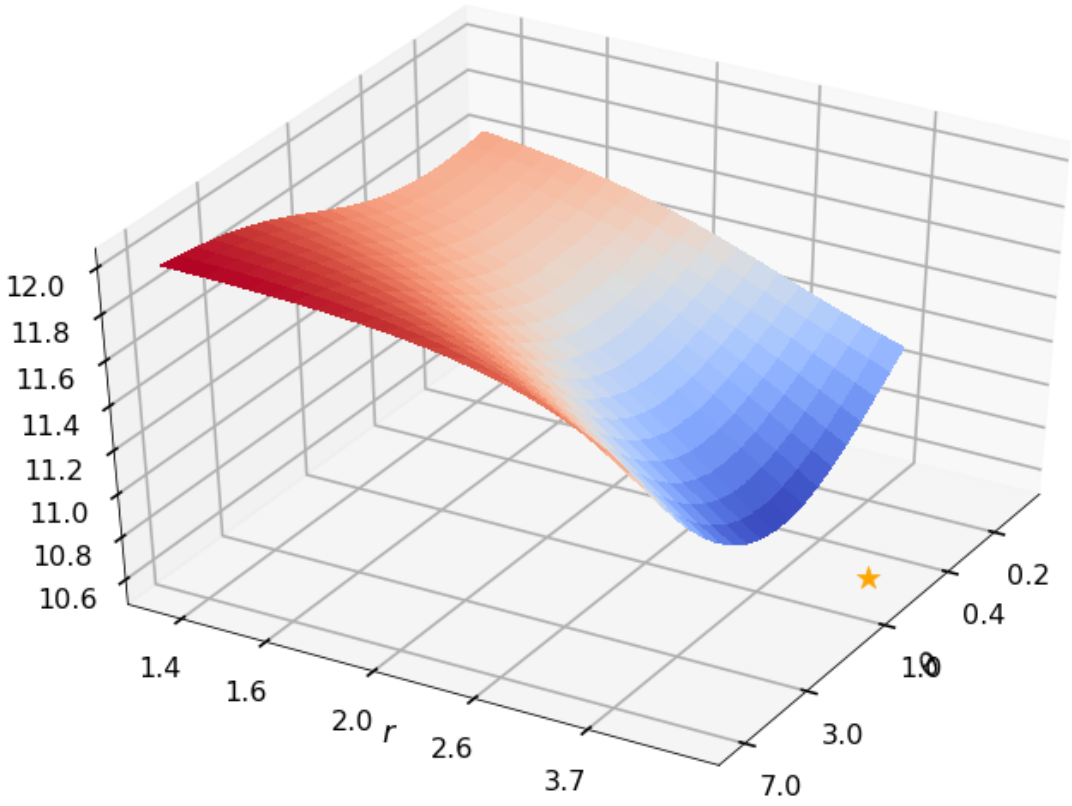
(continues on next page)

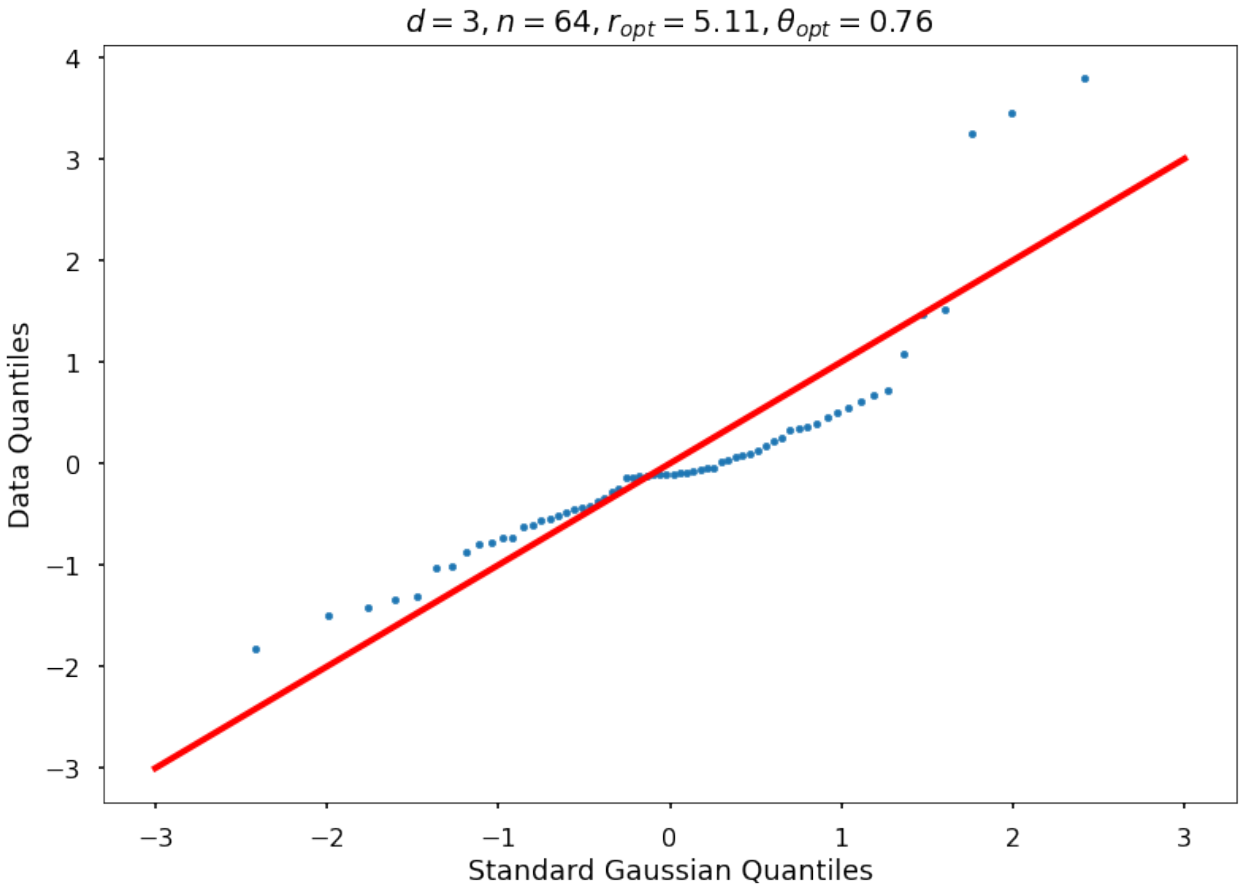
(continued from previous page)

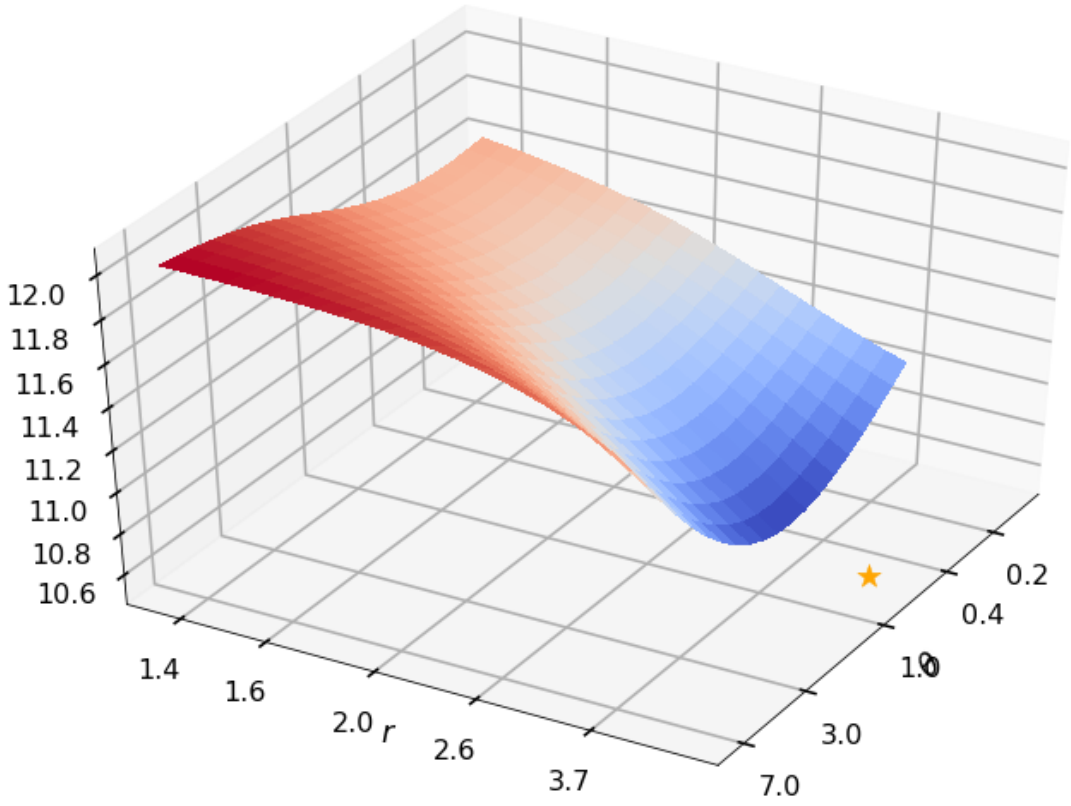
```
10.7428251348116
r = None, rOpt = 5.07111, theta = None, thetaOpt = 0.93066

11.04144749198928
10.892511325588508
r = None, rOpt = 5.10849, theta = None, thetaOpt = 0.91039

10.771539976900868
10.602263037779617
r = None, rOpt = 5.22262, theta = None, thetaOpt = 0.81070

10.919713159295926
10.703969564045602
r = None, rOpt = 5.38652, theta = None, thetaOpt = 0.92338

10.775915237177964
10.646594662551685
r = None, rOpt = 5.01369, theta = None, thetaOpt = 0.66747

11.02928162756276
10.874074219278276
r = None, rOpt = 5.12990, theta = None, thetaOpt = 0.92172

10.87752870870672
10.685153898271022
r = None, rOpt = 5.34982, theta = None, thetaOpt = 0.77387
```

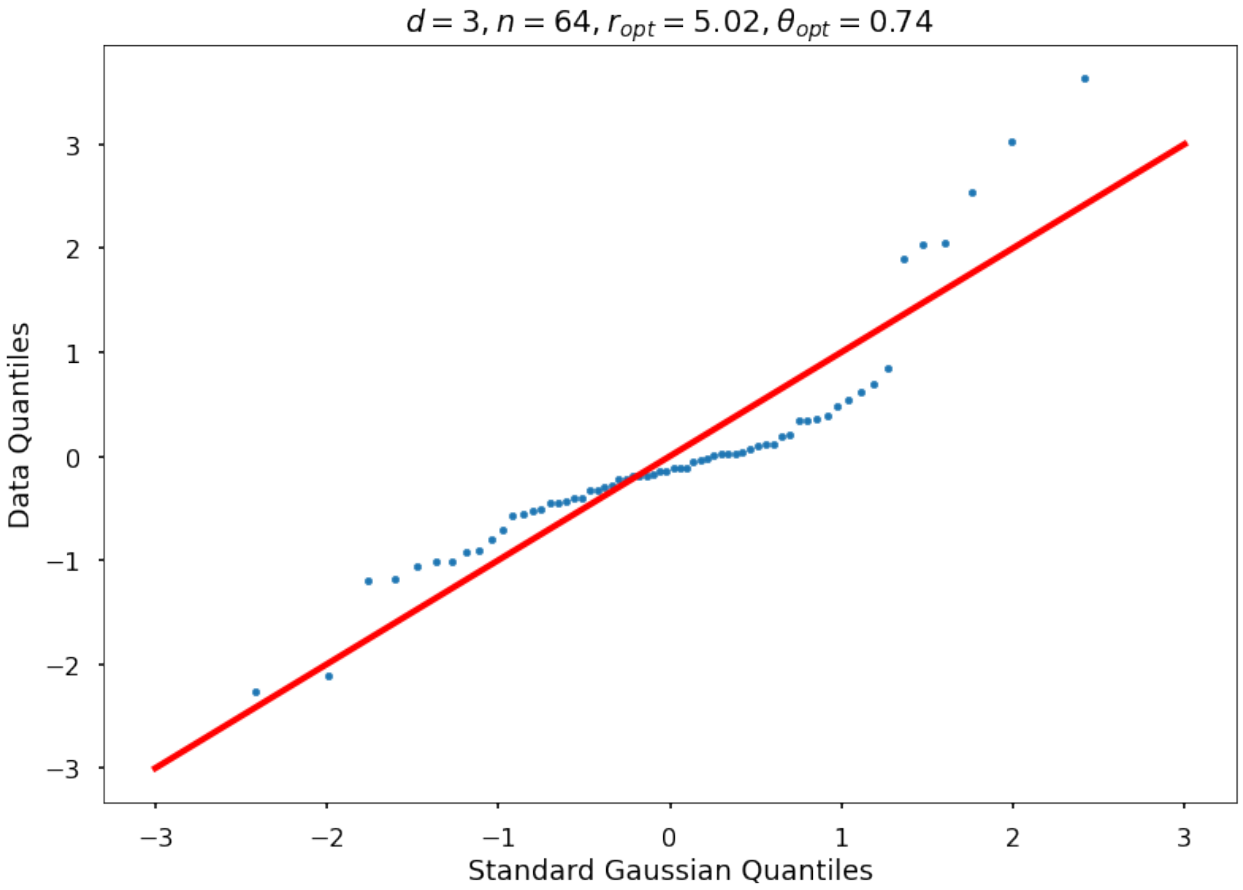

12.2
12.0
11.8
11.6
11.4
11.2
11.0
10.8
1.4
1.6
2.0
r
2.6
3.7
0.2
0.4
3.0
7.0

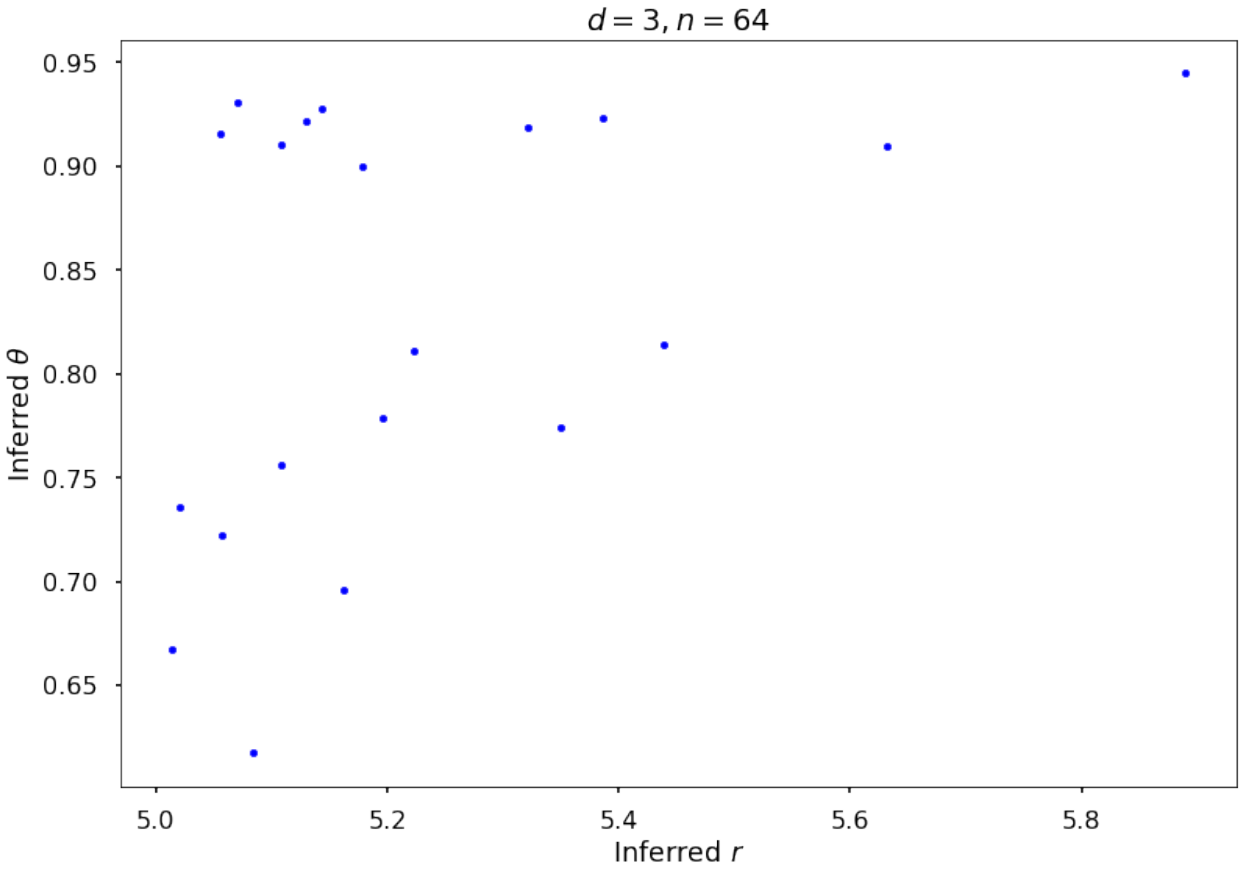

$d = 3, n = 64, r_{opt} = 5.18, \theta_{opt} = 0.90$
Data Quantiles
Standard Gaussian Quantiles
5
4
3
2
1
0
−1
−2
−3
−3
−2
−1
0
1
2
3


12.0
11.8
11.6
11.4
11.2
11.0
10.8
10.6
1.4
1.6
2.0
r
2.6
3.7
0.2
0.4
3.0
7.0


$d = 3, n = 64, r_{opt} = 5.11, \theta_{opt} = 0.76$
Data Quantiles
Standard Gaussian Quantiles
4
3
2
1
0
−1
−2
−3
−3
−2
−1
0
1
2
3


12.0
11.8
11.6
11.4
11.2
11.0
10.8
10.6
1.4
1.6
2.0
r
2.6
3.7
0.2
0.4
3.0
7.0


$d = 3, n = 64, r_{opt} = 5.02, \theta_{opt} = 0.74$
Data Quantiles
Standard Gaussian Quantiles
3
2
1
0
−1
−2
−3
−3
−2
−1
0
1
2
3



5.18.4 Example 3b: Keister integrand: npts = 1024

```
## Keister example
fwh = 2
dim = 3
npts = 2 ** 10
nRep = 20
nPlot = 2
_, rOptAll, thOptAll, fName = gaussian_diagnostics_engine(fwh, dim, npts, None, None,␣
↪nRep, nPlot)

## Plot Keister example
figH = plt.figure()
plt.scatter(rOptAll, thOptAll, s=20, color='blue')
# axis([4 6 0.5 1.5])
# set(gca,'yscale','log')
plt.xlabel('Inferred $r$')
plt.ylabel('Inferred $\\theta$')
plt.title(f'$d = {dim}, n = {npts}$')
figH.savefig(f'{fName}-rthInfer-n-{npts}-d-{dim}.jpg')
```

```
10.596244143639485
7.142978946028139
```

(continues on next page)

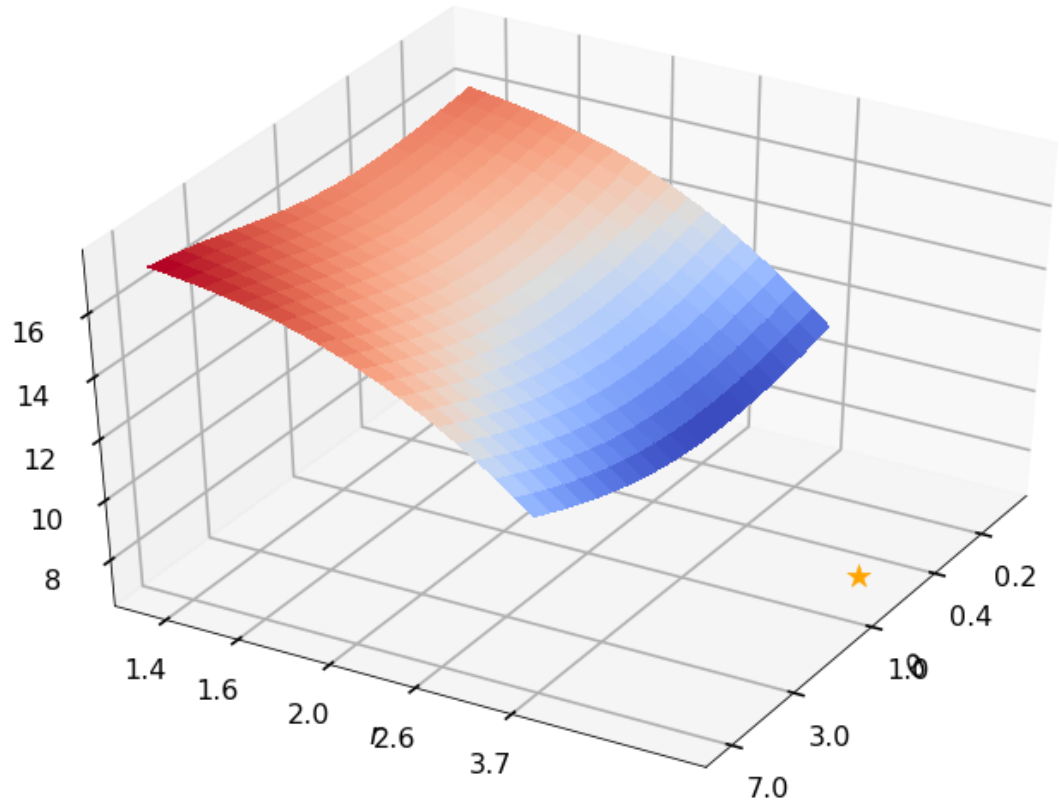
(continued from previous page)

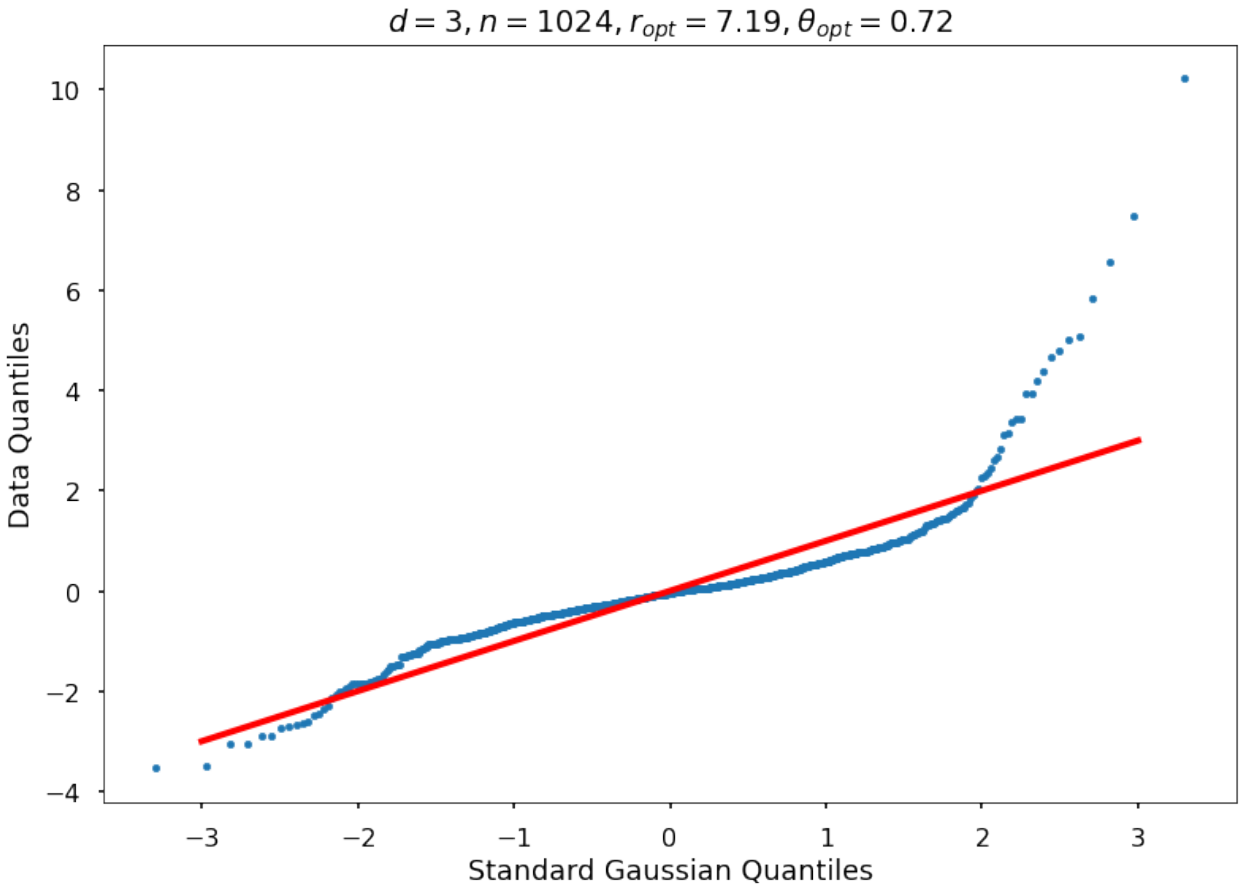
```
r = None, rOpt = 7.18724, theta = None, thetaOpt = 0.71553

10.594856691086207
7.096409091118618
r = None, rOpt = 7.26586, theta = None, thetaOpt = 0.76520

10.591375134255042
7.0166323209172194
r = None, rOpt = 7.34554, theta = None, thetaOpt = 0.74504

10.595755571171377
7.067858016018031
r = None, rOpt = 7.34121, theta = None, thetaOpt = 0.75696

10.595982354998483
7.1228385587643395
r = None, rOpt = 7.22718, theta = None, thetaOpt = 0.70060

10.596298140666004
7.104717917027118
r = None, rOpt = 7.27766, theta = None, thetaOpt = 0.71703

10.592187612685716
7.0859774608781745
r = None, rOpt = 7.26092, theta = None, thetaOpt = 0.75369

10.596120527059021
7.10600296848717
r = None, rOpt = 7.25766, theta = None, thetaOpt = 0.69220

10.594256056513693
7.105374164104877
r = None, rOpt = 7.25124, theta = None, thetaOpt = 0.73247

10.596614398709978
7.137622143665308
r = None, rOpt = 7.20010, theta = None, thetaOpt = 0.70688

10.598230772383527
7.186852606944651
r = None, rOpt = 7.13933, theta = None, thetaOpt = 0.69409

10.597856894192905
7.126442092925792
r = None, rOpt = 7.27342, theta = None, thetaOpt = 0.72154

10.595526689101934
7.101590379268407
r = None, rOpt = 7.26605, theta = None, thetaOpt = 0.72566

10.593990233649055
7.128838039518072
```

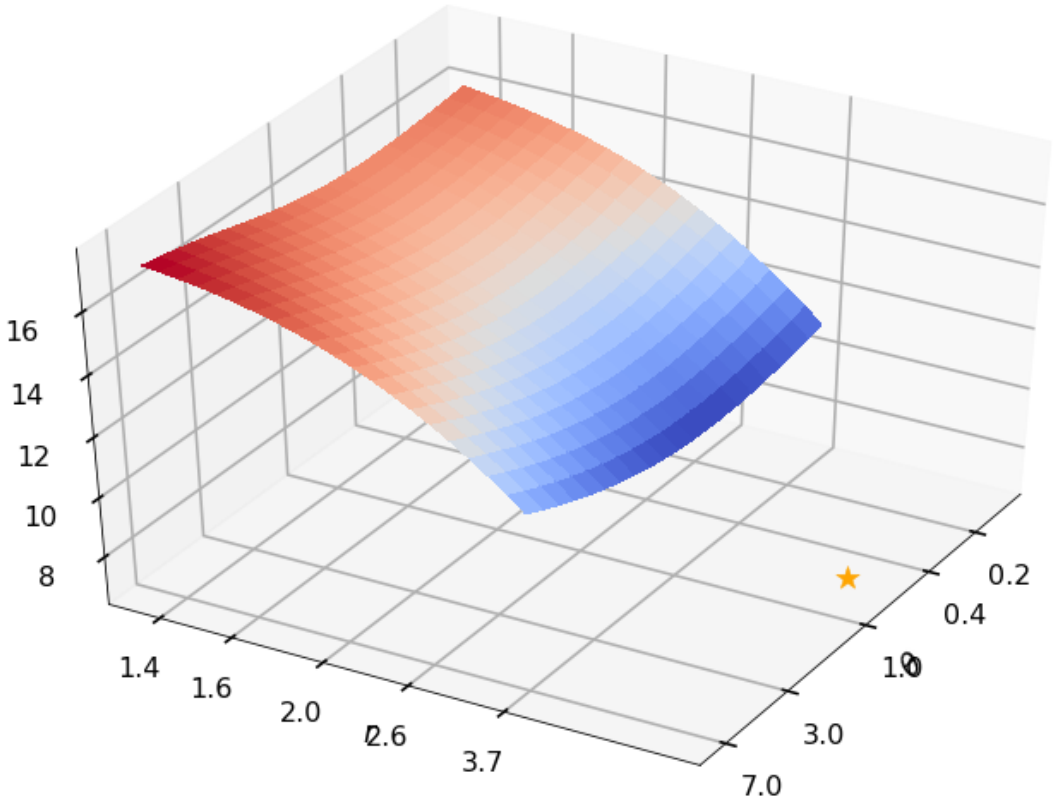
(continues on next page)

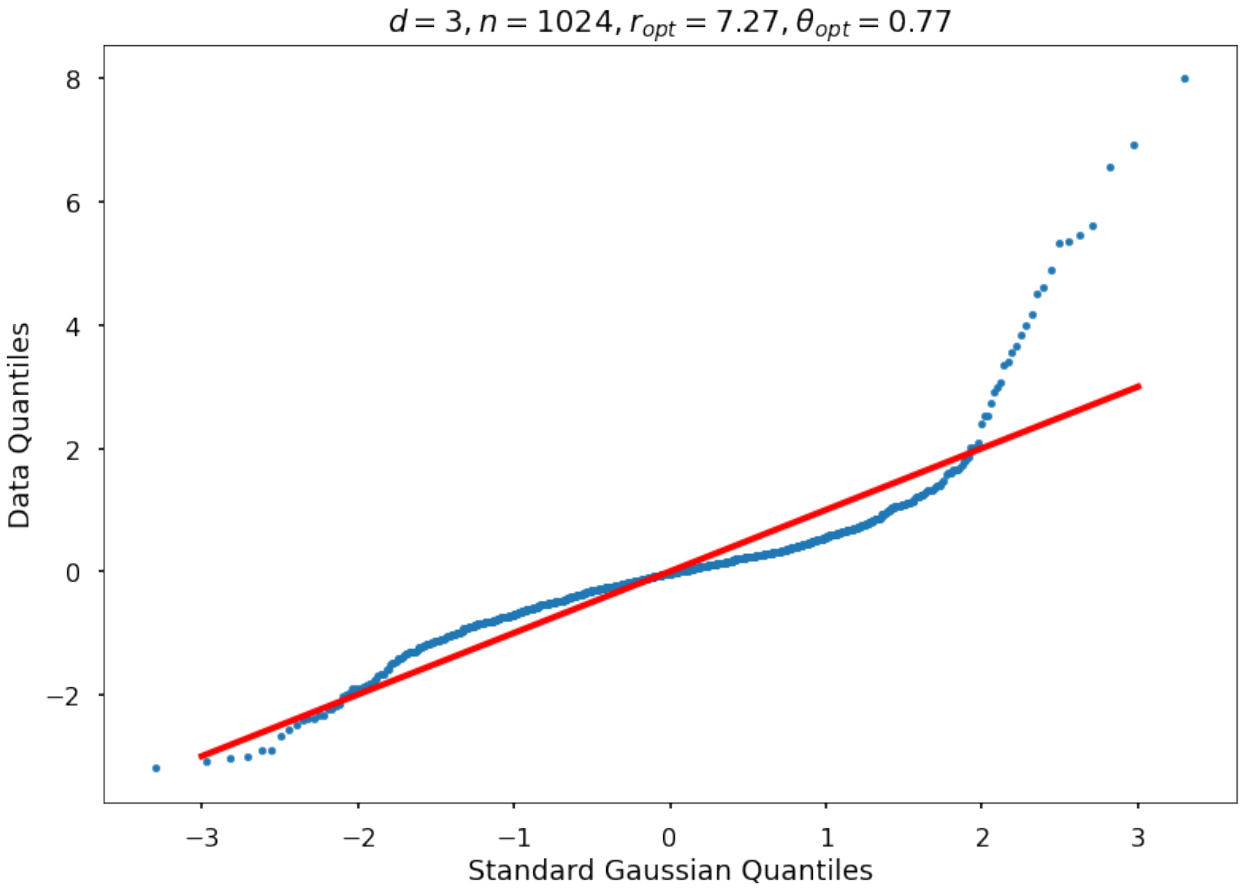
(continued from previous page)

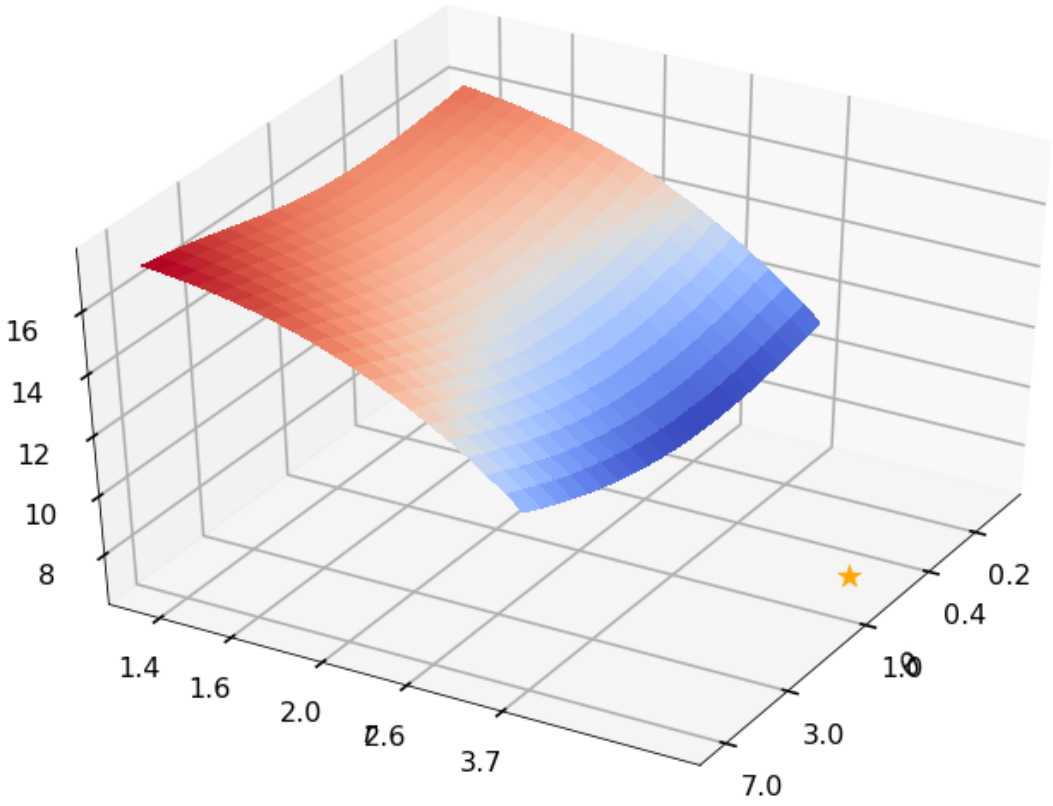
```
r = None, rOpt = 7.17539, theta = None, thetaOpt = 0.72916

10.594285418816956
7.138947266052696
r = None, rOpt = 7.16931, theta = None, thetaOpt = 0.70287

10.597310034462971
7.117069509729076
r = None, rOpt = 7.30321, theta = None, thetaOpt = 0.73752

10.592677212556865
7.085904146817816
r = None, rOpt = 7.26545, theta = None, thetaOpt = 0.75314

10.592216478157543
7.05396738656259
r = None, rOpt = 7.29427, theta = None, thetaOpt = 0.74802

10.59392746212006
7.109982966182546
r = None, rOpt = 7.20702, theta = None, thetaOpt = 0.69972

10.595997284727325
7.111744447354244
r = None, rOpt = 7.24907, theta = None, thetaOpt = 0.72795
```

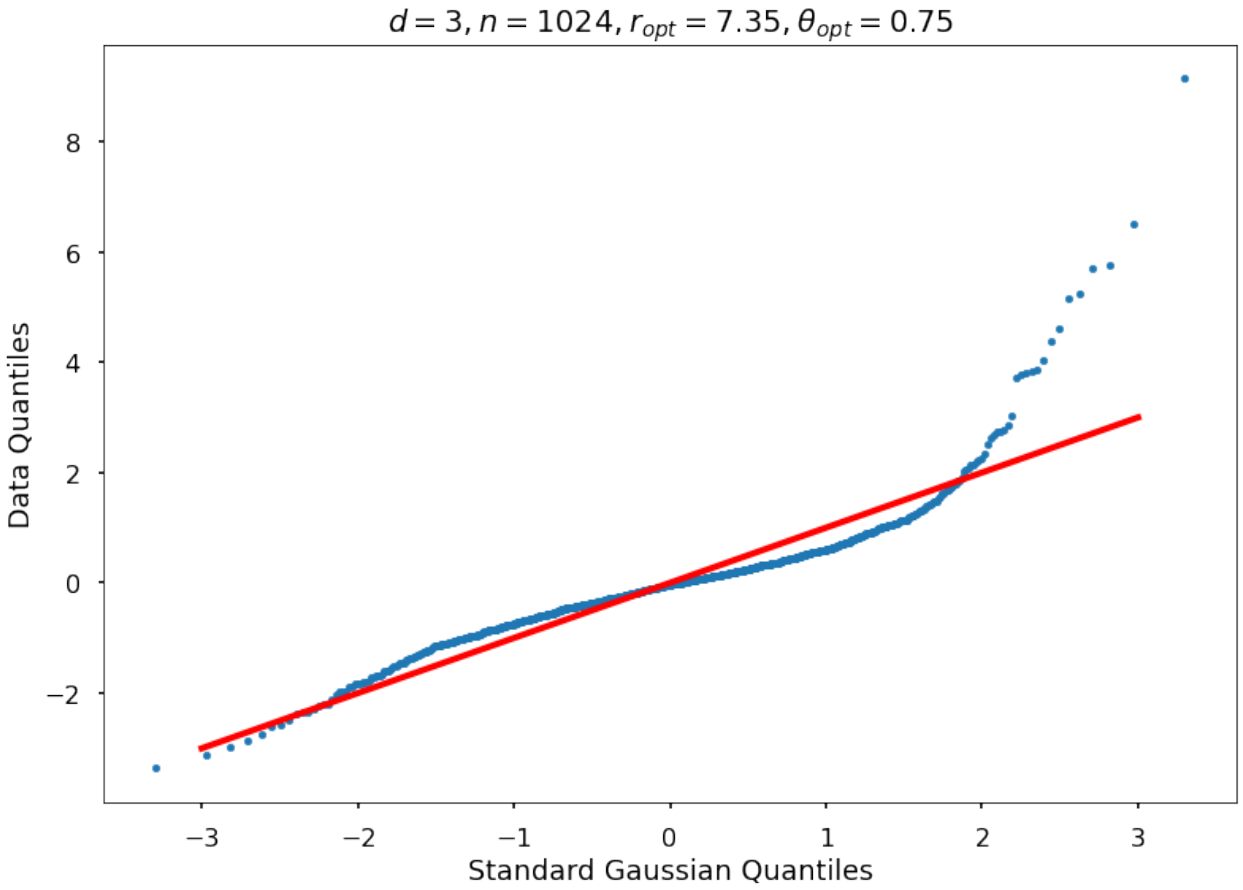


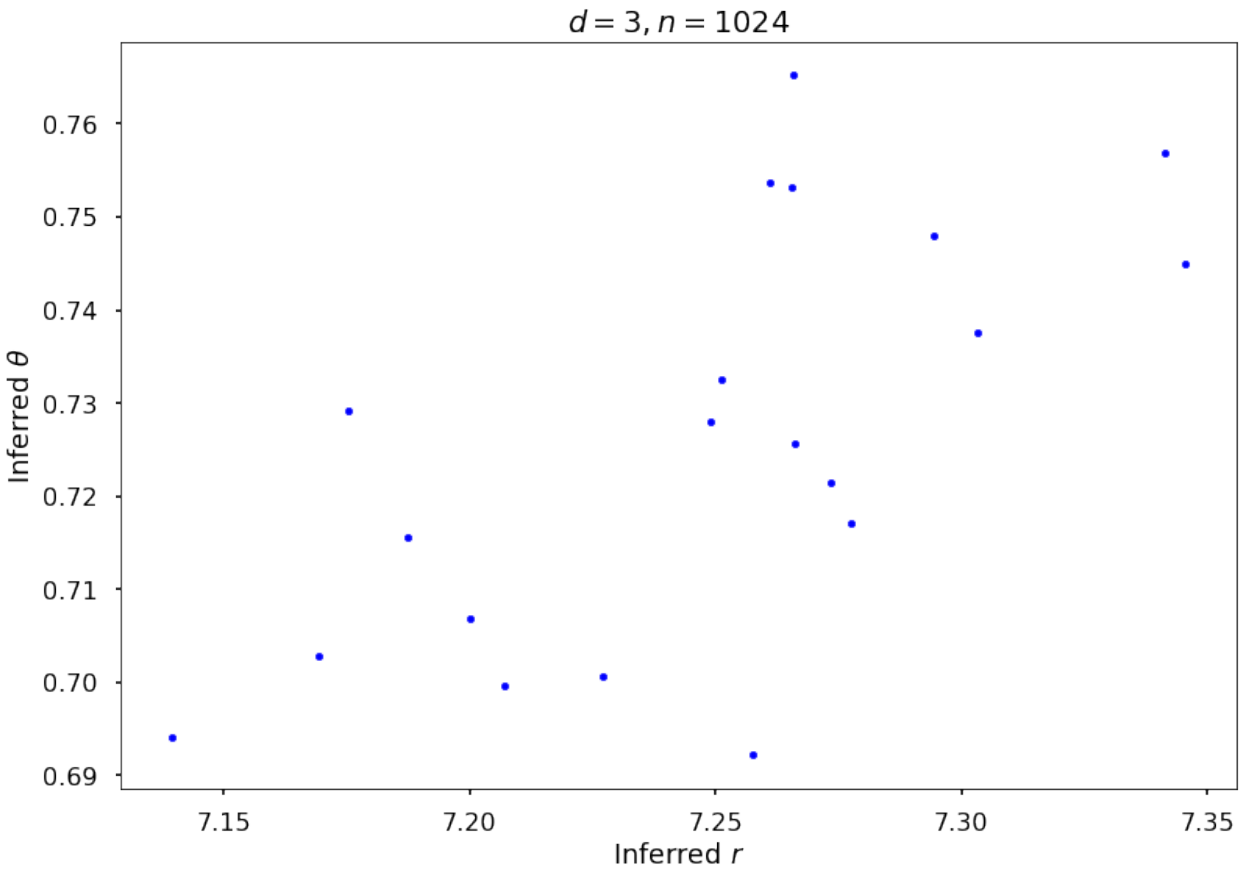

$d = 3, n = 1024, r_{opt} = 7.19, \theta_{opt} = 0.72$
Data Quantiles
Standard Gaussian Quantiles
10
8
6
4
2
0
−2
−4
−3
−2
−1
0
1
2
3


16
14
12
10
8
1.4
1.6
2.0
2.6
3.7
0.2
0.4
3.0
7.0


$d = 3, n = 1024, r_{opt} = 7.27, \theta_{opt} = 0.77$
Data Quantiles
Standard Gaussian Quantiles
8
6
4
2
0
−2
−3
−2
−1
0
1
2
3


16
14
12
10
8
1.4
1.6
2.0
2.6
3.7
0.2
0.4
3.0
7.0


$d = 3, n = 1024, r_{opt} = 7.35, \theta_{opt} = 0.75$
Data Quantiles
Standard Gaussian Quantiles
8
6
4
2
0
−2
−3
−2
−1
0
1
2
3



5.19 ML Sensitivity Indices

This notebook demonstrates QMCPy's support for vectorized sensitivity index computation. We preview this functionality by performing classification of Iris species using a decision tree. The computed sensitivity indices provide insight into input subset importance for a classic machine learning problem.

```
from numpy import *
from qmcpy import *
import pandas as pd
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier,plot_tree
from sklearn.model_selection import train_test_split
from skopt import gp_minimize
from matplotlib import pyplot
```

5.19.1 Load Data

We begin by reading in the Iris dataset and providing some basic summary statistics. Our goal will be to predict the Iris class (Setosa, Versicolour, or Virginica) based on Iris attributes (sepal length, sepal width, petal length, and petal width).

```
data = load_iris()
print(data['DESCR'])
```

```
.. _iris_dataset:

Iris plants dataset
--------------------

Data Set Characteristics:

    :Number of Instances: 150 (50 in each of three classes)
    :Number of Attributes: 4 numeric, predictive attributes and the class
    :Attribute Information:
        - sepal length in cm
        - sepal width in cm
        - petal length in cm
        - petal width in cm
        - class:
                - Iris-Setosa
                - Iris-Versicolour
                - Iris-Virginica

    :Summary Statistics:

    ============== ==== ==== ======= ===== ====================
                    Min  Max   Mean    SD   Class Correlation
    ============== ==== ==== ======= ===== ====================
    sepal length:   4.3  7.9   5.84   0.83    0.7826
    sepal width:    2.0  4.4   3.05   0.43   -0.4194
    petal length:   1.0  6.9   3.76   1.76    0.9490  (high!)
    petal width:    0.1  2.5   1.20   0.76    0.9565  (high!)
    ============== ==== ==== ======= ===== ====================

    :Missing Attribute Values: None
    :Class Distribution: 33.3% for each of 3 classes.
    :Creator: R.A. Fisher
    :Donor: Michael Marshall (MARSHALL%PLU@io.arc.nasa.gov)
    :Date: July, 1988

The famous Iris database, first used by Sir R.A. Fisher. The dataset is taken
from Fisher's paper. Note that it's the same as in R, but not as in the UCI
Machine Learning Repository, which has two wrong data points.

This is perhaps the best known database to be found in the
pattern recognition literature.  Fisher's paper is a classic in the field and
is referenced frequently to this day.  (See Duda & Hart, for example.)  The
data set contains 3 classes of 50 instances each, where each class refers to a
type of iris plant.  One class is linearly separable from the other 2; the
```

```
latter are NOT linearly separable from each other.

.. topic:: References

   - Fisher, R.A. "The use of multiple measurements in taxonomic problems"
     Annual Eugenics, 7, Part II, 179-188 (1936); also in "Contributions to
     Mathematical Statistics" (John Wiley, NY, 1950).
   - Duda, R.O., & Hart, P.E. (1973) Pattern Classification and Scene Analysis.
     (Q327.D83) John Wiley & Sons.  ISBN 0-471-22361-1.  See page 218.
   - Dasarathy, B.V. (1980) "Nosing Around the Neighborhood: A New System
     Structure and Classification Rule for Recognition in Partially Exposed
     Environments".  IEEE Transactions on Pattern Analysis and Machine
     Intelligence, Vol. PAMI-2, No. 1, 67-71.
   - Gates, G.W. (1972) "The Reduced Nearest Neighbor Rule".  IEEE Transactions
     on Information Theory, May 1972, 431-433.
   - See also: 1988 MLC Proceedings, 54-64.  Cheeseman et al"s AUTOCLASS II
     conceptual clustering system finds 3 classes in the data.
   - Many, many more ...
```

```
x = data['data']
y = data['target']
feature_names = data['feature_names']
df = pd.DataFrame(hstack((x,y[:,None])),columns=feature_names+['iris type'])
print('df shape:',df.shape)
target_names = data['target_names']
iris_type_map = {i:target_names[i] for i in range(len(target_names))}
print('iris species map:',iris_type_map)
df.head()
```

```
df shape: (150, 5)
iris species map: {0: 'setosa', 1: 'versicolor', 2: 'virginica'}
```

```
xt,xv,yt,yv = train_test_split(x,y,test_size=1/3,random_state=7)
print('training data   (xt) shape: %s'%str(xt.shape))
print('training labels (yt) shape: %s'%str(yt.shape))
print('testing data    (xv) shape: %s'%str(xv.shape))
print('testing labels  (yv) shape: %s'%str(yv.shape))
```

```
training data   (xt) shape: (100, 4)
training labels (yt) shape: (100,)
testing data    (xv) shape: (50, 4)
testing labels  (yv) shape: (50,)
```

5.19.2 Importance of Decision Tree Hyperparameters

We would like to predict Iris species using a Decision Tree (DT) classifier. When initializing a DT, we arrive at the question of how to set hyperparameters such as tree depth or the minimum weight fraction for each leaf. These hyperparameters can greatly effect classification accuracy, so it is worthwhile to consider their importance to determining classification performance.

Note that while this notebook uses decision trees and the Iris dataset, the methodology is directly applicable to other datasets and models.

We begin this exploration by setting up a hyperparameter domain in which to uniformly sample DT hyperparameter configurations. A helper function and its tie into QMCPy are also created.

```
hp_domain = [
    {'name':'max_depth', 'bounds':[1,8]},
    {'name':'min_weight_fraction_leaf', 'bounds':[0,.5]}]
hpnames = [param['name'] for param in hp_domain]
hp_lb = array([param['bounds'][0] for param in hp_domain])
hp_ub = array([param['bounds'][1] for param in hp_domain])
d = len(hp_domain)
def get_dt_accuracy(hparams):
    accuracies = zeros(len(hparams))
    for i,hparam in enumerate(hparams):
        kwargs = {hp_domain[j]['name']:hparam[j] for j in range(d)}
        kwargs['max_depth'] = int(kwargs['max_depth'])
        dt = DecisionTreeClassifier(random_state=7,**kwargs).fit(xt,yt)
        yhat = dt.predict(xv)
        accuracies[i] = mean(yhat==yv)
    return accuracies
cf = CustomFun(
    true_measure = Uniform(DigitalNetB2(d,seed=7),lower_bound=hp_lb,upper_bound=hp_ub),
    g = get_dt_accuracy,
    parallel=False)
```

Average Accuracy

Our first goal will be to find the average DT accuracy across the hyperparameter domain. To do so, we perform quasi-Monte Carlo numerical integration to approximate the mean testing accuracy.

```
avg_accuracy,data_avg_accuracy = CubQMCNetG(cf,abs_tol=1e-4).integrate()
data_avg_accuracy
```

```
LDTransformData (AccumulateData Object)
    solution        0.787
    comb_bound_low  0.787
    comb_bound_high 0.787
    comb_flags      1
    n_total         2^(14)
    n               2^(14)
    time_integrate  8.745
CubQMCNetG (StoppingCriterion Object)
    abs_tol         1.00e-04
    rel_tol         0
```

(continues on next page)

(continued from previous page)

```
    n_init          2^(10)
    n_max           2^(35)
CustomFun (Integrand Object)
Uniform (TrueMeasure Object)
    lower_bound     [1 0]
    upper_bound     [8.  0.5]
DigitalNetB2 (DiscreteDistribution Object)
    d               2^(1)
    dvec            [0 1]
    randomize       LMS_DS
    graycode        0
    entropy         7
    spawn_key       ()
```

Here we find the average accuracy to be 78.7% using 2^{14} samples.

Sensitivity Indices

Next, we wish to quantify how important individual hyperparamters are to determining testing accuracy. To do this, we compute the sensitivity indices of our hyperparameters. In QMCPy we use the `SensitivityIndices` class to compute these sensitivity indices.

```
si = SensitivityIndices(cf)
solution_importances,data_importances = CubQMCNetG(si,abs_tol=2.5e-2).integrate()
data_importances
```

```
LDTransformData (AccumulateData Object)
    solution        [[0.164 0.747]
                    [0.257 0.837]]
    comb_bound_low  [[0.148 0.726]
                    [0.238 0.819]]
    comb_bound_high [[0.179 0.768]
                    [0.277 0.856]]
    comb_flags      [[ True  True]
                    [ True  True]]
    n_total         2^(13)
    n               [[[4096. 8192.]
                      [4096. 8192.]
                      [4096. 8192.]]

                    [[2048. 8192.]
                      [2048. 8192.]
                      [2048. 8192.]]]
    time_integrate  14.999
CubQMCNetG (StoppingCriterion Object)
    abs_tol         0.025
    rel_tol         0
    n_init          2^(10)
    n_max           2^(35)
SensitivityIndices (Integrand Object)
    indices         [[0]
```

(continues on next page)

(continued from previous page)

```
                    [1]]
    n_multiplier    2^(1)
Uniform (TrueMeasure Object)
    lower_bound     [1 0]
    upper_bound     [8.  0.5]
DigitalNetB2 (DiscreteDistribution Object)
    d               4
    dvec            [0 1 2 3]
    randomize       LMS_DS
    graycode        0
    entropy         7
    spawn_key       (0,)
```

```
print('closed sensitivity indices: %s'%str(solution_importances[0].squeeze()))
print('total sensitivity indices: %s'%str(solution_importances[1].squeeze()))
```

```
closed sensitivity indices: [0.16375873 0.74701157]
total sensitivity indices: [0.25724624 0.83732705]
```

Looking closer at the output, we see that the second hyperparameter (`min_weight_fraction_leaf`) is more important than the first one (`max_depth`). The closed sensitivity indices measure how much that hyperparameter contributes to testing accuracy variance. The total sensitivity indices measure how much that hyperparameter, or any subset of hyperparameters containing that one contributes to testing accuracy variance. For example, the first closed sensitivity index approximates the variability attributable to {`max_depth`} while the first total sensitivity index approximates the variability attributable to both {`max_depth`} and {`max_depth`,`min_weight_fraction_leaf`}.

Marginals

We may also use QMCPy's support for vectorized quasi-Monte Carlo to compute marginal distributions. This is relatively straightforward to do for the Uniform true measure used here, but caution should be taken when adapting these techniques to distributions without independent marginals.

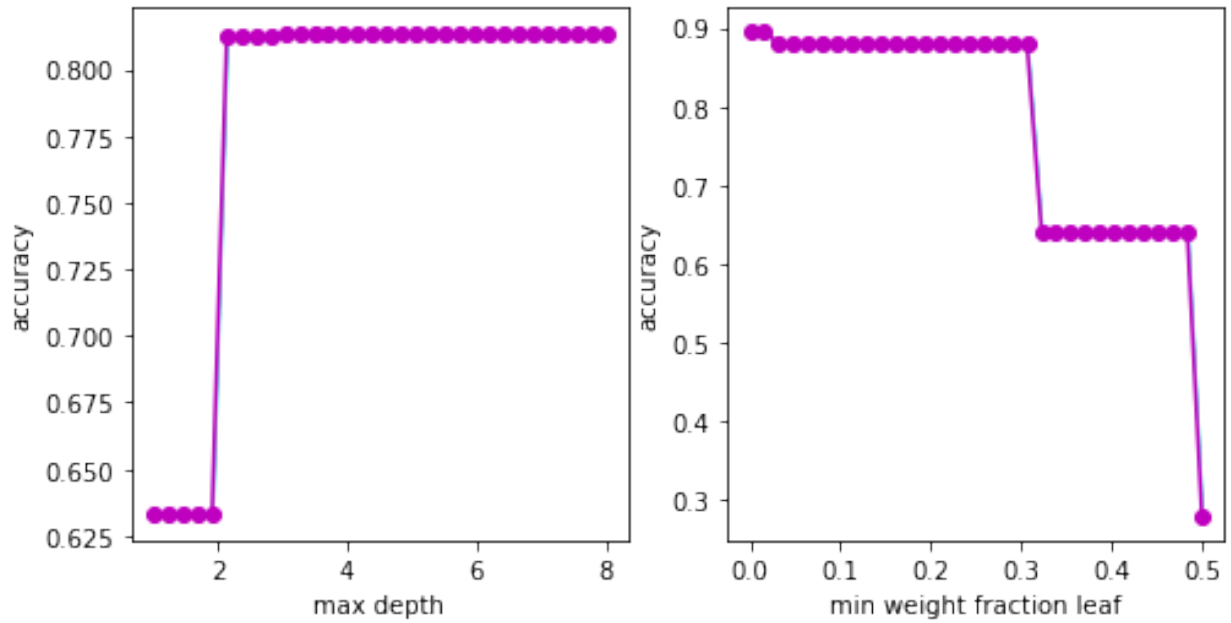
```
def marginal(x,compute_flags,xpts,bools,not_bools):
    n,_ = x.shape
    x2 = zeros((n,d),dtype=float)
    x2[:,bools] = x
    y = zeros((n,len(xpts)),dtype=float)
    for k,xpt in enumerate(xpts):
        if not compute_flags[k]: continue
        x2[:,not_bools] = xpt
        y[:,k] = get_dt_accuracy(x2)
    return y
```

```
fig,ax = pyplot.subplots(nrows=1,ncols=2,figsize=(8,4))
nticks = 32
xpts01 = linspace(0,1,nticks)
for i in range(2):
    xpts = xpts01*(hp_ub[i]-hp_lb[i])+hp_lb[i]
    bools = array([True if j not in [i] else False for j in range(d)])
    def marginal_i(x,compute_flags): return marginal(x,compute_flags,xpts,bools,~bools)
    cf = CustomFun(
```

(continues on next page)

(continued from previous page)

```
        true_measure = Uniform(DigitalNetB2(1,seed=7),lower_bound=hp_lb[bools],upper_
↪bound=hp_ub[bools]),
        g = marginal_i,
        dimension_indv = len(xpts),
        parallel=False)
    sol,data = CubQMCNetG(cf,abs_tol=5e-2).integrate()
    ax[i].plot(xpts,sol,'-o',color='m')
    ax[i].fill_between(xpts,data.comb_bound_high,data.comb_bound_low,color='c',alpha=.5)
    ax[i].set_xlabel(hpnames[i].replace('_',' '))
    ax[i].set_ylabel('accuracy')
```



5.19.3 Bayesian Optimization of Hyperparameters

Having explored DT hyperparameter importance, we are now ready to construct our optimal DT. We already have quite a bit of data relating hyperparameter settings to testing accuracy, so we may simply select the best configuration and call this an optimal DT. However, if we are looking to squeeze out even more performance, we may choose to perform Bayesian Optimization which incorporates our past metadata. Sample code is provided below despite not finding an improved configuration for this problem.

```
x0 = data_avg_accuracy.xfull*(hp_ub-hp_lb)+hp_lb
y0 = -data_avg_accuracy.yfull.squeeze()
print('best result before BO is %d%% accuracy'%(-100*y0.min()))
result = gp_minimize(
    func = lambda hparams: get_dt_accuracy(atleast_2d(hparams)).squeeze().item(),
    dimensions = [(l,u) for l,u in zip(hp_lb,hp_ub)],
    n_calls = 32,
    n_initial_points = 0,
    x0 = x0[:128].tolist(),
    y0 = y0[:128].tolist(),
    random_state = 7)
```

(continues on next page)

(continued from previous page)

```
xbo_best = result.x
ybo_best = -result.fun
print('best result from BO is %d%% accuracy'%(100*ybo_best))
xbo = array(result.x_iters)
ybo = -array(result.func_vals)
```

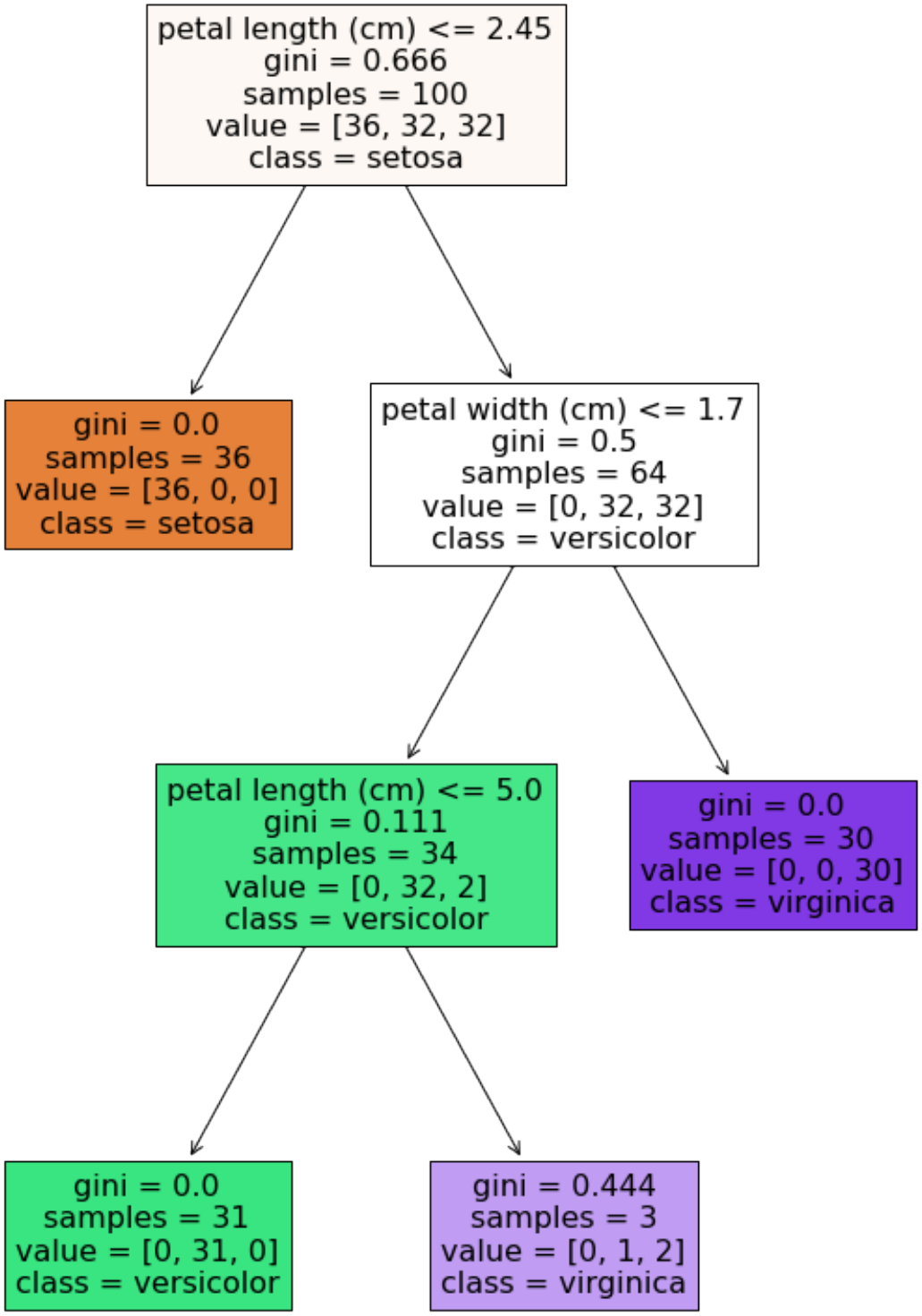
```
best result before BO is 94% accuracy
best result from BO is 94% accuracy
```

5.19.4 Best Decision Tree Analysis

Below we print the configuration that rested in the best DT. We also print the optimal accuracy achieved (at this configuration) and visualize the branches of this tree.

```
best_kwargs = {name:val for name,val in zip(hpnames,xbo_best)}
best_kwargs['max_depth'] = int(best_kwargs['max_depth'])
print(best_kwargs)
dt = DecisionTreeClassifier(random_state=7,**best_kwargs).fit(xt,yt)
yhat = dt.predict(xv)
accuracy = mean(yhat==yv)
print('best decision tree accuracy: %.1f%%'%(100*accuracy))
fig = pyplot.figure(figsize=(10,15))
plot_tree(dt,feature_names=feature_names,class_names=target_names,filled=True);
```

```
{'max_depth': 6, 'min_weight_fraction_leaf': 0.01750528148841113}
best decision tree accuracy: 94.0%
```


petal length (cm) <= 2.45
gini = 0.666
samples = 100
value = [36, 32, 32]
class = setosa
gini = 0.0
samples = 36
value = [36, 0, 0]
class = setosa
petal width (cm) <= 1.7
gini = 0.5
samples = 64
value = [0, 32, 32]
class = versicolor
petal length (cm) <= 5.0
gini = 0.111
samples = 34
value = [0, 32, 2]
class = versicolor
gini = 0.0
samples = 30
value = [0, 0, 30]
class = virginica
gini = 0.0
samples = 31
value = [0, 31, 0]
class = versicolor
gini = 0.444
samples = 3
value = [0, 1, 2]
class = virginica

Feature Importance

With the optimal DT in hand, we may now question how important the Irises features are in determining the class/species. To answer this question, we again perform sensitivity analysis, but this time we select a uniform measure over the domain of Iris features. Our output which we wish to quantify the variance of is now a length 3 vector of class probabilities. How variable is each species classification as a function of each Iris feature?

```
xfeatures = df.to_numpy()
xfeatures_low = xfeatures[:,:-1].min(0)
xfeatures_high = xfeatures[:,:-1].max(0)
d_features = len(xfeatures_low)
def dt_pp(t,compute_flags): return dt.predict_proba(t)
cf = CustomFun(
    true_measure = Uniform(DigitalNetB2(d_features,seed=7),
        lower_bound = xfeatures_low,
        upper_bound = xfeatures_high),
    g = dt_pp,
    dimension_indv = 3,
    parallel = False)
indices = [[0],[1],[2],[3],[0,1],[0,2],[0,3],[1,2],[1,3],[2,3],[1,2,3],[0,2,3],[0,1,2]]
si_cf = SobolIndices(cf,indices)
solution,data = CubQMCNetG(si_cf,abs_tol=1e-3,n_init=2**10).integrate()
data
```

```
LDTransformData (AccumulateData Object)
    solution        [[[0.    0.    0.   ]
                      [0.    0.    0.   ]
                      [0.999 0.43  0.456]
                      ...
                      [0.999 0.999 1.   ]
                      [0.999 0.999 1.   ]
                      [0.999 0.43  0.456]]

                     [[0.    0.    0.   ]
                      [0.    0.    0.   ]
                      [1.    0.646 0.662]
                      ...
                      [1.    0.999 0.999]
                      [1.    0.999 0.999]
                      [1.    0.646 0.662]]]
    comb_bound_low  [[[0.    0.    0.   ]
                      [0.    0.    0.   ]
                      [0.999 0.43  0.455]
                      ...
                      [0.999 0.998 0.999]
                      [0.999 0.998 0.999]
                      [0.999 0.43  0.455]]

                     [[0.    0.    0.   ]
                      [0.    0.    0.   ]
                      [0.999 0.645 0.661]
                      ...
                      [0.999 0.998 0.999]
```

(continues on next page)

(continued from previous page)

```
                    [0.999 0.998 0.999]
                    [0.999 0.645 0.661]]]
    comb_bound_high [[[0.    0.    0.   ]
                    [0.    0.    0.   ]
                    [1.    0.431 0.457]
                    ...
                    [1.    1.    1.   ]
                    [1.    1.    1.   ]
                    [1.    0.431 0.457]]

                   [[0.    0.    0.   ]
                    [0.    0.    0.   ]
                    [1.    0.647 0.662]
                    ...
                    [1.    1.    1.   ]
                    [1.    1.    1.   ]
                    [1.    0.647 0.662]]]
    comb_flags      [[[ True  True  True]
                    [ True  True  True]
                    [ True  True  True]
                    ...
                    [ True  True  True]
                    [ True  True  True]
                    [ True  True  True]]

                   [[ True  True  True]
                    [ True  True  True]
                    [ True  True  True]
                    ...
                    [ True  True  True]
                    [ True  True  True]
                    [ True  True  True]]]
    n_total         2^(17)
    n               [[[[  1024.   1024.   1024.]
                      [  1024.   1024.   1024.]
                      [  8192. 131072.  65536.]
                      ...
                      [  8192.  65536.  32768.]
                      [  8192.  65536.  32768.]
                      [  8192. 131072.  65536.]]

                     [[  1024.   1024.   1024.]
                      [  1024.   1024.   1024.]
                      [  8192. 131072.  65536.]
                      ...
                      [  8192.  65536.  32768.]
                      [  8192.  65536.  32768.]
                      [  8192. 131072.  65536.]]

                     [[  1024.   1024.   1024.]
                      [  1024.   1024.   1024.]
                      [  8192. 131072.  65536.]
```

(continues on next page)

(continued from previous page)

```
                      ...
                      [  8192.  65536.  32768.]
                      [  8192.  65536.  32768.]
                      [  8192. 131072.  65536.]]]


                    [[[  1024.   1024.   1024.]
                      [  1024.   1024.   1024.]
                      [  8192.  65536.  65536.]
                      ...
                      [  8192.  32768.  16384.]
                      [  8192.  32768.  16384.]
                      [  8192.  65536.  65536.]]

                     [[  1024.   1024.   1024.]
                      [  1024.   1024.   1024.]
                      [  8192.  65536.  65536.]
                      ...
                      [  8192.  32768.  16384.]
                      [  8192.  32768.  16384.]
                      [  8192.  65536.  65536.]]

                     [[  1024.   1024.   1024.]
                      [  1024.   1024.   1024.]
                      [  8192.  65536.  65536.]
                      ...
                      [  8192.  32768.  16384.]
                      [  8192.  32768.  16384.]
                      [  8192.  65536.  65536.]]]]
    time_integrate  3.418
CubQMCNetG (StoppingCriterion Object)
    abs_tol         0.001
    rel_tol         0
    n_init          2^(10)
    n_max           2^(35)
SobolIndices (Integrand Object)
    indices         [[0], [1], [2], [3], [0, 1], [0, 2], [0, 3], [1, 2], [1, 3], [2, 3],␣
↪[1, 2, 3], [0, 2, 3], [0, 1, 2]]
    n_multiplier    13
Uniform (TrueMeasure Object)
    lower_bound     [4.3 2.  1.  0.1]
    upper_bound     [7.9 4.4 6.9 2.5]
DigitalNetB2 (DiscreteDistribution Object)
    d               8
    dvec            [0 1 2 3 4 5 6 7]
    randomize       LMS_DS
    graycode        0
    entropy         7
    spawn_key       (0,)
```

While the solution looks unwieldy, it has quite a natural interpretation. The first axis determines whether we are looking at a closed (index 0) or total (index 1) sensitivity index as before. The second axis indexes the subset of features we are testing. The third and final axis is length 3 for the 3 class probabilities we are interested in. For example, `solution[0,`

2,2] looks at the closed sensitivity index of our index 2 feature (petal length) for our index 2 probability (virginica) AKA how important is petal length alone to determining if an Iris is virginica.

The results indicate that setosa Irises can be completely determined based on petal length while the versicolor and virginica Irises can be completely determined by looking at both petal length and petal width. Interestingly sepal length and sepal width do not contribute significantly to determining species.

These insights are not surprising or especially insightful for a decision tree where the tree structure indicates importance and the scores may even be computed directly. However, for more complicated models and datasets, this analysis pipeline may provide advanced insight into both hyperparameter tuning and feature importance.

```
print('solution shape:',solution.shape,'\n')
si_closed = solution[0]
si_total = solution[1]
print('SI Closed')
print(si_closed,'\n')
print('SI Total')
print(si_total)
```

```
solution shape: (2, 13, 3)

SI Closed
[[0.         0.         0.        ]
 [0.         0.         0.        ]
 [0.99938581 0.43048672 0.45606325]
 [0.         0.35460247 0.33840028]
 [0.         0.         0.        ]
 [0.99938581 0.43048672 0.45606325]
 [0.         0.35460247 0.33840028]
 [0.99938581 0.43048672 0.45606325]
 [0.         0.35460247 0.33840028]
 [0.99938581 0.99922725 0.99970515]
 [0.99938581 0.99922725 0.99970515]
 [0.99938581 0.99922725 0.99970515]
 [0.99938581 0.43048672 0.45606325]]

SI Total
[[0.         0.         0.        ]
 [0.         0.         0.        ]
 [0.9996645  0.64566888 0.66160745]
 [0.         0.56908397 0.54359418]
 [0.         0.         0.        ]
 [0.9996645  0.64566888 0.66160745]
 [0.         0.56908397 0.54359418]
 [0.9996645  0.64566888 0.66160745]
 [0.         0.56908397 0.54359418]
 [0.9996645  0.99903766 0.99927206]
 [0.9996645  0.99903766 0.99927206]
 [0.9996645  0.99903766 0.99927206]
 [0.9996645  0.64566888 0.66160745]]
```

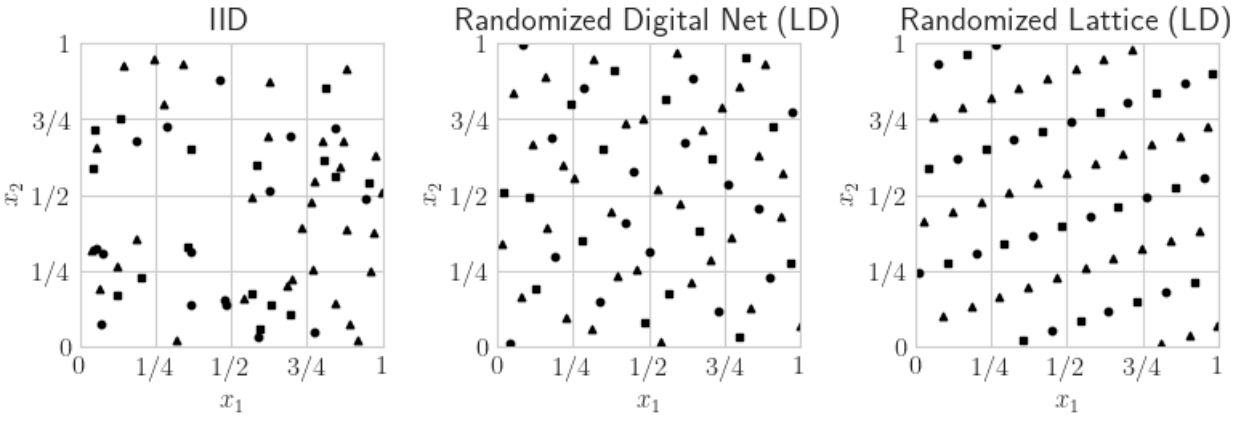
5.20 Vectorized QMC

```
import qmcpy as qp
import numpy as np
```

```
from matplotlib import pyplot
pyplot.style.use('../qmcpy/qmcpy.mplstyle')
%matplotlib inline
root = '../_ags/vec_qmc_out/'
```

5.20.1 LD Sequence

```
n = 2**6
s = 10
fig,ax = pyplot.subplots(figsize=(8,4),nrows=1,ncols=3)
for i,(dd,name) in enumerate(zip(
    [qp.IIDStdUniform(2,seed=7),qp.DigitalNetB2(2,seed=7),qp.Lattice(2,seed=7)],
    ['IID','Randomized Digital Net (LD)','Randomized Lattice (LD)'])):
    pts = dd.gen_samples(n)
    ax[i].scatter(pts[0:n//4,0],pts[0:n//4,1],color='k',marker='s',s=s)
    ax[i].scatter(pts[n//4:n//2,0],pts[n//4:n//2,1],color='k',marker='o',s=s)
    ax[i].scatter(pts[n//2:n,0],pts[n//2:n,1],color='k',marker='^',s=s)
    ax[i].set_aspect(1)
    ax[i].set_xlabel(r'$x_{1}$')
    ax[i].set_ylabel(r'$x_{2}$')
    ax[i].set_xlim([0,1])
    ax[i].set_ylim([0,1])
    ax[i].set_xticks([0,.25,.5,.75,1])
    ax[i].set_xticklabels([r'$0$',r'$1/4$',r'$1/2$',r'$3/4$',r'$1$'])
    ax[i].set_yticks([0,.25,.5,.75,1])
    ax[i].set_yticklabels([r'$0$',r'$1/4$',r'$1/2$',r'$3/4$',r'$1$'])
    ax[i].set_title(name)
if root: fig.savefig(root+'ld_seqs.pdf',transparent=True)
```



5.20.2 Simple Example

```
def cantilever_beam_function(T,compute_flags): # T is (n x 3)
    Y = np.zeros((len(T),2),dtype=float) # (n x 2)
    l,w,t = 100,4,2
    T1,T2,T3 = T[:,0],T[:,1],T[:,2] # Python is indexed from 0
    if compute_flags[0]: # compute D. x^2 is "x**2" in Python
        Y[:,0] = 4*l**3/(T1*w*t)*np.sqrt(T2**2/t**4+T3**2/w**4)
    if compute_flags[1]: # compute S
        Y[:,1] = 600*(T2/(w*t**2)+T3/(w**2*t))
    return Y
true_measure = qp.Gaussian(
    sampler = qp.DigitalNetB2(dimension=3,seed=7),
    mean = [2.9e7,500,1000],
    covariance = np.diag([(1.45e6)**2,(100)**2,(100)**2]))
integrand = qp.CustomFun(true_measure,
    g = cantilever_beam_function,
    dimension_indv = 2)
qmc_stop_crit = qp.CubQMCNetG(integrand,
    abs_tol = 1e-3,
    rel_tol = 1e-6)
solution,data = qmc_stop_crit.integrate()
print(solution)
# [2.42575885e+00 3.74999973e+04]
```

```
[2.42575885e+00 3.74999973e+04]
```

5.20.3 BO QEI

See the QEI Demo in QMCPy or the BoTorch Acquisition documentation for details on Bayesian Optimization using q-Expected Improvement.

```
import scipy
from sklearn.gaussian_process import GaussianProcessRegressor,kernels

f = lambda x: np.cos(10*x)*np.exp(.2*x)+np.exp(-5*(x-.4)**2)
xplt = np.linspace(0,1,100)
yplt = f(xplt)
x = np.array([.1, .2, .4, .7, .9])
y = f(x)
ymax = y.max()

gp = GaussianProcessRegressor(kernel=kernels.RBF(length_scale=1.0,length_scale_
↪bounds=(1e-2, 1e2)),
    n_restarts_optimizer = 16).fit(x[:,None],y)
yhatplt,stdhatplt = gp.predict(xplt[:,None],return_std=True)

tpax = 32
x0mesh,x1mesh = np.meshgrid(np.linspace(0,1,tpax),np.linspace(0,1,tpax))
post_mus = np.zeros((tpax,tpax,2),dtype=float)
post_sqrtcovs = np.zeros((tpax,tpax,2,2),dtype=float)
```

(continues on next page)

(continued from previous page)

```
for j0 in range(tpax):
    for j1 in range(tpax):
        candidate = np.array([[x0mesh[j0,j1]],[x1mesh[j0,j1]]])
        post_mus[j0,j1],post_cov = gp.predict(candidate,return_cov=True)
        evals,evecs = scipy.linalg.eig(post_cov)
        post_sqrtcovs[j0,j1] = np.sqrt(np.maximum(evals.real,0))*evecs

def qei_acq_vec(x,compute_flags):
    xgauss = scipy.stats.norm.ppf(x)
    n = len(x)
    qei_vals = np.zeros((n,tpax,tpax),dtype=float)
    for j0 in range(tpax):
        for j1 in range(tpax):
            if compute_flags[j0,j1]==False: continue
            sqrt_cov = post_sqrtcovs[j0,j1]
            mu_post = post_mus[j0,j1]
            for i in range(len(x)):
                yij = sqrt_cov@xgauss[i]+mu_post
                qei_vals[i,j0,j1] = max((yij-ymax).max(),0)
    return qei_vals

qei_acq_vec_qmcpy = qp.CustomFun(
    true_measure = qp.Uniform(qp.DigitalNetB2(2,seed=7)),
    g = qei_acq_vec,
    dimension_indv = (tpax,tpax),
    parallel=False)
qei_vals,qei_data = qp.CubQMCNetG(qei_acq_vec_qmcpy,abs_tol=.025,rel_tol=0).integrate()
↪# .0005
print(qei_data)

a = np.unravel_index(np.argmax(qei_vals,axis=None),qei_vals.shape)
xnext = np.array([x0mesh[a[0],a[1]],x1mesh[a[0],a[1]]])
fnext = f(xnext)
```

```
LDTransformData (AccumulateData Object)
    solution        [[0.06  0.079 0.072 ... 0.06  0.06  0.067]
                    [0.079 0.064 0.067 ... 0.064 0.065 0.071]
                    [0.072 0.067 0.032 ... 0.032 0.033 0.039]
                    ...
                    [0.06  0.064 0.032 ... 0.    0.001 0.007]
                    [0.06  0.065 0.033 ... 0.    0.001 0.007]
                    [0.066 0.07  0.039 ... 0.007 0.007 0.007]]
    comb_bound_low  [[0.059 0.078 0.071 ... 0.058 0.059 0.065]
                    [0.078 0.063 0.066 ... 0.063 0.064 0.07 ]
                    [0.071 0.066 0.032 ... 0.032 0.032 0.038]
                    ...
                    [0.059 0.063 0.032 ... 0.    0.001 0.006]
                    [0.059 0.064 0.032 ... 0.    0.    0.006]
                    [0.065 0.069 0.038 ... 0.006 0.006 0.006]]
    comb_bound_high [[0.061 0.08  0.073 ... 0.061 0.061 0.068]
                    [0.08  0.065 0.068 ... 0.065 0.066 0.072]
                    [0.073 0.068 0.033 ... 0.033 0.033 0.04 ]
```

(continues on next page)

(continued from previous page)

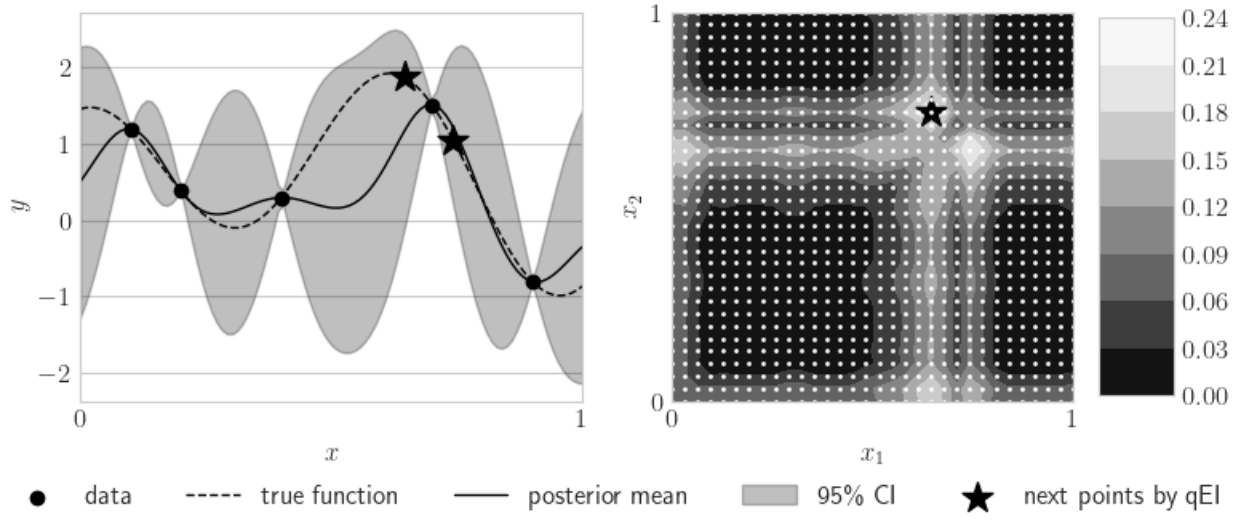
```
                    ...
                    [0.061 0.065 0.033 ... 0.    0.001 0.008]
                    [0.061 0.065 0.033 ... 0.001 0.001 0.008]
                    [0.067 0.072 0.04  ... 0.008 0.008 0.008]]
    comb_flags      [[ True  True  True ...  True  True  True]
                    [ True  True  True ...  True  True  True]
                    [ True  True  True ...  True  True  True]
                    ...
                    [ True  True  True ...  True  True  True]
                    [ True  True  True ...  True  True  True]
                    [ True  True  True ...  True  True  True]]
    n_total         2^(10)
    n               [[1024. 1024. 1024. ... 1024. 1024. 1024.]
                    [1024. 1024. 1024. ... 1024. 1024. 1024.]
                    [1024. 1024. 1024. ... 1024. 1024. 1024.]
                    ...
                    [1024. 1024. 1024. ... 1024. 1024. 1024.]
                    [1024. 1024. 1024. ... 1024. 1024. 1024.]
                    [1024. 1024. 1024. ... 1024. 1024. 1024.]]
    time_integrate  6.570
CubQMCNetG (StoppingCriterion Object)
    abs_tol         0.025
    rel_tol         0
    n_init          2^(10)
    n_max           2^(35)
CustomFun (Integrand Object)
Uniform (TrueMeasure Object)
    lower_bound     0
    upper_bound     1
DigitalNetB2 (DiscreteDistribution Object)
    d               2^(1)
    dvec            [0 1]
    randomize       LMS_DS
    graycode        0
    entropy         7
    spawn_key       ()
```

```
from matplotlib import cm
fig,ax = pyplot.subplots(nrows=1,ncols=2,figsize=(8,3.25))
ax[0].scatter(x,y,color='k',label='Query Points')
ax[0].plot(xplt,yplt,color='k',linestyle='--',label='True function',linewidth=1)
ax[0].plot(xplt,yhatplt,color='k',label='GP Mean',linewidth=1)
ax[0].fill_between(xplt,yhatplt-1.96*stdhatplt,yhatplt+1.96*stdhatplt,color='k',alpha=.
↪25,label='95% CI')
ax[0].scatter(xnext,fnext,color='k',marker='*',s=200,zorder=10)
ax[0].set_xlim([0,1])
ax[0].set_xticks([0,1])
ax[0].set_xlabel(r'$x$')
ax[0].set_ylabel(r'$y$')
fig.legend(labels=['data','true function','posterior mean',r'95\% CI','next points by qEI
↪'],loc='lower center',bbox_to_anchor=(.5,-.05),ncol=5)
contour = ax[1].contourf(x0mesh,x1mesh,qei_vals,cmap=cm.Greys_r)
```

(continues on next page)

(continued from previous page)

```
ax[1].scatter([xnext[0]],[xnext[1]],color='k',marker='*',s=200)
fig.colorbar(contour,ax=None,shrink=1,aspect=5)
ax[1].scatter(x0mesh.flatten(),x1mesh.flatten(),color='w',s=1)
ax[1].set_xlim([0,1])
ax[1].set_xticks([0,1])
ax[1].set_ylim([0,1])
ax[1].set_yticks([0,1])
ax[1].set_xlabel(r'$x_1$')
ax[1].set_ylabel(r'$x_2$')
if root: fig.savefig(root+'gp.pdf',transparent=True)
```



5.20.4 Bayesian Logistic Regression

```
import pandas as pd
from sklearn.model_selection import train_test_split
df = pd.read_csv('https://archive.ics.uci.edu/ml/machine-learning-databases/haberman/
↪haberman.data',header=None)
df.columns = ['Age','1900 Year','Axillary Nodes','Survival Status']
df.loc[df['Survival Status']==2,'Survival Status'] = 0
x,y = df[['Age','1900 Year','Axillary Nodes']],df['Survival Status']
xt,xv,yt,yv = train_test_split(x,y,test_size=.33,random_state=7)
```

```
print(df.head(),'\n')
print(df[['Age','1900 Year','Axillary Nodes']].describe(),'\n')
print(df['Survival Status'].astype(str).describe())
print('\ntrain samples: %d test samples: %d\n'%(len(xt),len(xv)))
print('train positives %d   train negatives: %d'%(np.sum(yt==1),np.sum(yt==0)))
print(' test positives %d    test negatives: %d'%(np.sum(yv==1),np.sum(yv==0)))
xt.head()
```

```
   Age  1900 Year  Axillary Nodes  Survival Status
0   30         64               1                1
```

(continues on next page)

(continued from previous page)

```
1   30         62               3                1
2   30         65               0                1
3   31         59               2                1
4   31         65               4                1

              Age   1900 Year  Axillary Nodes
count  306.000000  306.000000      306.000000
mean    52.457516   62.852941        4.026144
std     10.803452    3.249405        7.189654
min     30.000000   58.000000        0.000000
25%     44.000000   60.000000        0.000000
50%     52.000000   63.000000        1.000000
75%     60.750000   65.750000        4.000000
max     83.000000   69.000000       52.000000

count     306
unique      2
top         1
freq      225
Name: Survival Status, dtype: object

train samples: 205 test samples: 101

train positives 151   train negatives: 54
 test positives 74    test negatives: 27
```

```
blr = qp.BayesianLRCoeffs(
    sampler = qp.DigitalNetB2(4,seed=7),
    feature_array = xt, # np.ndarray of shape (n,d-1)
    response_vector = yt, # np.ndarray of shape (n,)
    prior_mean = 0, # normal prior mean = (0,0,...,0)
    prior_covariance = 5) # normal prior covariance = 5I
qmc_sc = qp.CubQMCNetG(blr,
    abs_tol = .05,
    rel_tol = .5,
    error_fun = lambda s,abs_tols,rel_tols:
        np.minimum(abs_tols,np.abs(s)*rel_tols))
blr_coefs,blr_data = qmc_sc.integrate()
print(blr_data)
# LDTransformData (AccumulateData Object)
#     solution        [-0.004  0.13  -0.157  0.008]
#     comb_bound_low  [-0.006  0.092 -0.205  0.007]
#     comb_bound_high [-0.003  0.172 -0.109  0.012]
#     comb_flags      [ True  True  True  True]
#     n_total         2^(18)
#     n               [[  1024.   1024. 262144.   2048.]
#                     [  1024.   1024. 262144.   2048.]]
#     time_integrate  2.229
```

```
LDTransformData (AccumulateData Object)
    solution        [-0.004  0.13  -0.157  0.008]
```

(continues on next page)

(continued from previous page)

```
    comb_bound_low  [-0.006  0.092 -0.205  0.007]
    comb_bound_high [-0.003  0.172 -0.109  0.012]
    comb_flags      [ True  True  True  True]
    n_total         2^(18)
    n               [[  1024.   1024. 262144.   2048.]
                    [  1024.   1024. 262144.   2048.]]
    time_integrate  2.318
CubQMCNetG (StoppingCriterion Object)
    abs_tol         0.050
    rel_tol         2^(-1)
    n_init          2^(10)
    n_max           2^(35)
BayesianLRCoeffs (Integrand Object)
Gaussian (TrueMeasure Object)
    mean            0
    covariance      5
    decomp_type     PCA
DigitalNetB2 (DiscreteDistribution Object)
    d               2^(2)
    dvec            [0 1 2 3]
    randomize       LMS_DS
    graycode        0
    entropy         7
    spawn_key       ()
```

```
from sklearn.linear_model import LogisticRegression
def metrics(y,yhat):
    y,yhat = np.array(y),np.array(yhat)
    tp = np.sum((y==1)*(yhat==1))
    tn = np.sum((y==0)*(yhat==0))
    fp = np.sum((y==0)*(yhat==1))
    fn = np.sum((y==1)*(yhat==0))
    accuracy = (tp+tn)/(len(y))
    precision = tp/(tp+fp)
    recall = tp/(tp+fn)
    return [accuracy,precision,recall]

results = pd.DataFrame({name:[] for name in ['method','Age','1900 Year','Axillary Nodes',
↪'Intercept','Accuracy','Precision','Recall']})
for i,l1_ratio in enumerate([0,.5,1]):
    lr = LogisticRegression(random_state=7,penalty="elasticnet",solver='saga',l1_
↪ratio=l1_ratio).fit(xt,yt)
    results.loc[i] = [r'Elastic-Net \lambda=%.1f'%l1_ratio]+lr.coef_.squeeze().
↪tolist()+[lr.intercept_.item()]+metrics(yv,lr.predict(xv))

blr_predict = lambda x: 1/(1+np.exp(-np.array(x)@blr_coefs[:-1]-blr_coefs[-1]))>=.5
blr_train_accuracy = np.mean(blr_predict(xt)==yt)
blr_test_accuracy = np.mean(blr_predict(xv)==yv)
results.loc[len(results)] = ['Bayesian']+blr_coefs.squeeze().tolist()+metrics(yv,blr_
↪predict(xv))

import warnings
```

(continues on next page)

(continued from previous page)

```
warnings.simplefilter('ignore',FutureWarning)
results.set_index('method',inplace=True)
print(results.head())
#root: results.to_latex(root+'lr_table.tex',formatters={'%s'%tt:lambda v:'%.1f'%(100*v) for␣
↪tt in ['accuracy','precision','recall']},float_format="%.2e")
```

```
                              Age  1900 Year  Axillary Nodes  Intercept  method
Elastic-Net lambda=0.0 -0.012279   0.034401       -0.115153   0.001990
Elastic-Net lambda=0.5 -0.012041   0.034170       -0.114770   0.002025
Elastic-Net lambda=1.0 -0.011803   0.033940       -0.114387   0.002061
Bayesian               -0.004138   0.129921        -0.156901   0.008034

                         Accuracy  Precision    Recall
method
Elastic-Net lambda=0.0  0.742574   0.766667  0.932432
Elastic-Net lambda=0.5  0.742574   0.766667  0.932432
Elastic-Net lambda=1.0  0.742574   0.766667  0.932432
Bayesian                0.742574   0.740000  1.000000
```

5.20.5 Sensitivity Indices

Ishigami Function

```
a,b = 7,0.1
dnb2 = qp.DigitalNetB2(3,seed=7)
ishigami = qp.Ishigami(dnb2,a,b)
idxs =[[0], [1], [2], [0,1], [0,2], [1,2]]
ishigami_si = qp.SensitivityIndices(ishigami,idxs)
qmc_algo = qp.CubQMCNetG(ishigami_si,abs_tol=.05)
solution,data = qmc_algo.integrate()
print(data)
si_closed = solution[0].squeeze()
si_total = solution[1].squeeze()
ci_comb_low_closed = data.comb_bound_low[0].squeeze()
ci_comb_high_closed = data.comb_bound_high[0].squeeze()
ci_comb_low_total = data.comb_bound_low[1].squeeze()
ci_comb_high_total = data.comb_bound_high[1].squeeze()
print("\nApprox took %.1f sec and n = 2^(%d)"%
    (data.time_integrate,np.log2(data.n_total)))
print('\t si_closed:',si_closed)
print('\t si_total:',si_total)
print('\t ci_comb_low_closed:',ci_comb_low_closed)
print('\t ci_comb_high_closed:',ci_comb_high_closed)
print('\t ci_comb_low_total:',ci_comb_low_total)
print('\t ci_comb_high_total:',ci_comb_high_total)

true_indices = qp.Ishigami._exact_sensitivity_indices(idxs,a,b)
si_closed_true = true_indices[0]
si_total_true = true_indices[1]
```

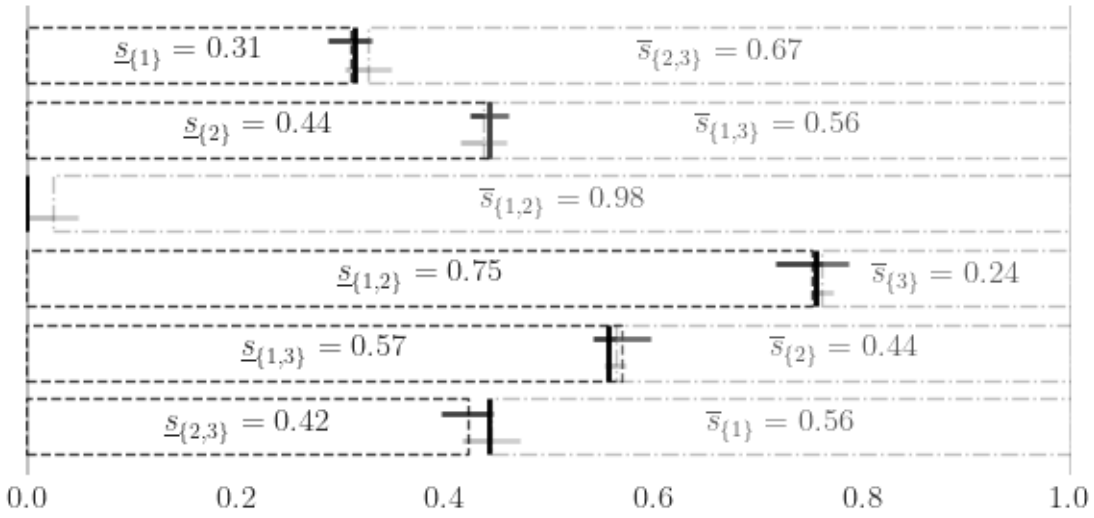
```
LDTransformData (AccumulateData Object)
    solution        [[0.31  0.442 0.    0.752 0.57  0.422]
                    [0.555 0.436 0.239 0.976 0.563 0.673]]
    comb_bound_low  [[0.289 0.424 0.    0.717 0.542 0.397]
                    [0.527 0.426 0.228 0.952 0.54  0.651]]
    comb_bound_high [[0.33  0.461 0.    0.787 0.598 0.447]
                    [0.583 0.446 0.249 1.    0.585 0.695]]
    comb_flags      [[ True  True  True  True  True  True]
                    [ True  True  True  True  True  True]]
    n_total         2^(10)
    n               [[[1024. 1024. 1024. 1024. 1024. 1024.]
                      [1024. 1024. 1024. 1024. 1024. 1024.]
                      [1024. 1024. 1024. 1024. 1024. 1024.]]

                    [[1024. 1024. 1024. 1024. 1024. 1024.]
                      [1024. 1024. 1024. 1024. 1024. 1024.]
                      [1024. 1024. 1024. 1024. 1024. 1024.]]]
    time_integrate  0.030
CubQMCNetG (StoppingCriterion Object)
    abs_tol         0.050
    rel_tol         0
    n_init          2^(10)
    n_max           2^(35)
SensitivityIndices (Integrand Object)
    indices         [list([0]) list([1]) list([2]) list([0, 1]) list([0, 2]) list([1,␣
↪2])]
    n_multiplier    6
Uniform (TrueMeasure Object)
    lower_bound     -3.142
    upper_bound     3.142
DigitalNetB2 (DiscreteDistribution Object)
    d               6
    dvec            [0 1 2 3 4 5]
    randomize       LMS_DS
    graycode        0
    entropy         7
    spawn_key       (0,)

Approx took 0.0 sec and n = 2^(10)
     si_closed: [0.30971745 0.4423662  0.         0.75200486 0.56984213 0.42224468]
     si_total: [0.55503279 0.43583295 0.23855184 0.97582279 0.56253104 0.67338673]
     ci_comb_low_closed: [0.28915713 0.42422349 0.         0.71701308 0.54177848 0.
↪3974432 ]
     ci_comb_high_closed: [0.33027778 0.46050891 0.         0.78699664 0.59790578 0.
↪44704617]
     ci_comb_low_total: [0.5274417  0.42560848 0.22838115 0.95164558 0.54006293 0.
↪65149999]
     ci_comb_high_total: [0.58262388 0.44605742 0.24872253 1.         0.58499915 0.
↪69527348]
```

```
/Users/alegresor/Desktop/QMCSoftware/qmcpy/util/abstraction_functions.py:36
    VisibleDeprecationWarning: Creating an ndarray from ragged nested sequences (which␣
↪is a list-or-tuple of lists-or-tuples-or ndarrays with different lengths or shapes) is␣
↪deprecated. If you meant to do this, you must specify 'dtype=object' when creating the␣
↪ndarray.
```

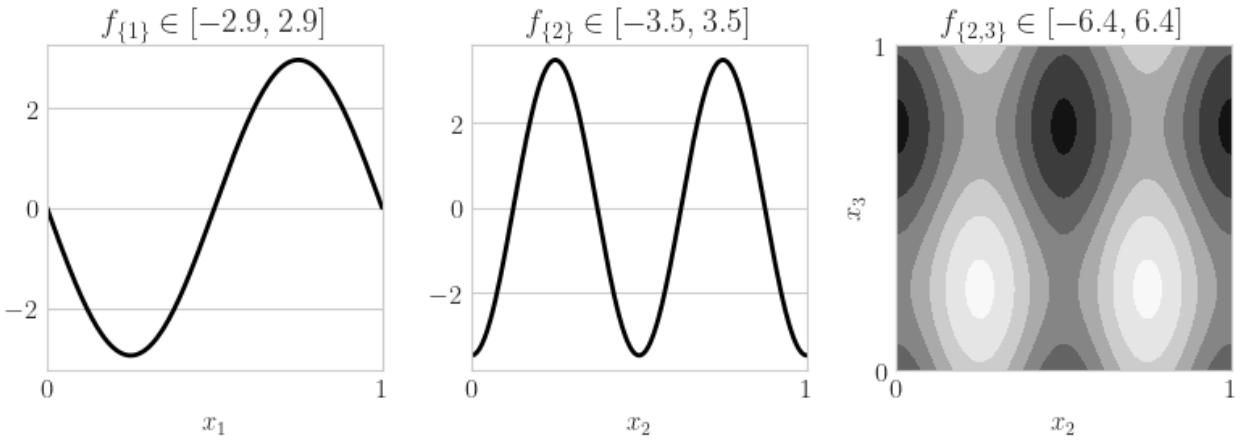
(continues on next page)

(continued from previous page)

```
fig,ax = pyplot.subplots(figsize=(6,3))
ax.grid(False)
for spine in ['top','left','right','bottom']: ax.spines[spine].set_visible(False)
width = .75
ax.errorbar(fmt='none',color='k',
    x = 1-si_total_true,
    y = np.arange(len(si_closed)),
    xerr = 0,
    yerr = width/2,
    alpha = 1)
bar_closed = ax.barh(np.arange(len(si_closed)),np.flip(si_closed),width,label='Closed SI
↪',color='w',edgecolor='k',alpha=.75,linestyle='--')
ax.errorbar(fmt='none',color='k',
    x = si_closed,
    y = np.flip(np.arange(len(si_closed)))+width/4,
    xerr = np.vstack((si_closed-ci_comb_low_closed,ci_comb_high_closed-si_closed)),
    yerr = 0,
    #elinewidth = 5,
    alpha = .75)
bar_total = ax.barh(np.arange(len(si_closed)),si_total,width,label='Total SI',color='w',
↪alpha=.25,edgecolor='k',left=1-si_total,zorder=10,linestyle='-.')
ax.errorbar(fmt='none',color='k',
    x = 1-si_total,
    y = np.arange(len(si_closed))-width/4,
    xerr = np.vstack((si_total-ci_comb_low_total,ci_comb_high_total-si_total)),
    yerr = 0,
    #elinewidth = 5,
    alpha = .25)
closed_labels = [r'$\underline{s}_{\{%s\}} = %.2f$'%(','.join([str(i+1) for i in idx]),
↪c) for idx,c in zip(idxs[::-1],np.flip(si_closed))]
closed_labels[3] = ''
total_labels = [r'$\overline{s}_{\{%s\}} = %.2f$'%(','.join([str(i+1) for i in idx]),t)␣
↪for idx,t in zip(idxs,si_total)]
ax.bar_label(bar_closed,label_type='center',labels=closed_labels)
ax.bar_label(bar_total,label_type='center',labels=total_labels)
ax.set_xlim([-.001,1.001])
ax.axvline(x=0,ymin=0,ymax=len(si_closed),color='k',alpha=.25)
ax.axvline(x=1,ymin=0,ymax=len(si_closed),color='k',alpha=.25)
ax.set_yticklabels([])
if root: fig.savefig(root+'ishigami.pdf',transparent=True)
```




```
fig,ax = pyplot.subplots(nrows=1,ncols=3,figsize=(8,3))
x_1d = np.linspace(0,1,num=128)
x_1d_mat = np.tile(x_1d,(3,1)).T
y_1d = qp.Ishigami._exact_fu_functions(x_1d_mat,indices=[[0],[1],[2]],a=a,b=b)
for i in range(2):
    ax[i].plot(x_1d,y_1d[:,i],color='k')
    ax[i].set_xlim([0,1])
    ax[i].set_xticks([0,1])
    ax[i].set_xlabel(r'$x_{%d}$'%(i+1))
    ax[i].set_title(r'$f_{\{%d\}} \in [%.1f,%.1f]$'%(i+1,y_1d[:,i].min(),y_1d[:,i].
↪max()))
x_mesh,y_mesh = np.meshgrid(x_1d,x_1d)
xquery = np.zeros((x_mesh.size,3))
for i,idx in enumerate([[1,2]]): # [[0,1],[0,2],[1,2]]
    xquery[:,idx[0]] = x_mesh.flatten()
    xquery[:,idx[1]] = y_mesh.flatten()
    zquery = qp.Ishigami._exact_fu_functions(xquery,indices=[idx],a=a,b=b)
    z_mesh = zquery.reshape(x_mesh.shape)
    ax[2+i].contourf(x_mesh,y_mesh,z_mesh,cmap=cm.Greys_r)
    ax[2+i].set_xlabel(r'$x_{%d}$'%(idx[0]+1))
    ax[2+i].set_ylabel(r'$x_{%d}$'%(idx[1]+1))
    ax[2+i].set_title(r'$f_{\{%d,%d\}} \in [%.1f,%.1f]$'%(tuple([i+1 for i in idx])+(z_
↪mesh.min(),z_mesh.max())))
    ax[2+i].set_xlim([0,1])
    ax[2+i].set_ylim([0,1])
    ax[2+i].set_xticks([0,1])
    ax[2+i].set_yticks([0,1])
if root: fig.savefig(root+'ishigami_fu.pdf')
```



Neural Network

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPClassifier
data = load_iris()
feature_names = data["feature_names"]
feature_names = [fn.replace('sepal ','S')\
    .replace('length ','L')\
    .replace('petal ','P')\
    .replace('width ','W')\
    .replace('(cm)','') for fn in feature_names]
target_names = data["target_names"]
xt,xv,yt,yv = train_test_split(data["data"],data["target"],
    test_size = 1/3,
    random_state = 7)
```

```
mlpc = MLPClassifier(random_state=7,max_iter=1024).fit(xt,yt)
yhat = mlpc.predict(xv)
print("accuracy: %.1f%%"%(100*(yv==yhat).mean()))
# accuracy: 98.0%
sampler = qp.DigitalNetB2(4,seed=7)
true_measure =  qp.Uniform(sampler,
    lower_bound = xt.min(0),
    upper_bound = xt.max(0))
fun = qp.CustomFun(
    true_measure = true_measure,
    g = lambda x,compute_flags: mlpc.predict_proba(x),
    dimension_indv = 3)
si_fun = qp.SensitivityIndices(fun,indices="all")
qmc_algo = qp.CubQMCNetG(si_fun,abs_tol=.005)
nn_sis,nn_sis_data = qmc_algo.integrate()
```

```
accuracy: 98.0%
```

```
#print(nn_sis_data.flags_indv.shape)
```

(continues on next page)

(continued from previous page)

```
#print(nn_sis_data.flags_comb.shape)
print('samples: 2^(%d)'%np.log2(nn_sis_data.n_total))
print('time: %.1e'%nn_sis_data.time_integrate)
print('indices:',nn_sis_data.integrand.indices)

import pandas as pd

df_closed = pd.DataFrame(nn_sis[0],columns=target_names,index=[str(idx) for idx in nn_
↪sis_data.integrand.indices])
print('\nClosed Indices')
print(df_closed)
df_total = pd.DataFrame(nn_sis[1],columns=target_names,index=[str(idx) for idx in nn_sis_
↪data.integrand.indices])
print('\nTotal Indices')
print(df_total)
df_closed_singletons = df_closed.T.iloc[:,:4]
df_closed_singletons['sum singletons'] = df_closed_singletons[['[%d]'%i for i in␣
↪range(4)]].sum(1)
df_closed_singletons.columns = data['feature_names']+['sum']
df_closed_singletons = df_closed_singletons*100

import warnings
warnings.simplefilter('ignore',FutureWarning)
#if root: df_closed_singletons.to_latex(root+'si_singletons_closed.tex',float_format='%.1f%
↪%')
```

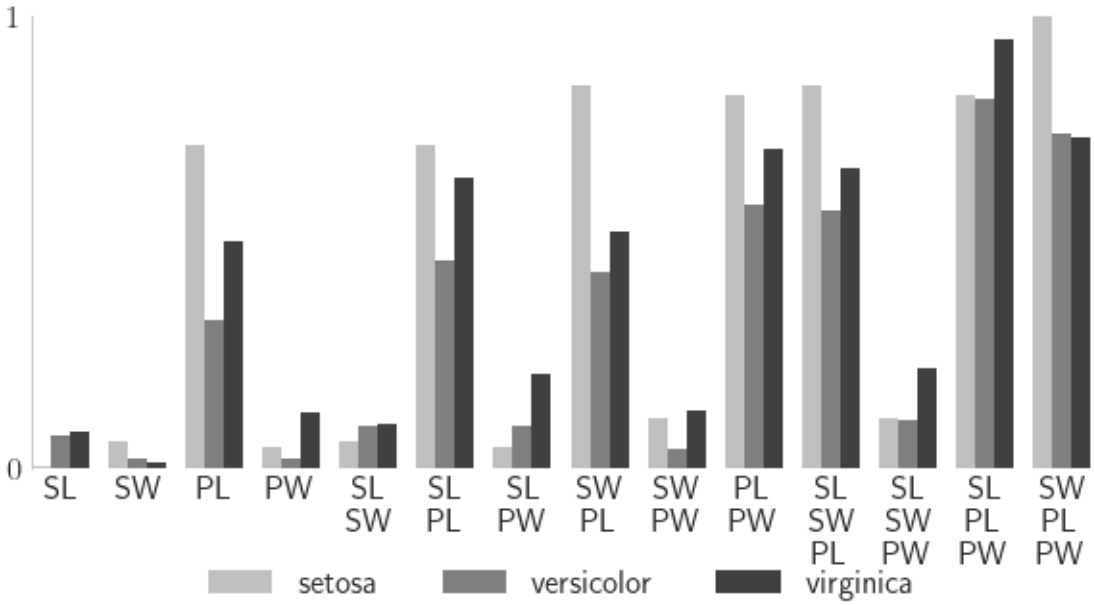
```
samples: 2^(15)
time: 1.7e+00
indices: [[0], [1], [2], [3], [0, 1], [0, 2], [0, 3], [1, 2], [1, 3], [2, 3], [0, 1, 2],␣
↪[0, 1, 3], [0, 2, 3], [1, 2, 3]]

Closed Indices
             setosa  versicolor  virginica
[0]        0.001504    0.071122   0.081736
[1]        0.058743    0.022073   0.010373
[2]        0.713777    0.328313   0.500059
[3]        0.046053    0.021077   0.120233
[0, 1]     0.059178    0.091764   0.098233
[0, 2]     0.715117    0.460138   0.642551
[0, 3]     0.046859    0.092322   0.207724
[1, 2]     0.843241    0.434629   0.520469
[1, 3]     0.108872    0.039572   0.127844
[2, 3]     0.823394    0.582389   0.705354
[0, 1, 2]  0.845359    0.570022   0.661100
[0, 1, 3]  0.108503    0.106081   0.218971
[0, 2, 3]  0.825389    0.814286   0.948331
[1, 2, 3]  0.996483    0.738213   0.729940

Total Indices
             setosa  versicolor  virginica
[0]        0.003199    0.259762   0.265085
[1]        0.172391    0.183159   0.045582
```

(continues on next page)

(continued from previous page)

```
[2]        0.889677    0.896874   0.780377
[3]        0.157794    0.433342   0.340092
[0, 1]     0.174905    0.414246   0.289565
[0, 2]     0.890477    0.966238   0.871992
[0, 3]     0.159744    0.566187   0.478082
[1, 2]     0.949190    0.907994   0.790445
[1, 3]     0.283542    0.540486   0.355714
[2, 3]     0.941651    0.919005   0.902203
[0, 1, 2]  0.949431    0.980367   0.880283
[0, 1, 3]  0.284118    0.674406   0.497364
[0, 2, 3]  0.942185    0.986186   0.989555
[1, 2, 3]  0.996057    0.933342   0.913711
```

```
nindices = len(nn_sis_data.integrand.indices)
fig,ax = pyplot.subplots(figsize=(6,3.5))
ticks = np.arange(nindices)
width = .25
for i,(alpha,species) in enumerate(zip([.25,.5,.75],data['target_names'])):
    cvals = df_closed[species].to_numpy()
    tvals = df_total[species].to_numpy()
    ticks_i = ticks+i*width
    ax.bar(ticks_i,cvals,width=width,align='edge',color='k',alpha=alpha,label=species)
    #ax.bar(ticks_i,np.flip(tvals),width=width,align='edge',bottom=1-np.flip(tvals),
↪color=color,alpha=.1)
ax.set_xlim([0,13+3*width])
ax.set_xticks(ticks+1.5*width)

# closed_labels = [r'$\underline{s}_{\{%s\}}$'%(','.join([r'\text{%s}'%feature_names[i] for␣
↪i in idx])) for idx in nn_sis_data.integrand.indices]
closed_labels = ['\n'.join([feature_names[i] for i in idx]) for idx in nn_sis_data.
↪integrand.indices]
ax.set_xticklabels(closed_labels,rotation=0)
ax.set_ylim([0,1]); ax.set_yticks([0,1])
ax.grid(False)
for spine in ['top','right','bottom']: ax.spines[spine].set_visible(False)
ax.legend(frameon=False,loc='upper center',bbox_to_anchor=(.5,-.2),ncol=3);
if root: fig.savefig(root+'nn_si.pdf')
```



5.21 Vectorized QMC (Bayesian)

```
import qmcpy as qp
import numpy as np
```

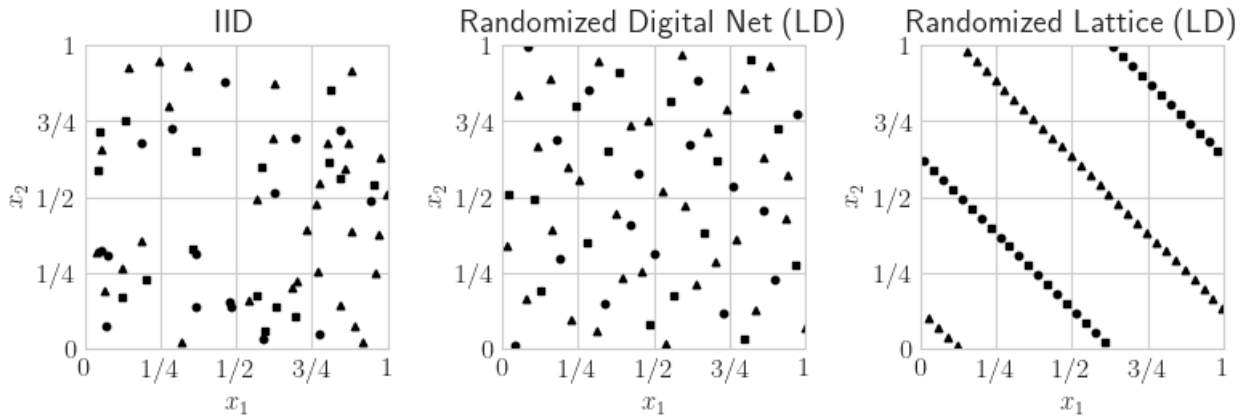
```
from matplotlib import pyplot
pyplot.style.use('../qmcpy/qmcpy.mplstyle')
%matplotlib inline
root_dir = '../_ags/vec_qmc_out/'
```

```
import os
if not os.path.exists(root_dir): os.makedirs(root_dir, exist_ok=True)

!pip install scikit-learn
```

```
Requirement already satisfied: scikit-learn in c:usersaaditminiconda3envsqmcpylibsite-packages␣
↪(1.2.2)
Requirement already satisfied: numpy>=1.17.3 in c:usersaaditminiconda3envsqmcpylibsite-packages␣
↪(from scikit-learn) (1.23.4)
Requirement already satisfied: joblib>=1.1.1 in c:usersaaditminiconda3envsqmcpylibsite-packages␣
↪(from scikit-learn) (1.2.0)
Requirement already satisfied: scipy>=1.3.2 in c:usersaaditminiconda3envsqmcpylibsite-packages␣
↪(from scikit-learn) (1.9.3)
Requirement already satisfied: threadpoolctl>=2.0.0 in c:usersaaditminiconda3envsqmcpylibsite-packages␣
↪(from scikit-learn) (3.1.0)
```

5.21.1 LD Sequence

```
n = 2**6
s = 10
fig,ax = pyplot.subplots(figsize=(8,4),nrows=1,ncols=3)
for i,(dd,name) in enumerate(zip(
    [qp.IIDStdUniform(2,seed=7),qp.DigitalNetB2(2,seed=7),qp.Lattice(2,seed=7)],
    ['IID','Randomized Digital Net (LD)','Randomized Lattice (LD)'])):
    pts = dd.gen_samples(n)
    ax[i].scatter(pts[0:n//4,0],pts[0:n//4,1],color='k',marker='s',s=s)
    ax[i].scatter(pts[n//4:n//2,0],pts[n//4:n//2,1],color='k',marker='o',s=s)
    ax[i].scatter(pts[n//2:n,0],pts[n//2:n,1],color='k',marker='^',s=s)
    ax[i].set_aspect(1)
    ax[i].set_xlabel(r'$x_{1}$')
    ax[i].set_ylabel(r'$x_{2}$')
    ax[i].set_xlim([0,1])
    ax[i].set_ylim([0,1])
    ax[i].set_xticks([0,.25,.5,.75,1])
    ax[i].set_xticklabels([r'$0$',r'$1/4$',r'$1/2$',r'$3/4$',r'$1$'])
    ax[i].set_yticks([0,.25,.5,.75,1])
    ax[i].set_yticklabels([r'$0$',r'$1/4$',r'$1/2$',r'$3/4$',r'$1$'])
    ax[i].set_title(name)
if os.path.exists(root_dir): fig.savefig(root_dir+'ld_seqs.pdf',transparent=True)
```



5.21.2 Simple Example

```
def cantilever_beam_function(T,compute_flags): # T is (n x 3)
    Y = np.zeros((len(T),2),dtype=float) # (n x 2)
    l,w,t = 100,4,2
    T1,T2,T3 = T[:,0],T[:,1],T[:,2] # Python is indexed from 0
    if compute_flags[0]: # compute D. x^2 is "x**2" in Python
        Y[:,0] = 4*l**3/(T1*w*t)*np.sqrt(T2**2/t**4+T3**2/w**4)
    if compute_flags[1]: # compute S
        Y[:,1] = 600*(T2/(w*t**2)+T3/(w**2*t))
    return Y
true_measure = qp.Gaussian(
    sampler = qp.DigitalNetB2(dimension=3,seed=7),
```

(continues on next page)

(continued from previous page)

```
    mean = [2.9e7,500,1000],
    covariance = np.diag([(1.45e6)**2,(100)**2,(100)**2]))
integrand = qp.CustomFun(true_measure,
    g = cantilever_beam_function,
    dimension_indv = 2)
qmc_stop_crit = qp.CubBayesNetG(integrand,
    abs_tol = 1e-3,
    rel_tol = 1e-6)
solution,data = qmc_stop_crit.integrate()
print(solution)
# [2.42575885e+00 3.74999973e+04]
```

```
[2.42575885e+00 3.74999973e+04]
```

5.21.3 BO QEI

See the QEI Demo in QMCPy or the BoTorch Acquision documentation for details on Bayesian Optimization using q-Expected Improvement.

```
import scipy
from sklearn.gaussian_process import GaussianProcessRegressor,kernels

f = lambda x: np.cos(10*x)*np.exp(.2*x)+np.exp(-5*(x-.4)**2)
xplt = np.linspace(0,1,100)
yplt = f(xplt)
x = np.array([.1, .2, .4, .7, .9])
y = f(x)
ymax = y.max()

gp = GaussianProcessRegressor(kernel=kernels.RBF(length_scale=1.0,length_scale_
↪bounds=(1e-2, 1e2)),
    n_restarts_optimizer = 16).fit(x[:,None],y)
yhatplt,stdhatplt = gp.predict(xplt[:,None],return_std=True)

tpax = 32
x0mesh,x1mesh = np.meshgrid(np.linspace(0,1,tpax),np.linspace(0,1,tpax))
post_mus = np.zeros((tpax,tpax,2),dtype=float)
post_sqrtcovs = np.zeros((tpax,tpax,2,2),dtype=float)
for j0 in range(tpax):
    for j1 in range(tpax):
        candidate = np.array([[x0mesh[j0,j1]],[x1mesh[j0,j1]]])
        post_mus[j0,j1],post_cov = gp.predict(candidate,return_cov=True)
        evals,evecs = scipy.linalg.eig(post_cov)
        post_sqrtcovs[j0,j1] = np.sqrt(np.maximum(evals.real,0))*evecs

def qei_acq_vec(x,compute_flags):
    xgauss = scipy.stats.norm.ppf(x)
    n = len(x)
    qei_vals = np.zeros((n,tpax,tpax),dtype=float)
    for j0 in range(tpax):
```

(continues on next page)

(continued from previous page)

```
        for j1 in range(tpax):
            if compute_flags[j0,j1]==False: continue
            sqrt_cov = post_sqrtcovs[j0,j1]
            mu_post = post_mus[j0,j1]
            for i in range(len(x)):
                yij = sqrt_cov@xgauss[i]+mu_post
                qei_vals[i,j0,j1] = max((yij-ymax).max(),0)
    return qei_vals

qei_acq_vec_qmcpy = qp.CustomFun(
    true_measure = qp.Uniform(qp.DigitalNetB2(2,seed=7)),
    g = qei_acq_vec,
    dimension_indv = (tpax,tpax),
    parallel=False)
qei_vals,qei_data = qp.CubBayesNetG(qei_acq_vec_qmcpy,abs_tol=.025,rel_tol=0).
↪integrate() # .0005
print(qei_data)

a = np.unravel_index(np.argmax(qei_vals,axis=None),qei_vals.shape)
xnext = np.array([x0mesh[a[0],a[1]],x1mesh[a[0],a[1]]])
fnext = f(xnext)
```

```
LDTransformBayesData (AccumulateData Object)
    solution        [[0.059 0.079 0.071 ... 0.059 0.059 0.066]
                    [0.079 0.064 0.067 ... 0.064 0.064 0.07 ]
                    [0.072 0.067 0.032 ... 0.032 0.032 0.038]
                    ...
                    [0.059 0.064 0.032 ... 0.    0.    0.006]
                    [0.06  0.064 0.032 ... 0.    0.    0.006]
                    [0.064 0.069 0.037 ... 0.006 0.006 0.006]]
    comb_bound_low  [[ 5.659e-02  7.572e-02  6.764e-02 ...  5.647e-02  5.647e-02  6.252e-
↪02]
                    [ 7.553e-02  6.151e-02  6.433e-02 ...  6.149e-02  6.173e-02  6.691e-
↪02]
                    [ 6.814e-02  6.443e-02  3.076e-02 ...  3.086e-02  3.086e-02  3.616e-
↪02]
                    ...
                    [ 5.659e-02  6.151e-02  3.071e-02 ... -2.220e-16  4.341e-05  4.142e-
↪03]
                    [ 5.700e-02  6.146e-02  3.077e-02 ...  5.235e-05 -2.220e-16  3.939e-
↪03]
                    [ 6.112e-02  6.595e-02  3.503e-02 ...  4.030e-03  3.668e-03  4.628e-
↪03]]
    comb_bound_high [[6.189e-02 8.222e-02 7.467e-02 ... 6.187e-02 6.187e-02 6.957e-02]
                    [8.184e-02 6.586e-02 6.933e-02 ... 6.592e-02 6.669e-02 7.316e-02]
                    [7.496e-02 6.935e-02 3.321e-02 ... 3.366e-02 3.366e-02 4.075e-02]
                    ...
                    [6.189e-02 6.586e-02 3.306e-02 ... 2.220e-16 5.695e-04 8.593e-03]
                    [6.238e-02 6.563e-02 3.312e-02 ... 6.869e-04 2.220e-16 8.591e-03]
                    [6.781e-02 7.199e-02 3.988e-02 ... 8.594e-03 8.541e-03 8.267e-03]]
    comb_flags      [[ True  True  True ...  True  True  True]
                    [ True  True  True ...  True  True  True]
```

(continues on next page)

(continued from previous page)

```
                    [ True  True  True ...  True  True  True]
                    ...
                    [ True  True  True ...  True  True  True]
                    [ True  True  True ...  True  True  True]
                    [ True  True  True ...  True  True  True]]
    n_total         2^(8)
    n               [[256. 256. 256. ... 256. 256. 256.]
                    [256. 256. 256. ... 256. 256. 256.]
                    [256. 256. 256. ... 256. 256. 256.]
                    ...
                    [256. 256. 256. ... 256. 256. 256.]
                    [256. 256. 256. ... 256. 256. 256.]
                    [256. 256. 256. ... 256. 256. 256.]]
    time_integrate  7.904
CubBayesNetG (StoppingCriterion Object)
    abs_tol         0.025
    rel_tol         0
    n_init          2^(8)
    n_max           2^(22)
CustomFun (Integrand Object)
Uniform (TrueMeasure Object)
    lower_bound     0
    upper_bound     1
DigitalNetB2 (DiscreteDistribution Object)
    d               2^(1)
    dvec            [0 1]
    randomize       LMS_DS
    graycode        0
    entropy         7
    spawn_key       ()
```

```
from matplotlib import cm
fig,ax = pyplot.subplots(nrows=1,ncols=2,figsize=(10,4))
ax[0].scatter(x,y,color='k',label='Query Points')
ax[0].plot(xplt,yplt,color='k',linestyle='--',label='True function',linewidth=1)
ax[0].plot(xplt,yhatplt,color='k',label='GP Mean',linewidth=1)
ax[0].fill_between(xplt,yhatplt-1.96*stdhatplt,yhatplt+1.96*stdhatplt,color='k',alpha=.
↪25,label='95% CI')
ax[0].scatter(xnext,fnext,color='k',marker='*',s=200,zorder=10)
ax[0].set_xlim([0,1])
ax[0].set_xticks([0,1])
ax[0].set_xlabel(r'$x$')
ax[0].set_ylabel(r'$y$')
fig.legend(labels=['data','true function','posterior mean',r'95\% CI','next points by QEI
↪'],loc='lower center',bbox_to_anchor=(.5,-.05),ncol=5)
contour = ax[1].contourf(x0mesh,x1mesh,qei_vals,cmap=cm.Greys_r)
ax[1].scatter([xnext[0]],[xnext[1]],color='k',marker='*',s=200)
fig.colorbar(contour,ax=None,shrink=1,aspect=5)
ax[1].scatter(x0mesh.flatten(),x1mesh.flatten(),color='w',s=1)
ax[1].set_xlim([0,1])
ax[1].set_xticks([0,1])
ax[1].set_ylim([0,1])
```

(continues on next page)

(continued from previous page)

```
ax[1].set_yticks([0,1])
ax[1].set_xlabel(r'$x_1$')
ax[1].set_ylabel(r'$x_2$')
if os.path.exists(root_dir): fig.savefig(root_dir+'gp.pdf',transparent=True)
```

5.21.4 Bayesian Logistic Regression

```
import pandas as pd
from sklearn.model_selection import train_test_split
df = pd.read_csv('https://archive.ics.uci.edu/ml/machine-learning-databases/haberman/
↪haberman.data',header=None)
df.columns = ['Age','1900 Year','Axillary Nodes','Survival Status']
df.loc[df['Survival Status']==2,'Survival Status'] = 0
x,y = df[['Age','1900 Year','Axillary Nodes']],df['Survival Status']
xt,xv,yt,yv = train_test_split(x,y,test_size=.33,random_state=7)
```

```
print(df.head(),'\n')
print(df[['Age','1900 Year','Axillary Nodes']].describe(),'\n')
print(df['Survival Status'].astype(str).describe())
print('\ntrain samples: %d test samples: %d\n'%(len(xt),len(xv)))
print('train positives %d   train negatives: %d'%(np.sum(yt==1),np.sum(yt==0)))
print(' test positives %d    test negatives: %d'%(np.sum(yv==1),np.sum(yv==0)))
xt.head()
```

```
   Age  1900 Year  Axillary Nodes  Survival Status
0   30         64               1                1
1   30         62               3                1
2   30         65               0                1
3   31         59               2                1
4   31         65               4                1

              Age   1900 Year  Axillary Nodes
count  306.000000  306.000000      306.000000
mean    52.457516   62.852941        4.026144
std     10.803452    3.249405        7.189654
min     30.000000   58.000000        0.000000
25%     44.000000   60.000000        0.000000
50%     52.000000   63.000000        1.000000
75%     60.750000   65.750000        4.000000
max     83.000000   69.000000       52.000000

count     306
unique      2
top         1
freq      225
Name: Survival Status, dtype: object

train samples: 205 test samples: 101

train positives 151   train negatives: 54
```

(continues on next page)

(continued from previous page)

```
test positives 74    test negatives: 27
```

```
# fails with MaxSamplesWarning
#
# blr = qp.BayesianLRCoeffs(
#     sampler = qp.DigitalNetB2(4,seed=7),
#     feature_array = xt, # np.ndarray of shape (n,d-1)
#     response_vector = yt, # np.ndarray of shape (n,)
#     prior_mean = 0, # normal prior mean = (0,0,...,0)
#     prior_covariance = 5) # normal prior covariance = 5I
# qmc_sc = qp.CubBayesNetG(blr,
#     abs_tol = .05,
#     rel_tol = .5,
#     error_fun = lambda s,abs_tols,rel_tols:
#         np.minimum(abs_tols,np.abs(s)*rel_tols))
# blr_coefs,blr_data = qmc_sc.integrate()
# print(blr_data)

# LDTransformData (AccumulateData Object)
#     solution        [-0.004  0.13  -0.157  0.008]
#     comb_bound_low  [-0.006  0.092 -0.205  0.007]
#     comb_bound_high [-0.003  0.172 -0.109  0.012]
#     comb_flags      [ True  True  True  True]
#     n_total         2^(18)
#     n               [[  1024.   1024. 262144.   2048.]
#                     [  1024.   1024. 262144.   2048.]]
#     time_integrate  2.229
```

```
# from sklearn.linear_model import LogisticRegression
# def metrics(y,yhat):
#     y,yhat = np.array(y),np.array(yhat)
#     tp = np.sum((y==1)*(yhat==1))
#     tn = np.sum((y==0)*(yhat==0))
#     fp = np.sum((y==0)*(yhat==1))
#     fn = np.sum((y==1)*(yhat==0))
#     accuracy = (tp+tn)/(len(y))
#     precision = tp/(tp+fp)
#     recall = tp/(tp+fn)
#     return [accuracy,precision,recall]

# results = pd.DataFrame({name:[] for name in ['method','Age','1900 Year','Axillary Nodes',
↪'Intercept','Accuracy','Precision','Recall']})
# for i,l1_ratio in enumerate([0,.5,1]):
#     lr = LogisticRegression(random_state=7,penalty="elasticnet",solver='saga',l1_
↪ratio=l1_ratio).fit(xt,yt)
#     results.loc[i] = [r'Elastic-Net \lambda=%.1f'%l1_ratio]+lr.coef_.squeeze().
↪tolist()+[lr.intercept_.item()]+metrics(yv,lr.predict(xv))

# blr_predict = lambda x: 1/(1+np.exp(-np.array(x)@blr_coefs[:-1]-blr_coefs[-1]))>=.5
# blr_train_accuracy = np.mean(blr_predict(xt)==yt)
# blr_test_accuracy = np.mean(blr_predict(xv)==yv)
```

(continues on next page)

(continued from previous page)

```
# results.loc[len(results)] = ['Bayesian']+blr_coefs.squeeze().tolist()+metrics(yv,blr_
↪predict(xv))

# import warnings
# warnings.simplefilter('ignore',FutureWarning)
# results.set_index('method',inplace=True)
# print(results.head())
#root_dir: results.to_latex(root_dir+'lr_table.tex',formatters={'%s'%tt:lambda v:'%.1f'
↪%(100*v) for tt in ['accuracy','precision','recall']},float_format="%.2e")
```

5.21.5 Sensitivity Indices

Ishigami Function

```
a,b = 7,0.1
dnb2 = qp.DigitalNetB2(3,seed=7)
ishigami = qp.Ishigami(dnb2,a,b)
idxs =[[0], [1], [2], [0,1], [0,2], [1,2]]
ishigami_si = qp.SensitivityIndices(ishigami,idxs)
qmc_algo = qp.CubBayesNetG(ishigami_si,abs_tol=.05)
solution,data = qmc_algo.integrate()
print(data)
si_closed = solution[0].squeeze()
si_total = solution[1].squeeze()
ci_comb_low_closed = data.comb_bound_low[0].squeeze()
ci_comb_high_closed = data.comb_bound_high[0].squeeze()
ci_comb_low_total = data.comb_bound_low[1].squeeze()
ci_comb_high_total = data.comb_bound_high[1].squeeze()
print("\nApprox took %.1f sec and n = 2^(%d)"%
    (data.time_integrate,np.log2(data.n_total)))
print('\t si_closed:',si_closed)
print('\t si_total:',si_total)
print('\t ci_comb_low_closed:',ci_comb_low_closed)
print('\t ci_comb_high_closed:',ci_comb_high_closed)
print('\t ci_comb_low_total:',ci_comb_low_total)
print('\t ci_comb_high_total:',ci_comb_high_total)

true_indices = qp.Ishigami._exact_sensitivity_indices(idxs,a,b)
si_closed_true = true_indices[0]
si_total_true = true_indices[1]
```

```
LDTransformBayesData (AccumulateData Object)
    solution        [[0.317 0.445 0.027 0.761 0.56  0.444]
                    [0.563 0.438 0.254 0.973 0.558 0.685]]
    comb_bound_low  [[0.29  0.398 0.    0.714 0.523 0.407]
                    [0.526 0.404 0.212 0.947 0.528 0.648]]
    comb_bound_high [[0.345 0.492 0.055 0.808 0.597 0.481]
                    [0.6   0.472 0.296 1.    0.588 0.722]]
    comb_flags      [[ True  True  True  True  True  True]
                    [ True  True  True  True  True  True]]
```

(continues on next page)

(continued from previous page)

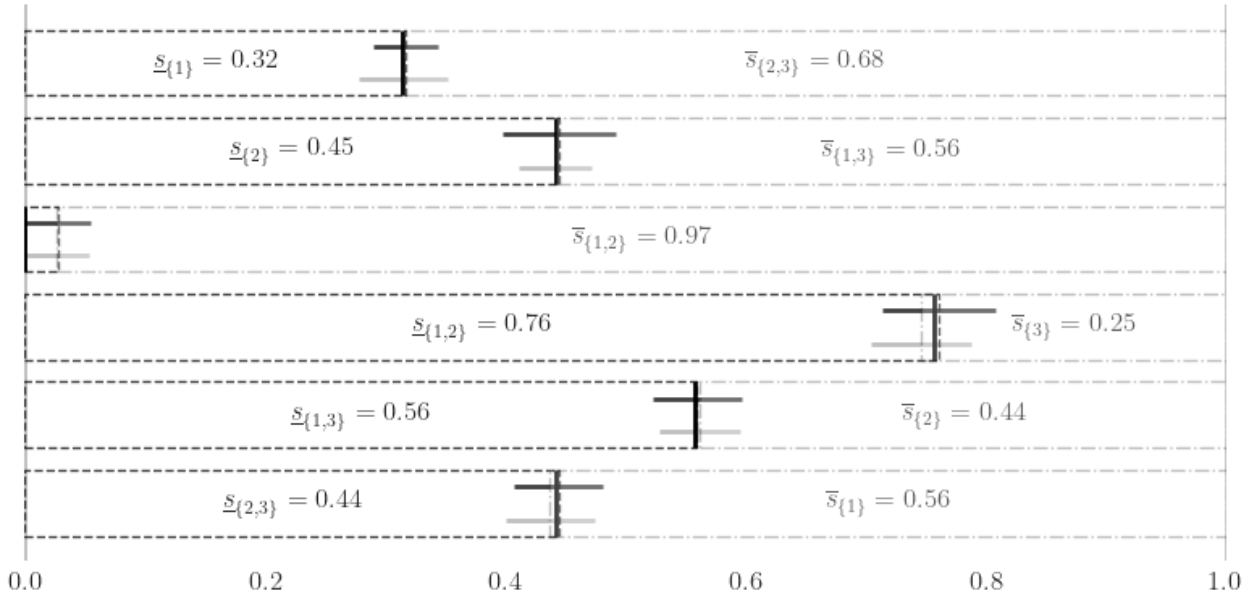
```
    n_total         2^(11)
    n               [[[2048. 1024. 1024. 2048. 2048. 2048.]
                      [2048. 1024. 1024. 2048. 2048. 2048.]
                      [2048. 1024. 1024. 2048. 2048. 2048.]]

                     [[2048. 1024.  512. 2048. 2048. 2048.]
                      [2048. 1024.  512. 2048. 2048. 2048.]
                      [2048. 1024.  512. 2048. 2048. 2048.]]]
    time_integrate  0.823
CubBayesNetG (StoppingCriterion Object)
    abs_tol         0.050
    rel_tol         0
    n_init          2^(8)
    n_max           2^(22)
SensitivityIndices (Integrand Object)
    indices         [[0], [1], [2], [0, 1], [0, 2], [1, 2]]
    n_multiplier    6
Uniform (TrueMeasure Object)
    lower_bound     -3.142
    upper_bound     3.142
DigitalNetB2 (DiscreteDistribution Object)
    d               6
    dvec            [0 1 2 3 4 5]
    randomize       LMS_DS
    graycode        0
    entropy         7
    spawn_key       (0,)

Approx took 0.8 sec and n = 2^(11)
     si_closed: [0.31722253 0.44506938 0.0273244  0.76087937 0.55981713 0.44420515]
     si_total: [0.56290907 0.43784863 0.25397723 0.97329123 0.55830656 0.68479275]
     ci_comb_low_closed: [0.28990739 0.39777074 0.         0.71378418 0.52252396 0.
↪40694415]
     ci_comb_high_closed: [0.34453767 0.49236803 0.0546488  0.80797455 0.59711031 0.
↪48146614]
     ci_comb_low_total: [0.52579416 0.4039169  0.21233444 0.94658246 0.52841671 0.
↪64777255]
     ci_comb_high_total: [0.60002398 0.47178035 0.29562002 1.         0.58819641 0.
↪72181294]
```

```
fig,ax = pyplot.subplots(figsize=(8,4))
ax.grid(False)
for spine in ['top','left','right','bottom']: ax.spines[spine].set_visible(False)
width = .75
ax.errorbar(fmt='none',color='k',
    x = 1-si_total_true,
    y = np.arange(len(si_closed)),
    xerr = 0,
    yerr = width/2,
    alpha = 1)
bar_closed = ax.barh(np.arange(len(si_closed)),np.flip(si_closed),width,label='Closed SI
↪',color='w',edgecolor='k',alpha=.75,linestyle='--')
```

(continues on next page)

(continued from previous page)

```
ax.errorbar(fmt='none',color='k',
    x = si_closed,
    y = np.flip(np.arange(len(si_closed)))+width/4,
    xerr = np.vstack((si_closed-ci_comb_low_closed,ci_comb_high_closed-si_closed)),
    yerr = 0,
    #elinewidth = 5,
    alpha = .75)
bar_total = ax.barh(np.arange(len(si_closed)),si_total,width,label='Total SI',color='w',
↪alpha=.25,edgecolor='k',left=1-si_total,zorder=10,linestyle='-.')
ax.errorbar(fmt='none',color='k',
    x = 1-si_total,
    y = np.arange(len(si_closed))-width/4,
    xerr = np.vstack((si_total-ci_comb_low_total,ci_comb_high_total-si_total)),
    yerr = 0,
    #elinewidth = 5,
    alpha = .25)
closed_labels = [r'$\underline{s}_{\{%s\}} = %.2f$'%(','.join([str(i+1) for i in idx]),
↪c) for idx,c in zip(idxs[::-1],np.flip(si_closed))]
closed_labels[3] = ''
total_labels = [r'$\overline{s}_{\{%s\}} = %.2f$'%(','.join([str(i+1) for i in idx]),t)␣
↪for idx,t in zip(idxs,si_total)]
ax.bar_label(bar_closed,label_type='center',labels=closed_labels)
ax.bar_label(bar_total,label_type='center',labels=total_labels)
ax.set_xlim([-.001,1.001])
ax.axvline(x=0,ymin=0,ymax=len(si_closed),color='k',alpha=.25)
ax.axvline(x=1,ymin=0,ymax=len(si_closed),color='k',alpha=.25)
ax.set_yticklabels([])
if os.path.exists(root_dir): fig.savefig(root_dir+'ishigami.pdf',transparent=True)
```

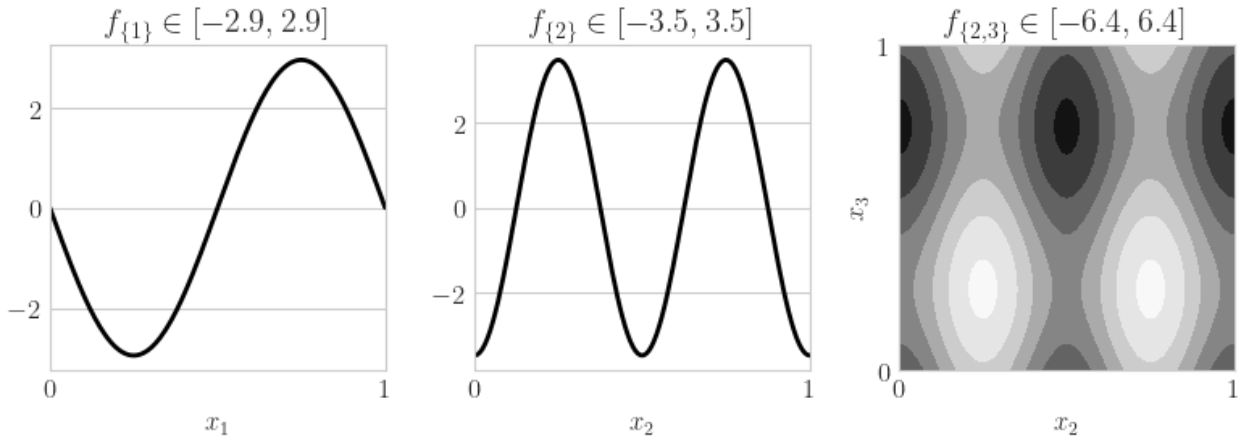



```
fig,ax = pyplot.subplots(nrows=1,ncols=3,figsize=(8,3))
x_1d = np.linspace(0,1,num=128)
x_1d_mat = np.tile(x_1d,(3,1)).T
```

(continues on next page)

(continued from previous page)

```
y_1d = qp.Ishigami._exact_fu_functions(x_1d_mat,indices=[[0],[1],[2]],a=a,b=b)
for i in range(2):
    ax[i].plot(x_1d,y_1d[:,i],color='k')
    ax[i].set_xlim([0,1])
    ax[i].set_xticks([0,1])
    ax[i].set_xlabel(r'$x_{%d}$'%(i+1))
    ax[i].set_title(r'$f_{\{%d\}} \in [%.1f,%.1f]$'%(i+1,y_1d[:,i].min(),y_1d[:,i].
↪max()))
x_mesh,y_mesh = np.meshgrid(x_1d,x_1d)
xquery = np.zeros((x_mesh.size,3))
for i,idx in enumerate([[1,2]]): # [[0,1],[0,2],[1,2]]
    xquery[:,idx[0]] = x_mesh.flatten()
    xquery[:,idx[1]] = y_mesh.flatten()
    zquery = qp.Ishigami._exact_fu_functions(xquery,indices=[idx],a=a,b=b)
    z_mesh = zquery.reshape(x_mesh.shape)
    ax[2+i].contourf(x_mesh,y_mesh,z_mesh,cmap=cm.Greys_r)
    ax[2+i].set_xlabel(r'$x_{%d}$'%(idx[0]+1))
    ax[2+i].set_ylabel(r'$x_{%d}$'%(idx[1]+1))
    ax[2+i].set_title(r'$f_{\{%d,%d\}} \in [%.1f,%.1f]$'%(tuple([i+1 for i in idx])+(z_
↪mesh.min(),z_mesh.max())))
    ax[2+i].set_xlim([0,1])
    ax[2+i].set_ylim([0,1])
    ax[2+i].set_xticks([0,1])
    ax[2+i].set_yticks([0,1])
if os.path.exists(root_dir): fig.savefig(root_dir+'ishigami_fu.pdf')
```



Neural Network

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPClassifier
data = load_iris()
feature_names = data["feature_names"]
feature_names = [fn.replace('sepal ','S')\
    .replace('length ','L')\
    .replace('petal ','P')\
```

(continues on next page)

(continued from previous page)

```
    .replace('width ','W')\
    .replace('(cm)','') for fn in feature_names]
target_names = data["target_names"]
xt,xv,yt,yv = train_test_split(data["data"],data["target"],
    test_size = 1/3,
    random_state = 7)
```

```
mlpc = MLPClassifier(random_state=7,max_iter=1024).fit(xt,yt)
yhat = mlpc.predict(xv)
print("accuracy: %.1f%%"%(100*(yv==yhat).mean()))
# accuracy: 98.0%
sampler = qp.DigitalNetB2(4,seed=7)
true_measure =  qp.Uniform(sampler,
    lower_bound = xt.min(0),
    upper_bound = xt.max(0))
fun = qp.CustomFun(
    true_measure = true_measure,
    g = lambda x,compute_flags: mlpc.predict_proba(x),
    dimension_indv = 3)
si_fun = qp.SensitivityIndices(fun,indices="all")
qmc_algo = qp.CubBayesNetG(si_fun,abs_tol=.005)
nn_sis,nn_sis_data = qmc_algo.integrate()
```

```
accuracy: 98.0%
```

```
#print(nn_sis_data.flags_indv.shape)
#print(nn_sis_data.flags_comb.shape)
print('samples: 2^(%d)'%np.log2(nn_sis_data.n_total))
print('time: %.1e'%nn_sis_data.time_integrate)
print('indices:',nn_sis_data.integrand.indices)

import pandas as pd

df_closed = pd.DataFrame(nn_sis[0],columns=target_names,index=[str(idx) for idx in nn_
↪sis_data.integrand.indices])
print('\nClosed Indices')
print(df_closed)
df_total = pd.DataFrame(nn_sis[1],columns=target_names,index=[str(idx) for idx in nn_sis_
↪data.integrand.indices])
print('\nTotal Indices')
print(df_total)
df_closed_singletons = df_closed.T.iloc[:,:4]
df_closed_singletons['sum singletons'] = df_closed_singletons[['[%d]'%i for i in␣
↪range(4)]].sum(1)
df_closed_singletons.columns = data['feature_names']+['sum']
df_closed_singletons = df_closed_singletons*100

import warnings
warnings.simplefilter('ignore',FutureWarning)
#if os.path.exists(root_dir): df_closed_singletons.to_latex(root_dir+'si_singletons_
↪closed.tex',float_format='%.1f%%')
```

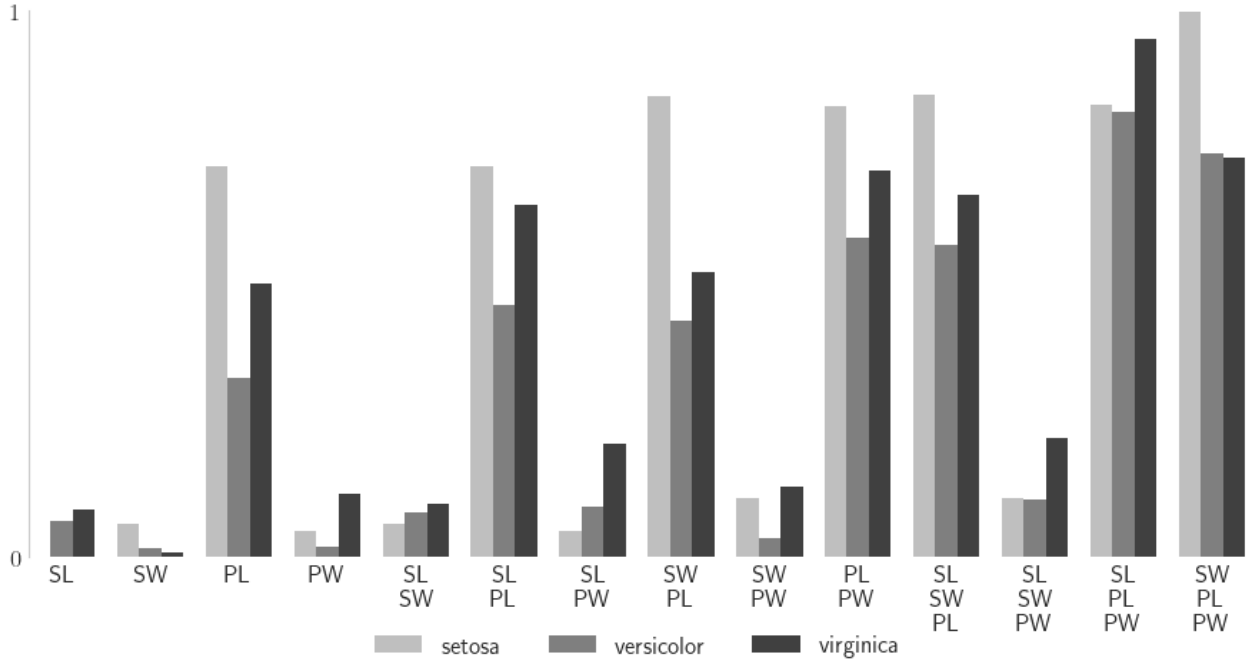
```
samples: 2^(17)
time: 1.5e+02
indices: [[0], [1], [2], [3], [0, 1], [0, 2], [0, 3], [1, 2], [1, 3], [2, 3], [0, 1, 2],␣
↪[0, 1, 3], [0, 2, 3], [1, 2, 3]]

Closed Indices
             setosa  versicolor  virginica
[0]        0.002241    0.066824   0.087086
[1]        0.060509    0.018068   0.009578
[2]        0.714576    0.327246   0.500785
[3]        0.048713    0.020175   0.117416
[0, 1]     0.061000    0.081183   0.098404
[0, 2]     0.715994    0.461352   0.643425
[0, 3]     0.049185    0.092441   0.207293
[1, 2]     0.843096    0.432901   0.520253
[1, 3]     0.108521    0.034234   0.129580
[2, 3]     0.824629    0.583650   0.706950
[0, 1, 2]  0.845177    0.571376   0.663010
[0, 1, 3]  0.108856    0.105634   0.219203
[0, 2, 3]  0.826709    0.815249   0.948155
[1, 2, 3]  0.996158    0.738581   0.730320

Total Indices
             setosa  versicolor  virginica
[0]        0.003151    0.263036   0.271131
[1]        0.173527    0.184647   0.051536
[2]        0.889956    0.894582   0.780736
[3]        0.157627    0.429371   0.337218
[0, 1]     0.175846    0.417716   0.293843
[0, 2]     0.890935    0.965843   0.869250
[0, 3]     0.159242    0.566774   0.480126
[1, 2]     0.948939    0.907382   0.791254
[1, 3]     0.283738    0.539776   0.357806
[2, 3]     0.941918    0.919074   0.902720
[0, 1, 2]  0.949492    0.979871   0.880512
[0, 1, 3]  0.285391    0.673551   0.499146
[0, 2, 3]  0.942710    0.987317   0.989545
[1, 2, 3]  0.995549    0.932610   0.914145
```

```
nindices = len(nn_sis_data.integrand.indices)
fig,ax = pyplot.subplots(figsize=(9,5))
ticks = np.arange(nindices)
width = .25
for i,(alpha,species) in enumerate(zip([.25,.5,.75],data['target_names'])):
    cvals = df_closed[species].to_numpy()
    tvals = df_total[species].to_numpy()
    ticks_i = ticks+i*width
    ax.bar(ticks_i,cvals,width=width,align='edge',color='k',alpha=alpha,label=species)
    #ax.bar(ticks_i,np.flip(tvals),width=width,align='edge',bottom=1-np.flip(tvals),
↪color=color,alpha=.1)
ax.set_xlim([0,13+3*width])
ax.set_xticks(ticks+1.5*width)
```

(continues on next page)

(continued from previous page)

```
# closed_labels = [r'$\underline{s}_{\{%s\}}$'%(','.join([r'\text{%s}'%feature_names[i] for
↪i in idx])) for idx in nn_sis_data.integrand.indices]
closed_labels = ['\n'.join([feature_names[i] for i in idx]) for idx in nn_sis_data.
↪integrand.indices]
ax.set_xticklabels(closed_labels,rotation=0)
ax.set_ylim([0,1]); ax.set_yticks([0,1])
ax.grid(False)
for spine in ['top','right','bottom']: ax.spines[spine].set_visible(False)
ax.legend(frameon=False,loc='lower center',bbox_to_anchor=(.5,-.2),ncol=3);
if os.path.exists(root_dir): fig.savefig(root_dir+'nn_si.pdf')
```



5.22 UM-Bridge with QMCPy

Using QMCPy to evaluate the UM-Bridge Cantilever Beam Function and approximate the expectation with respect to a uniform random variable.

5.22.1 Imports

```
import umbridge
import qmcpy as qp
```

5.22.2 Start Docker Container

See the UM-Bridge Documentation for image options.

```
!docker run --name muqbp -d -it -p 4243:4243 linusseelinger/benchmark-muq-beam-
↪propagation:latest
```

```
57a43d7926cf9838fd7273725b2415a5e6c1fe5d131fc5012a8bfbad4f531c0a
```

5.22.3 Problem Setup

Initialize a QMCPy sampler and distribution.

```
sampler = qp.DigitalNetB2(dimension=3,seed=7) # DISCRETE DISTRIBUTION
distribution = qp.Uniform(sampler,lower_bound=1,upper_bound=1.05) # TRUE MEASURE
```

Initialize a UM-Bridge model and wrap it into a QMCPy compatible Integrand

```
model = umbridge.HTTPModel('http://localhost:4243','forward')
umbridge_config = {"d": sampler.d}
integrand = qp.UMBridgeWrapper(distribution,model,umbridge_config,parallel=False) #␣
↪INTEGRAND
```

5.22.4 Model Evaluation

```
x = sampler(16) # same as sampler.gen_samples(16)
y = integrand.f(x)
print(y.shape)
print(type(y))
print(y.dtype)
```

```
(16, 31)
<class 'numpy.ndarray'>
float64
```

5.22.5 Automatically Approximate the Expectation

```
qmc_stop_crit = qp.CubQMCNetG(integrand,abs_tol=2.5e-2) # QMC STOPPING CRITERION
solution,data = qmc_stop_crit.integrate()
print(data)
```

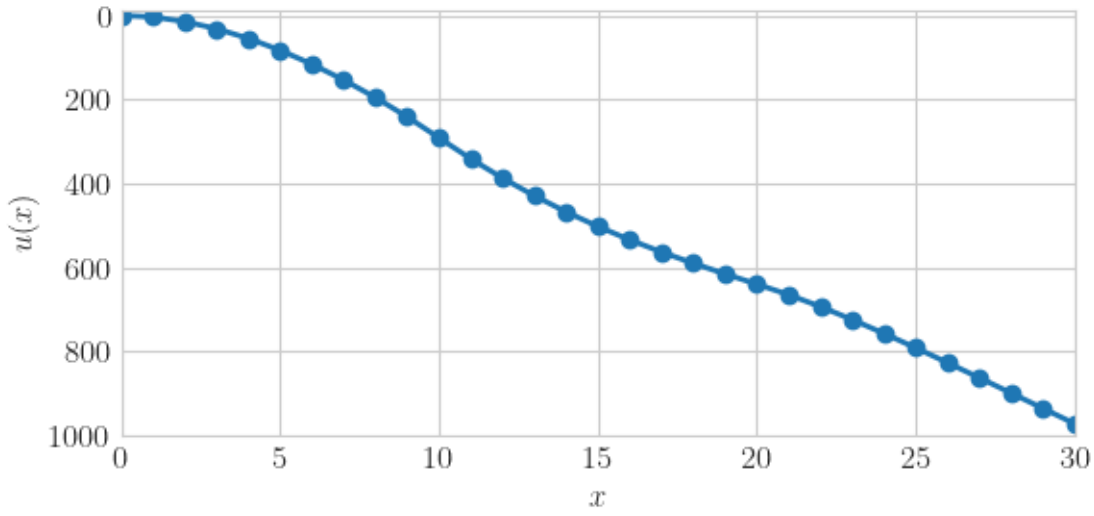
```
LDTransformData (AccumulateData Object)
    solution        [  0.      3.855  14.69  ... 898.921 935.383 971.884]
    comb_bound_low  [  0.      3.854  14.688 ... 898.901 935.363 971.863]
    comb_bound_high [  0.      3.855  14.691 ... 898.941 935.404 971.906]
    comb_flags      [ True  True  True ...  True  True  True]
    n_total         2^(11)
    n               [1024. 1024. 1024. ... 2048. 2048. 2048.]
```

(continues on next page)

(continued from previous page)

```
    time_integrate  25.311
CubQMCNetG (StoppingCriterion Object)
    abs_tol         0.025
    rel_tol         0
    n_init          2^(10)
    n_max           2^(35)
UMBridgeWrapper (Integrand Object)
Uniform (TrueMeasure Object)
    lower_bound     1
    upper_bound     1.050
DigitalNetB2 (DiscreteDistribution Object)
    d               3
    dvec            [0 1 2]
    randomize       LMS_DS
    graycode        0
    entropy         7
    spawn_key       ()
```

```
from matplotlib import pyplot
pyplot.style.use('../qmcpy/qmcpy.mplstyle')
fig,ax = pyplot.subplots(figsize=(6,3))
ax.plot(solution,'-o')
ax.set_xlim([0,len(solution)-1]); ax.set_xlabel(r'$x$')
ax.set_ylim([1000,-10]);  ax.set_ylabel(r'$u(x)$');
```



5.22.6 Parallel Evaluation

QMCPy can automatically multi-threaded requests to the model by setting `parallel=p` where `p` is the number of processors used by multiprocessing.pool.ThreadPool. Setting `parallel=True` is equivalent to setting `paralle=os.cpu_count()`.

```
import os
print('Available CPUs: %d'%os.cpu_count())
```

```
Available CPUs: 12
```

```
integrand = qp.UMBridgeWrapper(distribution,model,umbridge_config,parallel=8)
solution,data = qp.CubQMCNetG(integrand,abs_tol=2.5e-2).integrate()
data
```

```
LDTransformData (AccumulateData Object)
    solution        [  0.      3.855  14.69  ... 898.921 935.383 971.884]
    comb_bound_low  [  0.      3.854  14.688 ... 898.901 935.363 971.863]
    comb_bound_high [  0.      3.855  14.691 ... 898.941 935.404 971.906]
    comb_flags      [ True  True  True ...  True  True  True]
    n_total         2^(11)
    n               [1024. 1024. 1024. ... 2048. 2048. 2048.]
    time_integrate  14.785
CubQMCNetG (StoppingCriterion Object)
    abs_tol         0.025
    rel_tol         0
    n_init          2^(10)
    n_max           2^(35)
UMBridgeWrapper (Integrand Object)
Uniform (TrueMeasure Object)
    lower_bound     1
    upper_bound     1.050
DigitalNetB2 (DiscreteDistribution Object)
    d               3
    dvec            [0 1 2]
    randomize       LMS_DS
    graycode        0
    entropy         7
    spawn_key       ()
```

5.22.7 Shut Down Docker Image

```
!docker rm -f muqbp
```

```
muqbp
```

5.23 Random Lattice Generators Are Not Bad

```
import qmcpy as qp
import numpy as np  #basic numerical routines in Python
import time  #timing routines
from matplotlib import pyplot;  #plotting

pyplot.rc('font', size=16)  #set defaults so that the plots are readable
pyplot.rc('axes', titlesize=16)
pyplot.rc('axes', labelsize=16)
pyplot.rc('xtick', labelsize=16)
pyplot.rc('ytick', labelsize=16)
pyplot.rc('legend', fontsize=16)
pyplot.rc('figure', titlesize=16)

#a helpful plotting method to show increasing numbers of points
def plot_successive_points(distrib,ld_name,first_n=64,n_cols=1,pt_clr='bgkcmy',
                           xlim=[0,1],ylim=[0,1],coord1 = 0,coord2 = 1):
  fig,ax = pyplot.subplots(nrows=1,ncols=n_cols,figsize=(5*n_cols,5.5))
  if n_cols==1: ax = [ax]
  last_n = first_n*(2**n_cols)
  points = distrib.gen_samples(n=last_n)
  for i in range(n_cols):
    n = first_n
    nstart = 0
    for j in range(i+1):
      n = first_n*(2**j)
      ax[i].scatter(points[nstart:n,coord1],points[nstart:n,coord2],color=pt_clr[j])
      nstart = n
    ax[i].set_title('n = %d'%n)
    ax[i].set_xlim(xlim); ax[i].set_xticks(xlim); ax[i].set_xlabel('$x_{i,%d}$'
↪%(coord1+1))
    ax[i].set_ylim(ylim); ax[i].set_yticks(ylim); ax[i].set_ylabel('$x_{i,%d}$'
↪%(coord2+1))
    ax[i].set_aspect((xlim[1]-xlim[0])/(ylim[1]-ylim[0]))
  fig.suptitle('%s Points'%ld_name)
```

5.23.1 Lattice Declaration and the gen_samples function

```
lat = qp.Lattice()
help(lat.__init__)
```

```
Help on method __init__ in module qmcpy.discrete_distribution.lattice.lattice:

__init__(dimension=1, randomize=True, order='natural', seed=None, generating_vector=
↪'lattice_vec.3600.20.npy', d_max=None, m_max=None) method of qmcpy.discrete_
↪distribution.lattice.lattice.Lattice instance
```

(continues on next page)

(continued from previous page)

```
    Args:
        dimension (int or ndarray): dimension of the generator.
            If an int is passed in, use sequence dimensions [0,...,dimensions-1].
            If a ndarray is passed in, use these dimension indices in the sequence.
        randomize (bool): If True, apply shift to generated samples.
            Note: Non-randomized lattice sequence includes the origin.
        order (str): 'linear', 'natural', or 'mps' ordering.
        seed (None or int or numpy.random.SeedSeq): seed the random number generator for␣
↪reproducibility
        generating_vector (ndarray, str or int): generating matrix or path to generating␣
↪matrices.
            ndarray should have shape (d_max).
            a string generating_vector should be formatted like
            'lattice_vec.3600.20.npy' where 'name.d_max.m_max.npy'
            an integer should be an odd larger than 1; passing an integer M would create␣
↪a random generating vector supporting up to 2^M points.
            M is restricted between 2 and 26 for numerical percision. The generating␣
↪vector is [1,v_1,v_2,...,v_dimension], where v_i is an integer in {3,5,...,2*M-1}.
        d_max (int): maximum dimension
        m_max (int): 2^m_max is the max number of supported samples

    Note:
        d_max and m_max are required if generating_vector is a ndarray.
        If generating_vector is an string (path), d_max and m_max can be taken from the␣
↪file name if None
```

```
help(lat.gen_samples)
```

```
Help on method gen_samples in module qmcpy.discrete_distribution.lattice.lattice:

gen_samples(n=None, n_min=0, n_max=8, warn=True, return_unrandomized=False) method of␣
↪qmcpy.discrete_distribution.lattice.lattice.Lattice instance
    Generate lattice samples

    Args:
        n (int): if n is supplied, generate from n_min=0 to n_max=n samples.
            Otherwise use the n_min and n_max explicitly supplied as the following 2␣
↪arguments
        n_min (int): Starting index of sequence.
        n_max (int): Final index of sequence.
        return_unrandomized (bool): return samples without randomization as 2nd return␣
↪value.
            Will not be returned if randomize=False.

    Returns:
        ndarray: (n_max-n_min) x d (dimension) array of samples

    Note:
        Lattice generates in blocks from 2**m to 2**(m+1) so generating
        n_min=3 to n_max=9 requires necessarily produces samples from n_min=2 to n_max=16
        and automatically subsets. May be inefficient for non-powers-of-2 samples sizes.
```

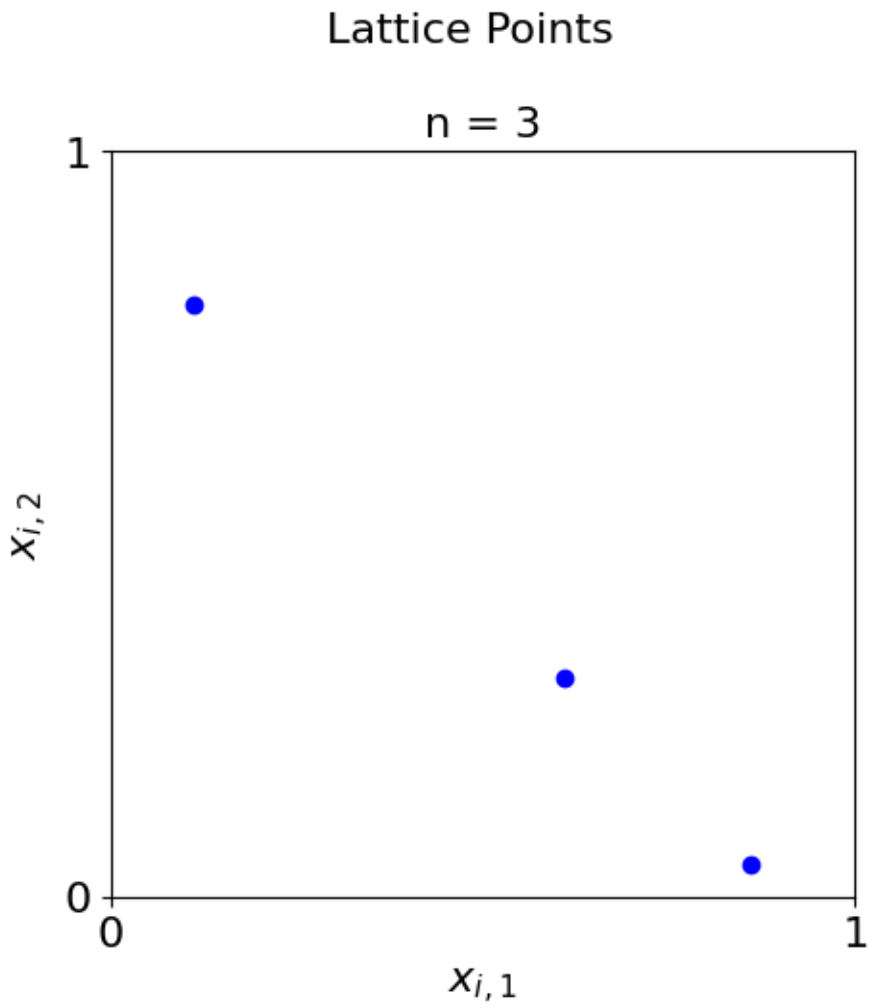
Driver code

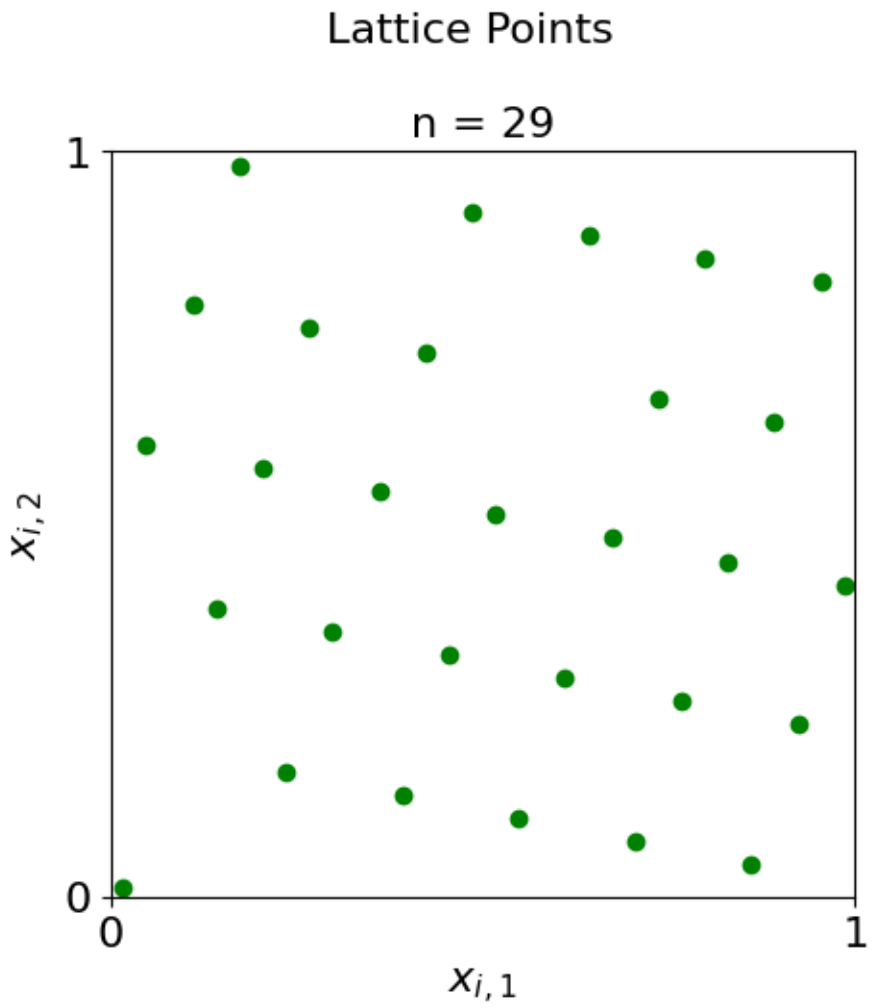
```
lat = qp.Lattice(dimension = 2,randomize= True, generating_vector=21, seed = 120)
print("Basic information of the lattice:")
print(lat)

print("\nA sample lattice generated by the random generating vector: ")

n = 16 #number of points in the sample
print(lat.gen_samples(n))
```

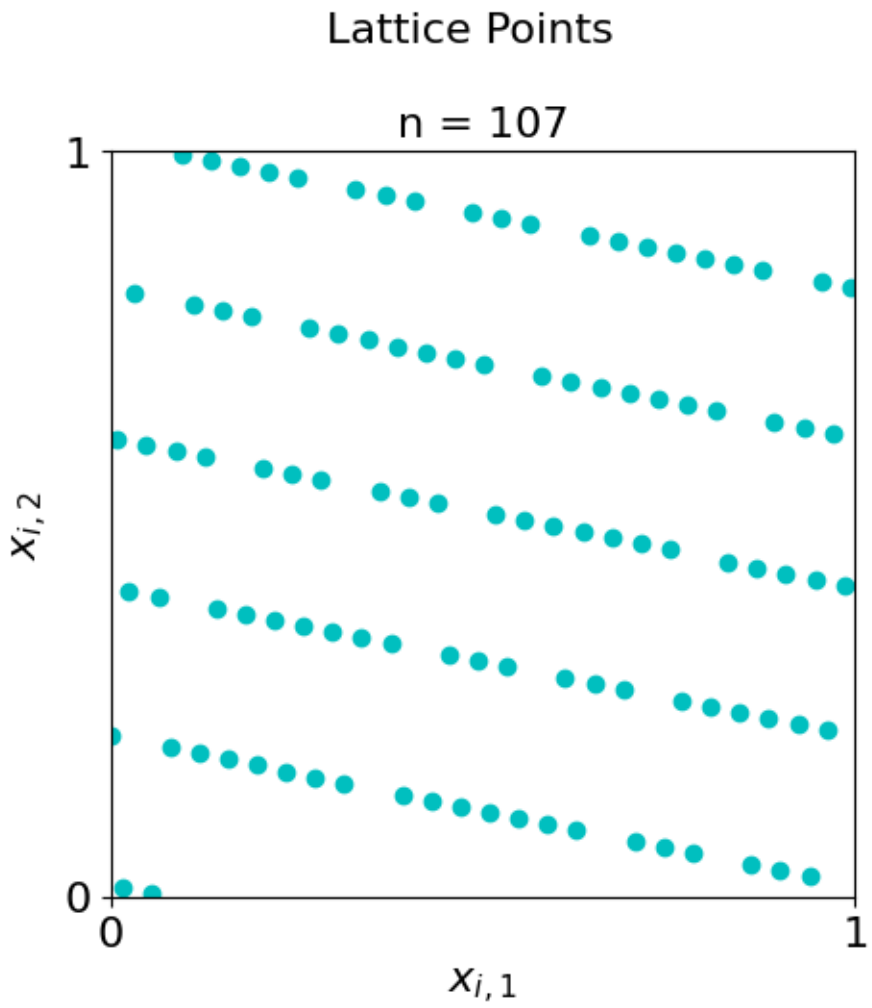
```
Basic information of the lattice:
Lattice (DiscreteDistribution Object)
    d               2^(1)
    dvec            [0 1]
    randomize       1
    order           natural
    gen_vec         [     1 249531]
    entropy         120
    spawn_key       ()

A sample lattice generated by the random generating vector:
[[0.34548142 0.46736834]
 [0.84548142 0.96736834]
 [0.59548142 0.21736834]
 [0.09548142 0.71736834]
 [0.47048142 0.84236834]
 [0.97048142 0.34236834]
 [0.72048142 0.59236834]
 [0.22048142 0.09236834]
 [0.40798142 0.15486834]
 [0.90798142 0.65486834]
 [0.65798142 0.90486834]
 [0.15798142 0.40486834]
 [0.53298142 0.52986834]
 [0.03298142 0.02986834]
 [0.78298142 0.27986834]
 [0.28298142 0.77986834]]
```

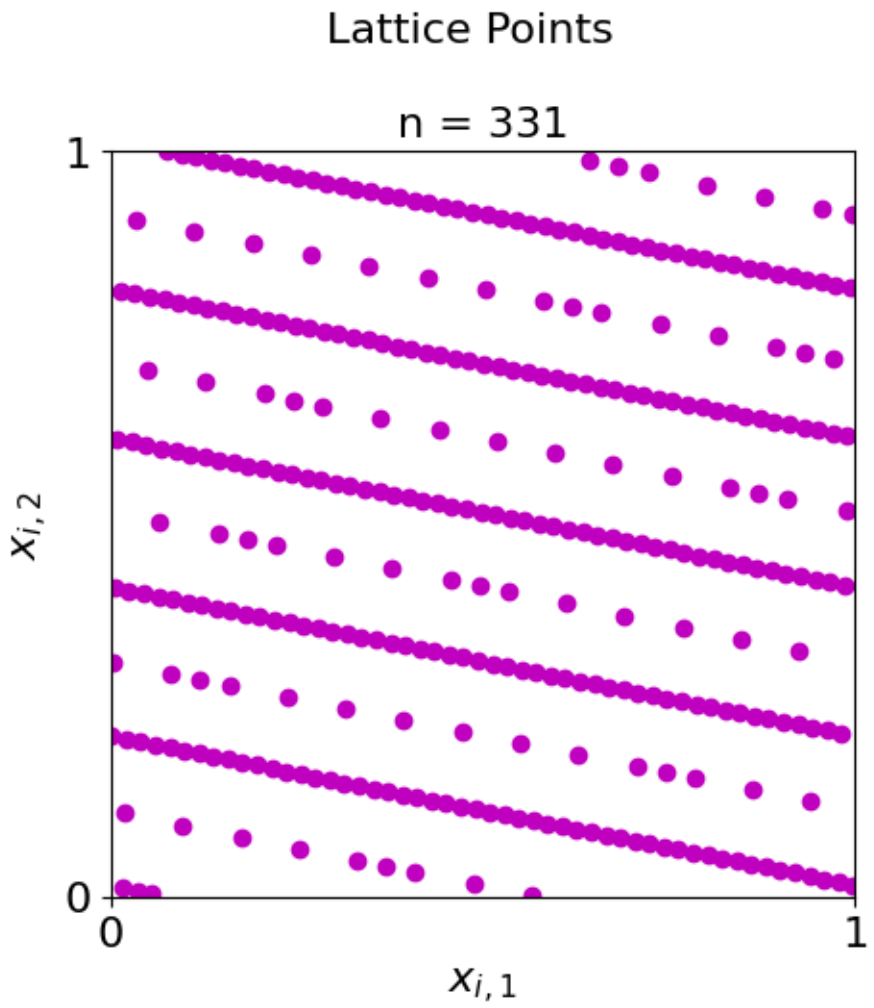
Plots of lattices

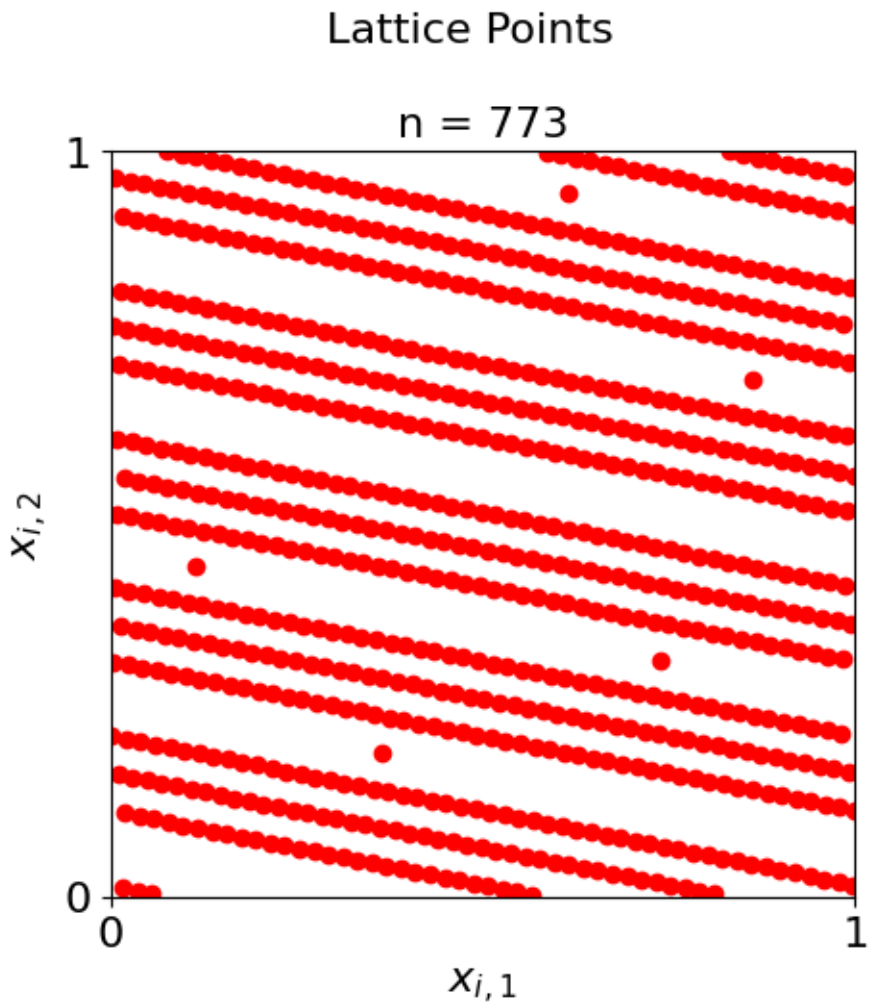
```
#Here the sample sizes are prime numbers
lat = qp.Lattice(dimension=2,generating_vector= 16,seed = 136)

primes = [3,29,107,331,773]

for i in range(0,5):
    plot_successive_points(distrib = lat,ld_name = "Lattice",first_n=primes[i],pt_clr=
↪"bgcmr"[i])
```


Lattice Points
n = 3
1
0
0
1
$x_{i,2}$
$x_{i,1}$


Lattice Points
n = 29
1
0
0
1
$x_{i,2}$
$x_{i,1}$


Lattice Points
n = 107
1
0
0
1
$x_{i,2}$
$x_{i,1}$


Lattice Points
n = 331
1
0
0
1
$x_{i,2}$
$x_{i,1}$



5.23.2 Integration

Runtime comparison bewteen radnom generator and hard-conded generator

```
import warnings
warnings.simplefilter('ignore')

d = 5  #coded as parameters so that
tol = 1E-3 #you can change here and propagate them through this example

data_random = qp.CubQMCLatticeG(qp.Keister(qp.Gaussian(qp.Lattice(d,generating_vector =␣
↪26), mean = 0, covariance = 1/2)), abs_tol = tol).integrate()[1]
data_default = qp.CubQMCLatticeG(qp.Keister(qp.Gaussian(qp.Lattice(d), mean = 0,␣
↪covariance = 1/2)),  abs_tol = tol).integrate()[1]
print("Integration data from a random lattice generator:")
print(data_random)
print("\nIntegration data from the default lattice generator:")
print(data_default)
```

```
Integration data from a random lattice generator:
LDTransformData (AccumulateData Object)
    solution        1.135
    comb_bound_low  1.134
    comb_bound_high 1.135
    comb_flags      1
    n_total         2^(17)
    n               2^(17)
    time_integrate  0.637
CubQMCLatticeG (StoppingCriterion Object)
    abs_tol         0.001
    rel_tol         0
    n_init          2^(10)
    n_max           2^(35)
Keister (Integrand Object)
Gaussian (TrueMeasure Object)
    mean            0
    covariance      2^(-1)
    decomp_type     PCA
    transform       Gaussian (TrueMeasure Object)
                       mean            0
                       covariance      2^(-1)
                       decomp_type     PCA
Lattice (DiscreteDistribution Object)
    d               5
    dvec            [0 1 2 3 4]
    randomize       1
    order           natural
    gen_vec         [       1 10247129 59899927 37429227  4270157]
    entropy         179627174552261816434188931064794162705
    spawn_key       ()

Integration data from the default lattice generator:
LDTransformData (AccumulateData Object)
    solution        1.136
    comb_bound_low  1.136
    comb_bound_high 1.137
    comb_flags      1
    n_total         2^(17)
    n               2^(17)
    time_integrate  0.631
CubQMCLatticeG (StoppingCriterion Object)
    abs_tol         0.001
    rel_tol         0
    n_init          2^(10)
    n_max           2^(35)
Keister (Integrand Object)
Gaussian (TrueMeasure Object)
    mean            0
    covariance      2^(-1)
    decomp_type     PCA
    transform       Gaussian (TrueMeasure Object)
                       mean            0
```

(continues on next page)

(continued from previous page)

```
                    covariance      2^(-1)
                    decomp_type     PCA
Lattice (DiscreteDistribution Object)
    d               5
    dvec            [0 1 2 3 4]
    randomize       1
    order           natural
    gen_vec         [     1 182667 469891 498753 110745]
    entropy         286106574132041199018132344291732219476
    spawn_key       ()
```

Mean vs. Median as a function of the sample size plot

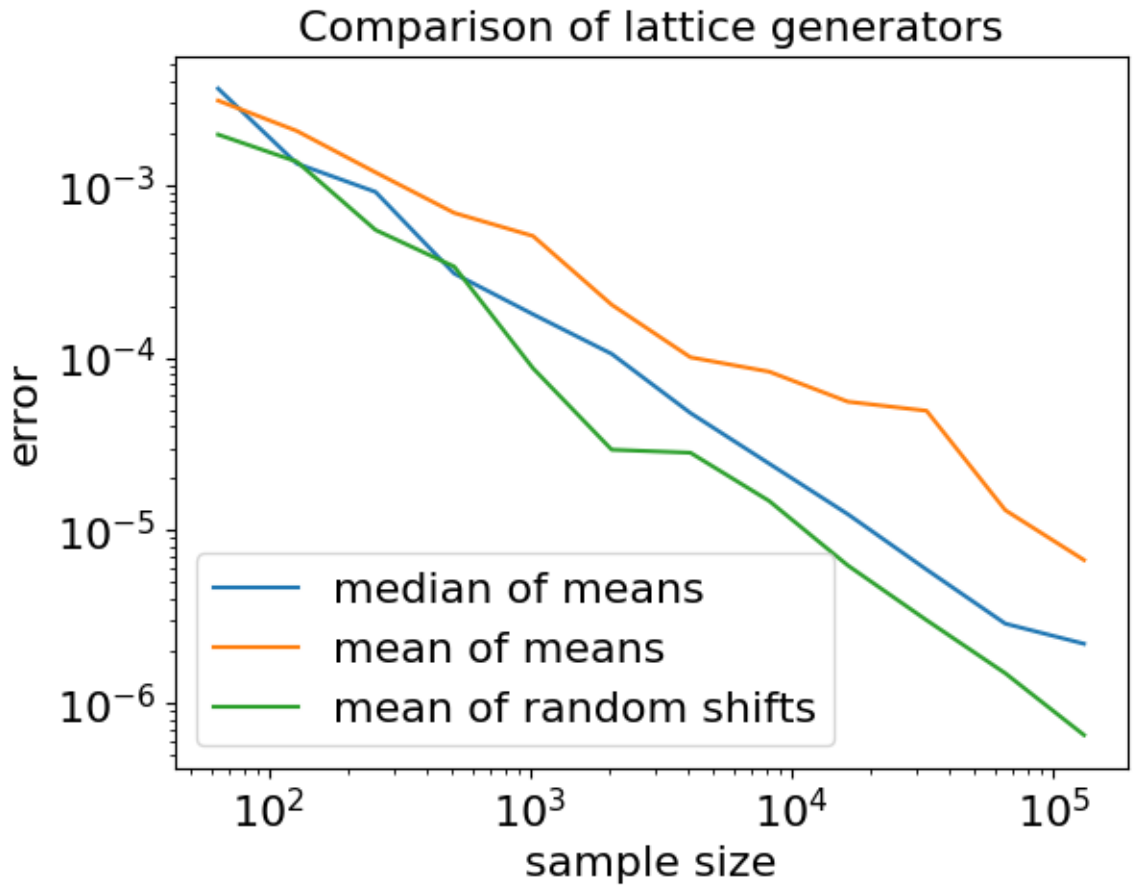
```
#mean vs. median plot
import qmcpy as qp
import numpy as np
import matplotlib.pyplot as plt



d = 2
N_min = 6
N_max = 18
N_list = 2**np.arange(N_min,N_max)
r = 11
num_trials = 25


error_median = np.zeros(N_max - N_min)
error_mean = np.zeros(N_max - N_min)
error_mean_onegen = np.zeros(N_max - N_min)
for i in range(num_trials):
    y_median = []
    y_mean = []
    y_mean_one_gen = []
    print(i)
    list_of_keister_objects_random = []
    list_of_keister_objects_default = []
    y_randomized_list = []
    y_default_list = []
    for k in range(r):
        lattice = qp.Lattice(generating_vector = 26,dimension=d)
        keister = qp.Keister(lattice)
        list_of_keister_objects_random.append(keister)
        x = keister.discrete_distrib.gen_samples(N_list.max())
        y = keister.f(x)
        y_randomized_list.append(y)
        keister = qp.Keister(qp.Lattice(d))
        list_of_keister_objects_default.append(keister)
        x = keister.discrete_distrib.gen_samples(N_list.max())
```

(continues on next page)

(continued from previous page)

```
        y = keister.f(x)
        y_default_list.append(y)

    for N in N_list:

        y_median.append(np.median([np.mean(y[:N]) for y in y_randomized_list]))
        y_mean_one_gen.append(np.mean([np.mean(y[:N]) for y in y_default_list]))
        y_mean.append(np.mean([np.mean(y[:N]) for y in y_randomized_list]))

    answer = keister.exact_integ(d)
    error_median += abs(answer-y_median)
    error_mean += abs(answer-y_mean)
    error_mean_onegen += abs(answer-y_mean_one_gen)

error_median /= num_trials
error_mean /= num_trials
error_mean_onegen /= num_trials

plt.loglog(N_list,error_median,label = "median of means")
plt.loglog(N_list,error_mean,label = "mean of means")
plt.loglog(N_list,error_mean_onegen,label = "mean of random shifts")
plt.xlabel("sample size")
plt.ylabel("error")
plt.title("Comparison of lattice generators")
plt.legend()
plt.savefig("./meanvsmedian.png")
```

```
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
```



5.24 Challenges in Developing Great QMC Software

Computations and Figures for the MCQMC 2022 Article: *Challenges in Developing Great Quasi-Monte Carlo Software*

5.24.1 Import the necessary packages and set up plotting routines

```
import matplotlib.pyplot as plt
import numpy as np
import qmcpy as qp
import time  #timing routines
import warnings  #to suppress warnings when needed
import pickle  #write output to a file and load it back in
from copy import deepcopy

plt.rc('font', size=16)  #set defaults so that the plots are readable
plt.rc('axes', titlesize=16)
plt.rc('axes', labelsize=16)
plt.rc('xtick', labelsize=16)
plt.rc('ytick', labelsize=16)
plt.rc('legend', fontsize=16)
```

(continues on next page)

(continued from previous page)

```
plt.rc('figure', titlesize=16)

#a helpful plotting method to show increasing numbers of points
def plot_successive_points(distrib,ld_name,first_n=64,n_cols=1,
                           pt_clr=['tab:blue', 'tab:green', 'k', 'tab:cyan', 'tab:purple
↪', 'tab:orange'],
                           xlim=[0,1],ylim=[0,1]):
  fig,ax = plt.subplots(nrows=1,ncols=n_cols,figsize=(5*n_cols,5.5))
  if n_cols==1: ax = [ax]
  last_n = first_n*(2**n_cols)
  points = distrib.gen_samples(n=last_n)
  for i in range(n_cols):
    n = first_n
    nstart = 0
    for j in range(i+1):
      n = first_n*(2**j)
      ax[i].scatter(points[nstart:n,0],points[nstart:n,1],color=pt_clr[j])
      nstart = n
    ax[i].set_title('n = %d'%n)
    ax[i].set_xlim(xlim); ax[i].set_xticks(xlim); ax[i].set_xlabel('$x_{i,1}$')
    ax[i].set_ylim(ylim); ax[i].set_yticks(ylim); ax[i].set_ylabel('$x_{i,2}$')
    ax[i].set_aspect((xlim[1]-xlim[0])/(ylim[1]-ylim[0]))
  fig.suptitle('%s Points'%ld_name, y=0.9)
  return fig

print('QMCPy Version',qp.__version__)
```

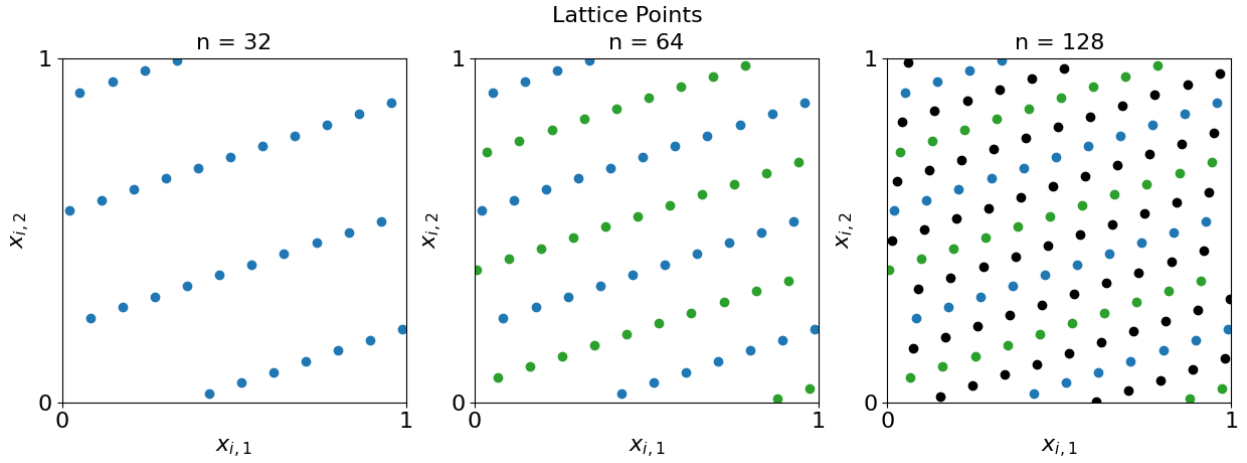
```
QMCPy Version 1.3.2
```

Make sure that you have the relevant path to store the figures

```
figpath = '' #this path sends the figures to the desired directory
```

5.24.2 Here are some plots of Low Discrepancy (LD) Lattice Points

```
d = 5 #dimension
n = 32 #number of points
cols = 3 #number of columns
ld = qp.Lattice(d) #define the generator
fig = plot_successive_points(ld,'Lattice',first_n=n,n_cols=cols)
fig.savefig(figpath+'latticepts.eps',format='eps',bbox_inches='tight')
```



5.24.3 Beam Example Plots

Plot the time and sample size required to solve for the deflection of the whole beam using low discrepancy with and without parallel

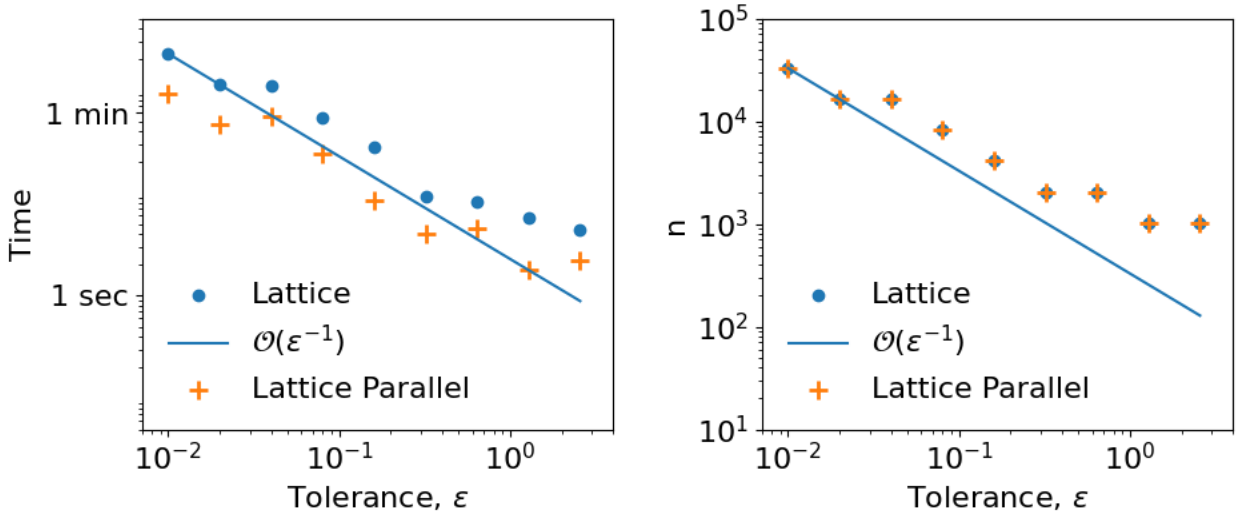
```
with open(figpath+'ld_parallel.pkl','rb') as myfile: tol_vec,n_tol,ld_time,ld_n,ld_p_
↪time,ld_p_n,best_solution = pickle.load(myfile)
print(best_solution)
fig,ax = plt.subplots(nrows=1,ncols=2,figsize=(11,4))
ax[0].scatter(tol_vec[0:n_tol],ld_time[0:n_tol],color='tab:blue');
ax[0].plot(tol_vec[0:n_tol],[(ld_time[0]*tol_vec[0])/tol_vec[jj] for jj in range(n_tol)],
↪color='tab:blue')
ax[0].scatter(tol_vec[0:n_tol],ld_p_time[0:n_tol],color='tab:orange',marker = '+',s=100,
↪linewidths=2);
#ax[0].plot(tol_vec[0:n_tol],[(ld_p_time[0]*tol_vec[0])/tol_vec[jj] for jj in range(n_
↪tol)],color='tab:orange')
ax[0].set_ylim([0.05,500]); ax[0].set_ylabel('Time')
ax[1].scatter(tol_vec[0:n_tol],ld_n[0:n_tol],color='tab:blue');
ax[1].plot(tol_vec[0:n_tol],[(ld_n[0]*tol_vec[0])/tol_vec[jj] for jj in range(n_tol)],
↪color='tab:blue')
ax[1].scatter(tol_vec[0:n_tol],ld_p_n[0:n_tol],color='tab:orange',marker = '+',s=100,
↪linewidths=2);
#ax[1].plot(tol_vec[0:n_tol],[(ld_p_n[0]*tol_vec[0])/tol_vec[jj] for jj in range(n_tol)],
↪color='tab:orange')
ax[1].set_ylim([10,1e5]); ax[1].set_ylabel('n')
for ii in range(2):
  ax[ii].set_xlim([0.007,4]); ax[ii].set_xlabel('Tolerance, '+r'$\varepsilon$')
  ax[ii].set_xscale('log'); ax[ii].set_yscale('log')
  ax[ii].legend(['Lattice',r'$\mathcal{O}(\varepsilon^{-1})$','Lattice Parallel',r'$\
↪mathcal{O}(\varepsilon^{-1})$'],frameon=False)
  ax[ii].set_aspect(0.6)
ax[0].set_yticks([1, 60], labels = ['1 sec', '1 min'])
fig.savefig(figpath+'ldparallelbeam.eps',format='eps',bbox_inches='tight')
```

```
[   0.            4.09650336   15.63089718   33.92196588   58.36259037
   88.42166624  123.63753779  163.62564457  208.07398743  256.74371736
```

(continues on next page)

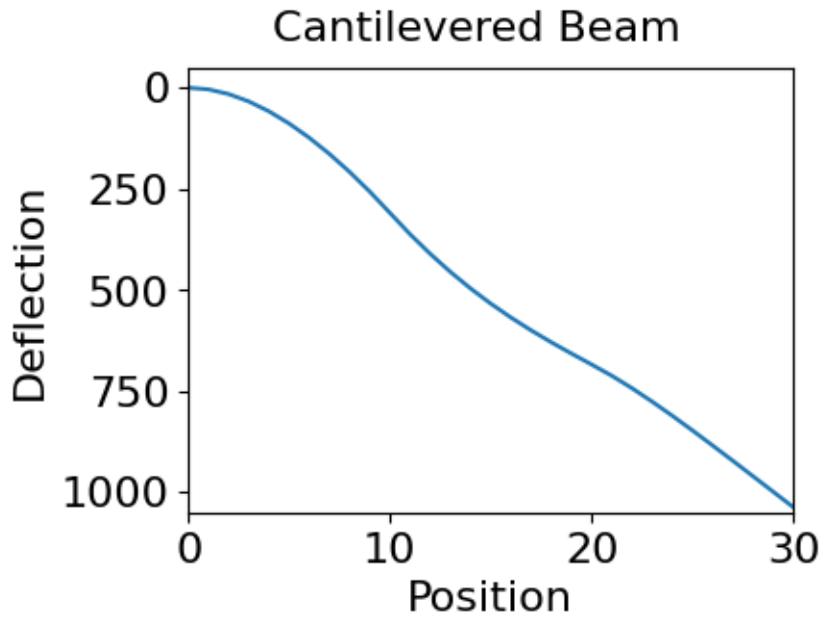
(continued from previous page)

```
  309.46852588  362.17337763  410.80019812  455.47543786  496.40155276
  533.85794874  568.20201672  599.87051305  629.38039738  657.32920342
  684.39496877  712.15829288  742.93687638  776.09146089  811.06031655
  847.35856113  884.57752606  922.38415394  960.52038491  998.8025606
 1037.12106673]
```



Plot of beam solution

```
fig,ax = plt.subplots(figsize=(6,3))
ax.plot(best_solution,'-')
ax.set_xlim([0,len(best_solution)-1]); ax.set_xlabel('Position')
ax.set_ylim([1050,-50]);  ax.set_ylabel('Deflection');
ax.set_aspect(0.02)
fig.suptitle('Cantilevered Beam')
fig.savefig(figpath+'cantileveredbeamwords.eps',format='eps',bbox_inches='tight')
```




```
qp.util.stop_notebook()
```

```
Type 'yes' to continue running notebookno
```

```
An exception has occurred, use %tb to see the full traceback.


SystemExit: Pausing notebook execution
```

Below is long-running code, that we rarely wish to run

5.24.4 Beam Example Computations

To run this, you need to be running the `docker` application, https://www.docker.com/products/docker-desktop/

Set up the problem using a docker container to solve the ODE

```
import umbridge #this is the connector
!docker run --name muqbp -d -it -p 4243:4243 linusseelinger/benchmark-muq-beam-
↪propagation:latest #get beam example
d = 3 #dimension of the randomness
lb = 1 #lower bound on randomness
ub = 1.2 #upper bound on randomness
umbridge_config = {"d": d}
model = umbridge.HTTPModel('http://localhost:4243','forward') #this is the original model
outindex = -1 #choose last element of the vector of beam deflections
modeli = deepcopy(model) #and construct a model for just that deflection
modeli.get_output_sizes = lambda *args : [1]
modeli.get_output_sizes()
modeli.__call__ = lambda *args,**kwargs: [[model.__call__(*args,**kwargs)[0][outindex]]]
```

```
docker: Error response from daemon: Conflict. The container name "/muqbp" is already in␣
↪use by container "7f9e0237bd3e72783743efb67f78ce8cc800f5a24835f4191bc423f960cdedac".␣
↪You have to remove (or rename) that container to be able to reuse that name.
See 'docker run --help'.
```

First we compute the time required to solve for the deflection of the end point using IID and low discrepancy

```
ld = qp.Uniform(qp.Lattice(d,seed=7),lower_bound=lb,upper_bound=ub) #lattice points for␣
↪this problem
ld_integ = qp.UMBridgeWrapper(ld,modeli,umbridge_config,parallel=False) #integrand
iid = qp.Uniform(qp.IIDStdUniform(d),lower_bound=lb,upper_bound=ub) #iid points for this␣
↪problem
iid_integ = qp.UMBridgeWrapper(iid,modeli,umbridge_config,parallel=False) #integrand
tol = 0.01  #smallest tolerance

n_tol = 14  #number of different tolerances
ii_iid = 9  #make this larger to reduce the time required by not running all cases for␣
↪IID
tol_vec = [tol*(2**ii) for ii in range(n_tol)]  #initialize vector of tolerances
ld_time = [0]*n_tol; ld_n = [0]*n_tol  #low discrepancy time and number of function␣
↪values
iid_time = [0]*n_tol; iid_n = [0]*n_tol  #IID time and number of function values
print(f'\nCantilever Beam\n')
print('iteration ', end = '')
for ii in range(n_tol):
  solution, data = qp.CubQMCLatticeG(ld_integ, abs_tol = tol_vec[ii]).integrate()
  if ii == 0:
    best_solution_i = solution
  ld_time[ii] = data.time_integrate
  ld_n[ii] = data.n_total
  if ii >= ii_iid:
    solution, data = qp.CubMCG(iid_integ, abs_tol = tol_vec[ii]).integrate()
    iid_time[ii] = data.time_integrate
    iid_n[ii] = data.n_total
  print(ii, end = ' ')
with open(figpath+'iid_ld.pkl','wb') as myfile:pickle.dump([tol_vec,n_tol,ii_iid,ld_time,
↪ld_n,iid_time,iid_n,best_solution_i],myfile)
```

```
Cantilever Beam

iteration 0 1 2 3 4 5 6 7 8 9 10 11 12 13
```

Next, we compute the time required to solve for the deflection of the whole beam using low discrepancy with and without parallel

```
ld_integ = qp.UMBridgeWrapper(ld,model,umbridge_config,parallel=False) #integrand
ld_integ_p = qp.UMBridgeWrapper(ld,model,umbridge_config,parallel=8) #integrand with␣
↪parallel processing

tol = 0.01
n_tol = 9  #number of different tolerances
tol_vec = [tol*(2**ii) for ii in range(n_tol)]  #initialize vector of tolerances
ld_time = [0]*n_tol; ld_n = [0]*n_tol  #low discrepancy time and number of function␣
↪values
ld_p_time = [0]*n_tol; ld_p_n = [0]*n_tol  #low discrepancy time and number of function␣
↪values with parallel
print(f'\nCantilever Beam\n')
print('iteration ', end = '')
for ii in range(n_tol):
  solution, data = qp.CubQMCLatticeG(ld_integ, abs_tol = tol_vec[ii]).integrate()
  if ii == 0:
    best_solution = solution
  ld_time[ii] = data.time_integrate
  ld_n[ii] = data.n_total
  solution, data = qp.CubQMCLatticeG(ld_integ_p, abs_tol = tol_vec[ii]).integrate()
  ld_p_time[ii] = data.time_integrate
  ld_p_n[ii] = data.n_total
  print(ii, end = ' ')
with open(figpath+'ld_parallel.pkl','wb') as myfile:pickle.dump([tol_vec,n_tol,ld_time,
↪ld_n,ld_p_time,ld_p_n,best_solution],myfile)
```

```
Cantilever Beam

iteration 0 1 2 3 4 5 6 7 8
```

```
!docker rm -f muqbp #shut down docker image
```

5.25 Purdue University Colloquim Talk

Computations and Figures for Department of Statistics Colloquim at Purdue University

presented on Friday, March 3, 2023, slides here

5.25.1 Import the necessary packages and set up plotting routines

```
import matplotlib.pyplot as plt
import numpy as np
import qmcpy as qp
import time  #timing routines
import warnings  #to suppress warnings when needed
import pickle  #write output to a file and load it back in
from copy import deepcopy

plt.rc('font', size=16)  #set defaults so that the plots are readable
plt.rc('axes', titlesize=16)
plt.rc('axes', labelsize=16)
plt.rc('xtick', labelsize=16)
plt.rc('ytick', labelsize=16)
plt.rc('legend', fontsize=16)
plt.rc('figure', titlesize=16)

#a helpful plotting method to show increasing numbers of points
def plot_successive_points(distrib,ld_name,first_n=64,n_cols=1,
                           pt_clr=['tab:blue', 'tab:green', 'k', 'tab:cyan', 'tab:purple
↪', 'tab:orange'],
                           xlim=[0,1],ylim=[0,1]):
  fig,ax = plt.subplots(nrows=1,ncols=n_cols,figsize=(5*n_cols,5.5))
  if n_cols==1: ax = [ax]
  last_n = first_n*(2**n_cols)
  points = distrib.gen_samples(n=last_n)
  for i in range(n_cols):
    n = first_n
    nstart = 0
    for j in range(i+1):
      n = first_n*(2**j)
      ax[i].scatter(points[nstart:n,0],points[nstart:n,1],color=pt_clr[j])
      nstart = n
    ax[i].set_title('n = %d'%n)
    ax[i].set_xlim(xlim); ax[i].set_xticks(xlim); ax[i].set_xlabel('$x_{i,1}$')
    ax[i].set_ylim(ylim); ax[i].set_yticks(ylim); ax[i].set_ylabel('$x_{i,2}$')
    ax[i].set_aspect((xlim[1]-xlim[0])/(ylim[1]-ylim[0]))
  fig.suptitle('%s Points'%ld_name, y=0.87)
  return fig

print('QMCPy Version',qp.__version__)
```

```
QMCPy Version 1.3.2
```

Set the path to save the figures here

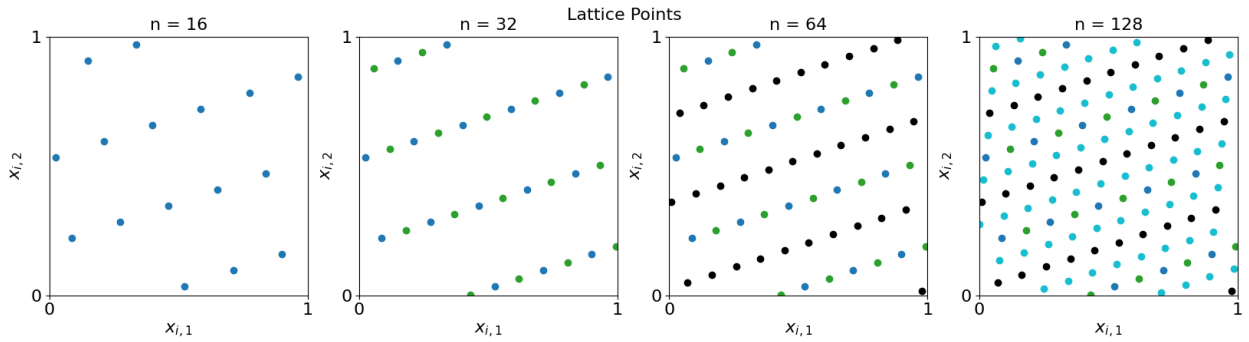
```
figpath = '' #this path sends the figures to the directory that you want
```

5.25.2 Here are some plots of IID and Low Discrepancy (LD) Points

Lattice points first

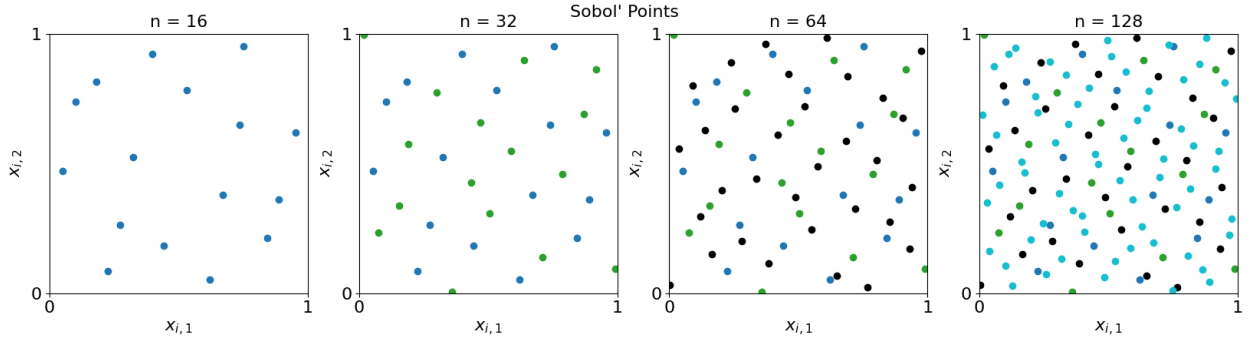
```
d = 5 #dimension
n = 16 #number of points
ld = qp.Lattice(d) #define the generator
xpts = ld.gen_samples(n) #generate points
print(xpts)
fig = plot_successive_points(ld,'Lattice',first_n=n,n_cols=4)
fig.savefig(figpath+'latticepts.eps',format='eps')
```

```
[[3.99318970e-01 6.57874778e-01 8.98029521e-01 3.75576673e-01
  8.76944929e-01]
 [8.99318970e-01 1.57874778e-01 3.98029521e-01 8.75576673e-01
  3.76944929e-01]
 [6.49318970e-01 4.07874778e-01 6.48029521e-01 6.25576673e-01
  1.26944929e-01]
 [1.49318970e-01 9.07874778e-01 1.48029521e-01 1.25576673e-01
  6.26944929e-01]
 [5.24318970e-01 3.28747777e-02 2.73029521e-01 5.00576673e-01
  1.94492924e-03]
 [2.43189699e-02 5.32874778e-01 7.73029521e-01 5.76672905e-04
  5.01944929e-01]
 [7.74318970e-01 7.82874778e-01 2.30295212e-02 7.50576673e-01
  2.51944929e-01]
 [2.74318970e-01 2.82874778e-01 5.23029521e-01 2.50576673e-01
  7.51944929e-01]
 [4.61818970e-01 3.45374778e-01 8.55295212e-02 4.38076673e-01
  4.39444929e-01]
 [9.61818970e-01 8.45374778e-01 5.85529521e-01 9.38076673e-01
  9.39444929e-01]
 [7.11818970e-01 9.53747777e-02 8.35529521e-01 6.88076673e-01
  6.89444929e-01]
 [2.11818970e-01 5.95374778e-01 3.35529521e-01 1.88076673e-01
  1.89444929e-01]
 [5.86818970e-01 7.20374778e-01 4.60529521e-01 5.63076673e-01
  5.64444929e-01]
 [8.68189699e-02 2.20374778e-01 9.60529521e-01 6.30766729e-02
  6.44449292e-02]
 [8.36818970e-01 4.70374778e-01 2.10529521e-01 8.13076673e-01
  8.14444929e-01]
 [3.36818970e-01 9.70374778e-01 7.10529521e-01 3.13076673e-01
  3.14444929e-01]]
```



Next Sobol' points

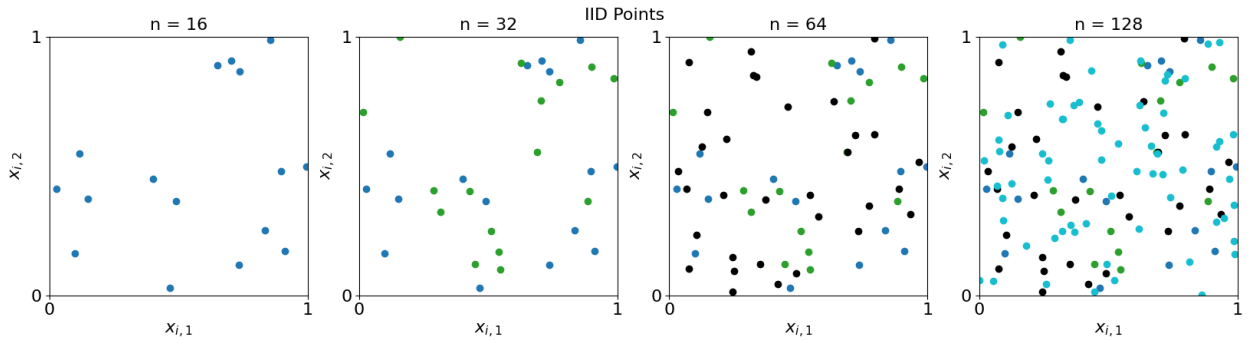
```
ld = qp.Sobol(d) #define the generator
xpts_Sobol = ld.gen_samples(n) #generate points
fig = plot_successive_points(ld,'Sobol\'',first_n=n,n_cols=4)
fig.savefig(figpath+'sobolpts.eps',format='eps')
```



Compare to IID

Note that there are more gaps and clusters

```
iid = qp.IIDStdUniform(d) #define the generator
xpts = ld.gen_samples(n) #generate points
xpts
fig = plot_successive_points(iid,'IID',first_n=n,n_cols=4)
fig.savefig(figpath+'iidpts.eps',format='eps')
```



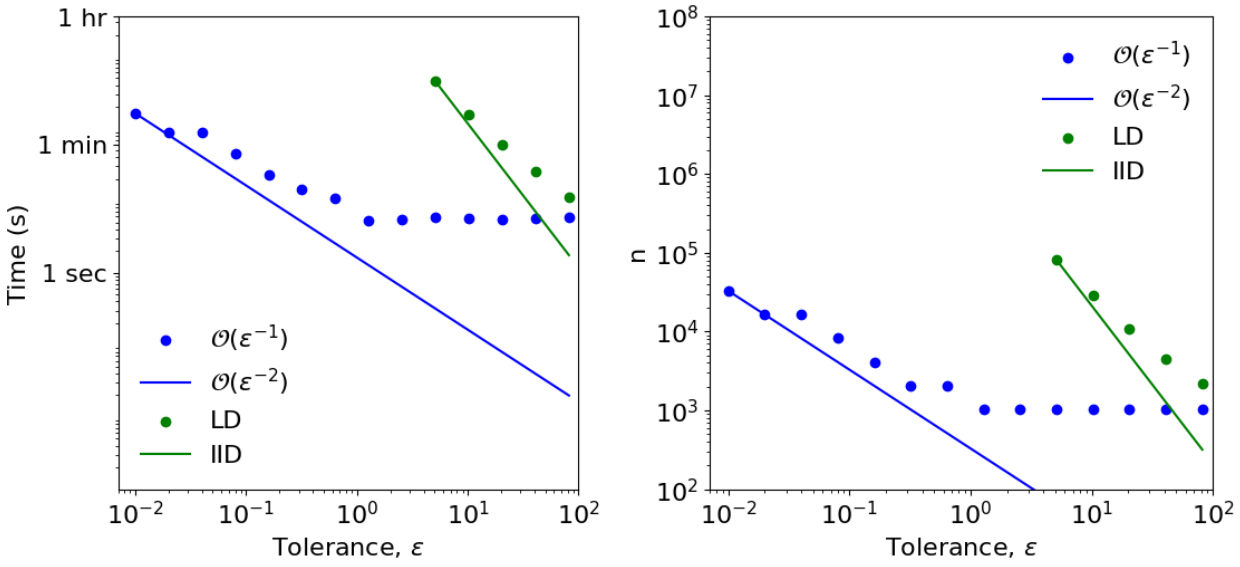
5.25.3 Beam Example Figures

Using computations done below

Plot the time and sample size required to solve for the deflection of the end point using IID and low discrepancy

```
with open(figpath+'iid_ld.pkl','rb') as myfile: tol_vec,n_tol,ii_iid,ld_time,ld_n,iid_
↪time,iid_n,best_solution_i = pickle.load(myfile)
print(best_solution_i)
fig,ax = plt.subplots(nrows=1,ncols=2,figsize=(13,5.5))
ax[0].scatter(tol_vec[0:n_tol],ld_time[0:n_tol],color='b');
ax[0].plot(tol_vec[0:n_tol],[(ld_time[0]*tol_vec[0])/tol_vec[jj] for jj in range(n_tol)],
↪color='b')
ax[0].scatter(tol_vec[ii_iid:n_tol],iid_time[ii_iid:n_tol],color='g');
ax[0].plot(tol_vec[ii_iid:n_tol],[(iid_time[ii_iid]*(tol_vec[ii_iid]**2))/(tol_
↪vec[jj]**2) for jj in range(ii_iid,n_tol)],color='g')
ax[0].set_ylim([0.001,1000]); ax[0].set_ylabel('Time (s)')
ax[1].scatter(tol_vec[0:n_tol],ld_n[0:n_tol],color='b');
ax[1].plot(tol_vec[0:n_tol],[(ld_n[0]*tol_vec[0])/tol_vec[jj] for jj in range(n_tol)],
↪color='b')
ax[1].scatter(tol_vec[ii_iid:n_tol],iid_n[ii_iid:n_tol],color='g');
ax[1].plot(tol_vec[ii_iid:n_tol],[(iid_n[ii_iid]*(tol_vec[ii_iid]**2))/(tol_vec[jj]**2)␣
↪for jj in range(ii_iid,n_tol)],color='g')
ax[1].set_ylim([1e2,1e8]); ax[1].set_ylabel('n')
for ii in range(2):
  ax[ii].set_xlim([0.007,100]); ax[ii].set_xlabel('Tolerance, '+r'$\varepsilon$')
  ax[ii].set_xscale('log'); ax[ii].set_yscale('log')
  ax[ii].legend([r'$\mathcal{O}(\varepsilon^{-1})$',r'$\mathcal{O}(\varepsilon^{-2})$',
↪'LD','IID'],frameon=False)
  ax[ii].set_aspect(0.65)
ax[0].set_yticks([1, 60, 3600], labels = ['1 sec', '1 min', '1 hr'])
fig.savefig(figpath+'iidldbeam.eps',format='eps',bbox_inches='tight')
```

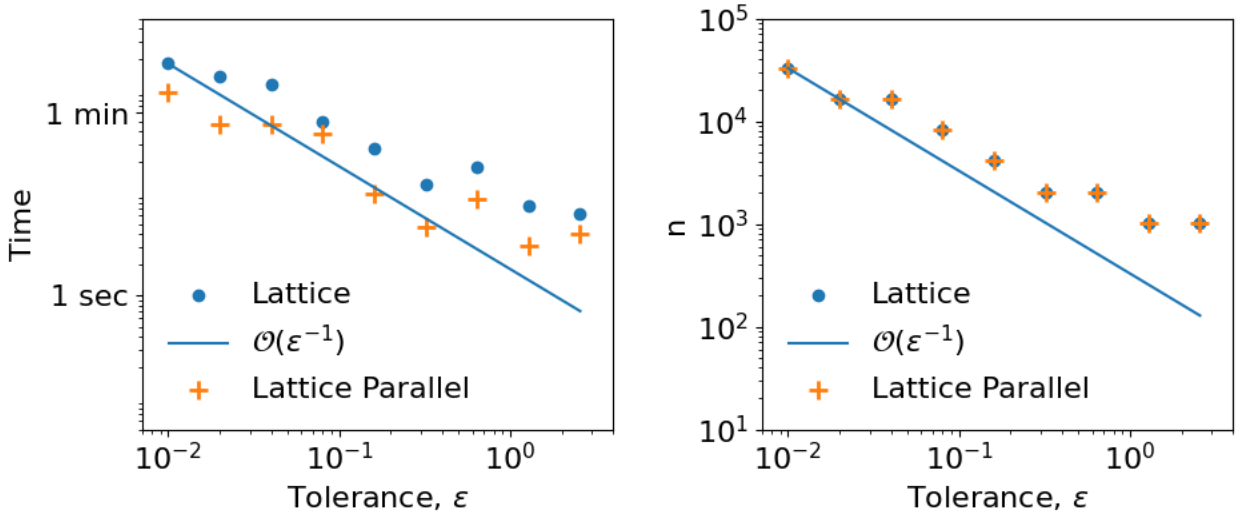
```
[1037.12106673]
```



Plot the time and sample size required to solve for the deflection of the whole beam using low discrepancy with and without parallel

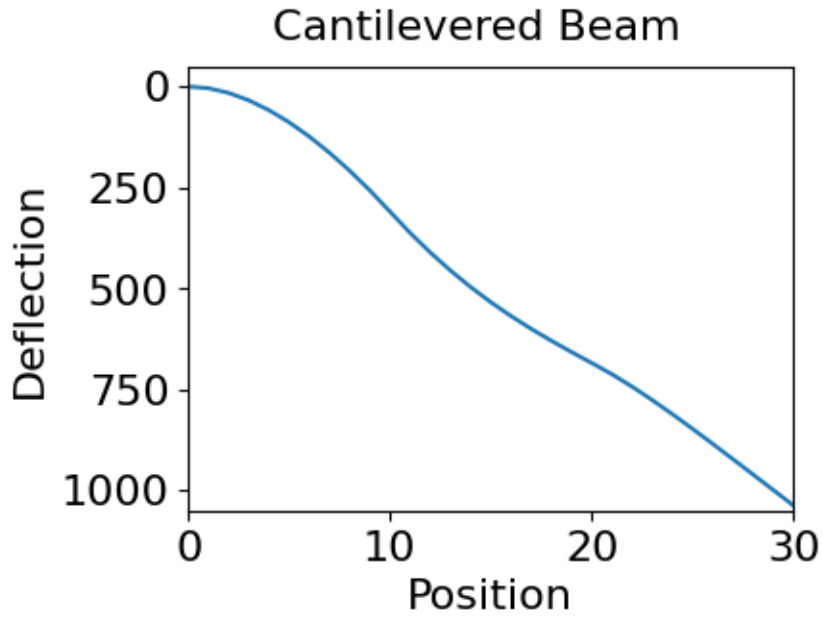
```
with open(figpath+'ld_parallel.pkl','rb') as myfile: tol_vec,n_tol,ld_time,ld_n,ld_p_
↪time,ld_p_n,best_solution = pickle.load(myfile)
print(best_solution_i)
fig,ax = plt.subplots(nrows=1,ncols=2,figsize=(11,4))
ax[0].scatter(tol_vec[0:n_tol],ld_time[0:n_tol],color='tab:blue');
ax[0].plot(tol_vec[0:n_tol],[(ld_time[0]*tol_vec[0])/tol_vec[jj] for jj in range(n_tol)],
↪color='tab:blue')
ax[0].scatter(tol_vec[0:n_tol],ld_p_time[0:n_tol],color='tab:orange',marker = '+',s=100,
↪linewidths=2);
#ax[0].plot(tol_vec[0:n_tol],[(ld_p_time[0]*tol_vec[0])/tol_vec[jj] for jj in range(n_
↪tol)],color='tab:orange')
ax[0].set_ylim([0.05,500]); ax[0].set_ylabel('Time')
ax[1].scatter(tol_vec[0:n_tol],ld_n[0:n_tol],color='tab:blue');
ax[1].plot(tol_vec[0:n_tol],[(ld_n[0]*tol_vec[0])/tol_vec[jj] for jj in range(n_tol)],
↪color='tab:blue')
ax[1].scatter(tol_vec[0:n_tol],ld_p_n[0:n_tol],color='tab:orange',marker = '+',s=100,
↪linewidths=2);
#ax[1].plot(tol_vec[0:n_tol],[(ld_p_n[0]*tol_vec[0])/tol_vec[jj] for jj in range(n_tol)],
↪color='tab:orange')
ax[1].set_ylim([10,1e5]); ax[1].set_ylabel('n')
for ii in range(2):
  ax[ii].set_xlim([0.007,4]); ax[ii].set_xlabel('Tolerance, '+r'$\varepsilon$')
  ax[ii].set_xscale('log'); ax[ii].set_yscale('log')
  ax[ii].legend(['Lattice',r'$\mathcal{O}(\varepsilon^{-1})$','Lattice Parallel',r'$\
↪mathcal{O}(\varepsilon^{-1})$'],frameon=False)
  ax[ii].set_aspect(0.6)
ax[0].set_yticks([1, 60], labels = ['1 sec', '1 min'])
fig.savefig(figpath+'ldparallelbeam.eps',format='eps',bbox_inches='tight')
```

```
[1037.12106673]
```



Plot of beam solution

```
fig,ax = plt.subplots(figsize=(6,3))
ax.plot(best_solution,'-')
ax.set_xlim([0,len(best_solution)-1]); ax.set_xlabel('Position')
ax.set_ylim([1050,-50]);  ax.set_ylabel('Mean Deflection');
ax.set_aspect(0.02)
fig.suptitle('Cantilevered Beam')
fig.savefig(figpath+'cantileveredbeamwords.eps',format='eps',bbox_inches='tight')
```



```
qp.util.stop_notebook()
```

```
Type 'yes' to continue running notebookyes
```

Below is long-running code, that we rarely wish to run

5.25.4 Beam Example Computations

Set up the problem using a docker container to solve the ODE

To run this, you need to be running the docker application, https://www.docker.com/products/docker-desktop/

```
import umbridge #this is the connector
!docker run --name muqbp -d -it -p 4243:4243 linusseelinger/benchmark-muq-beam-
↪propagation:latest #get beam example
d = 3 #dimension of the randomness
lb = 1 #lower bound on randomness
ub = 1.2 #upper bound on randomness
umbridge_config = {"d": d}
model = umbridge.HTTPModel('http://localhost:4243','forward') #this is the original model
outindex = -1 #choose last element of the vector of beam deflections
modeli = deepcopy(model) #and construct a model for just that deflection
modeli.get_output_sizes = lambda *args : [1]
modeli.get_output_sizes()
modeli.__call__ = lambda *args,**kwargs: [[model.__call__(*args,**kwargs)[0][outindex]]]
```

```
docker: Error response from daemon: Conflict. The container name "/muqbp" is already in␣
↪use by container "7f9e0237bd3e72783743efb67f78ce8cc800f5a24835f4191bc423f960cdedac".␣
↪You have to remove (or rename) that container to be able to reuse that name.
See 'docker run --help'.
```

First we compute the time required to solve for the deflection of the end point using IID and low discrepancy

```
ld = qp.Uniform(qp.Lattice(d,seed=7),lower_bound=lb,upper_bound=ub) #lattice points for␣
↪this problem
ld_integ = qp.UMBridgeWrapper(ld,modeli,umbridge_config,parallel=False) #integrand
iid = qp.Uniform(qp.IIDStdUniform(d),lower_bound=lb,upper_bound=ub) #iid points for this␣
↪problem
iid_integ = qp.UMBridgeWrapper(iid,modeli,umbridge_config,parallel=False) #integrand
tol = 0.01  #smallest tolerance

n_tol = 14  #number of different tolerances
ii_iid = 9  #make this larger to reduce the time required by not running all cases for␣
↪IID
tol_vec = [tol*(2**ii) for ii in range(n_tol)]  #initialize vector of tolerances
ld_time = [0]*n_tol; ld_n = [0]*n_tol  #low discrepancy time and number of function␣
↪values
iid_time = [0]*n_tol; iid_n = [0]*n_tol  #IID time and number of function values
print(f'\nCantilever Beam\n')
```

(continues on next page)

(continued from previous page)

```
print('iteration ', end = '')
for ii in range(n_tol):
  solution, data = qp.CubQMCLatticeG(ld_integ, abs_tol = tol_vec[ii]).integrate()
  if ii == 0:
    best_solution_i = solution
  ld_time[ii] = data.time_integrate
  ld_n[ii] = data.n_total
  if ii >= ii_iid:
    solution, data = qp.CubMCG(iid_integ, abs_tol = tol_vec[ii]).integrate()
    iid_time[ii] = data.time_integrate
    iid_n[ii] = data.n_total
  print(ii, end = ' ')
with open(figpath+'iid_ld.pkl','wb') as myfile:pickle.dump([tol_vec,n_tol,ii_iid,ld_time,
↪ld_n,iid_time,iid_n,best_solution_i],myfile)
```

```
Cantilever Beam

iteration 0 1 2 3 4 5 6 7 8 9 10 11 12 13
```

Next, we compute the time required to solve for the deflection of the whole beam using low discrepancy with and without parallel

```
ld_integ = qp.UMBridgeWrapper(ld,model,umbridge_config,parallel=False) #integrand
ld_integ_p = qp.UMBridgeWrapper(ld,model,umbridge_config,parallel=8) #integrand with␣
↪parallel processing

tol = 0.01
n_tol = 9  #number of different tolerances
tol_vec = [tol*(2**ii) for ii in range(n_tol)]  #initialize vector of tolerances
ld_time = [0]*n_tol; ld_n = [0]*n_tol  #low discrepancy time and number of function␣
↪values
ld_p_time = [0]*n_tol; ld_p_n = [0]*n_tol  #low discrepancy time and number of function␣
↪values with parallel
print(f'\nCantilever Beam\n')
print('iteration ', end = '')
for ii in range(n_tol):
  solution, data = qp.CubQMCLatticeG(ld_integ, abs_tol = tol_vec[ii]).integrate()
  if ii == 0:
    best_solution = solution
  ld_time[ii] = data.time_integrate
  ld_n[ii] = data.n_total
  solution, data = qp.CubQMCLatticeG(ld_integ_p, abs_tol = tol_vec[ii]).integrate()
  ld_p_time[ii] = data.time_integrate
  ld_p_n[ii] = data.n_total
  print(ii, end = ' ')
with open(figpath+'ld_parallel.pkl','wb') as myfile:pickle.dump([tol_vec,n_tol,ld_time,
↪ld_n,ld_p_time,ld_p_n,best_solution],myfile)
```

```
Cantilever Beam

```

(continues on next page)

(continued from previous page)

```
iteration 0 1 2 3 4 5 6 7 8
```

Shut down docker

```
!docker rm -f muqbp #shut down docker image
```

5.26 Genz Function in Dakota and QMCPy

A QMCPy implementation and comparison of Dakota's Genz function

```
from numpy import *
from qmcpy import *
import pandas as pd
from matplotlib import pyplot
import tempfile
import os
import subprocess
import numpy as np
pyplot.style.use('../qmcpy/qmcpy.mplstyle')
%matplotlib inline
```

```
kinds_func = ['oscilatory','corner-peak']
kinds_coeff = [1,2,3]
ds = 2**arange(8)
ns = 2**arange(7,19)
ds
```

```
array([  1,   2,   4,   8,  16,  32,  64, 128])
```

```
ref_sols = {}
print('logging: ',end='',flush=True)
x_full = DigitalNetB2(ds.max(),seed=7).gen_samples(2**22)
for kind_func in kinds_func:
    for kind_coeff in kinds_coeff:
        tag = '%s.%d'%(kind_func,kind_coeff)
        print('%s, '%tag,end='',flush=True)
        mu_hats = zeros(len(ds),dtype=float)
        for j,d in enumerate(ds):
            genz = Genz(IIDStdUniform(d),kind_func=kind_func,kind_coeff=kind_coeff)
            y = genz.f(x_full[:,:d])
            mu_hats[j] = y.mean()
        ref_sols[tag] = mu_hats
print()
ref_sols = pd.DataFrame(ref_sols)
ref_sols['d'] = ds
ref_sols.set_index('d',inplace=True)
ref_sols
```

```
logging: oscilatory.1, oscilatory.2, oscilatory.3, corner-peak.1, corner-peak.2, corner-
↪peak.3,
```

```
# with tempfile.TemporaryDirectory() as tmp:
#     with open(os.path.join(tmp, "dakota.in"), "w") as io:
#         io.write(f"environment\
#             \ttabular_data\n\
#             method\
#             \tfsu_quasi_mc halton\
#             \t\tsamples = {ns.max()}\
#             \toutput silent\n\
#             variables\
#             \tcontinuous_design = {ds.max()}\
#             \tlower_bounds = {' '.join(['0.0' for _ in range(ds.max())])}\
#             \tupper_bounds = {' '.join(['1.0' for _ in range(ds.max())])}\n\
#             interface\
#             \tfork\
#             \t\tanalysis_driver = 'dummy'\
#             \tbatch\
#             \twork_directory named 'work'\n\
#             responses\
#             \tobjective_functions = 1\
#             \tno_gradients\
#             \tno_hessians"
#         )
#     subprocess.run(["dakota", "dakota.in"], cwd=tmp, stdout=subprocess.DEVNULL,␣
↪stderr=subprocess.DEVNULL)
#     file = os.listdir(os.path.join(tmp, "work"))[0]
#     with open(os.path.join(tmp, "work", file), "r") as io:
#         lines = io.readlines()
#         x_full_dakota = []
#         for n, line in enumerate(lines):
#             if f"{ds.max()} variables" in line:
#                 x_full_dakota.append([float(lines[n + 1 + j].split()[0]) for j in␣
↪range(ds.max())])
#         x_full_dakota = np.vstack(x_full_dakota)
x_full_dakota = np.loadtxt("x_full_dakota.txt")
```

```
n_max,d_max = ns.max(),ds.max()
pts = {
    'IID Standard Uniform': IIDStdUniform(d_max).gen_samples(n_max),
    'Lattice (random shift)': Lattice(d_max).gen_samples(n_max),
    'Digital Net (random scramble + shift)': DigitalNetB2(d_max).gen_samples(n_max),
    'Halton (not random, not general)': Halton(d_max,randomize=False,generalize=False).
↪gen_samples(n_max,warn=False),
    'Halton (not random, general)': Halton(d_max,randomize=False,generalize=True).gen_
↪samples(n_max,warn=False),
    'Halton (random, not general)': Halton(d_max,randomize=True,generalize=False).gen_
↪samples(n_max),
    'Halton (random, general)': Halton(d_max,randomize=True,generalize=True).gen_
↪samples(n_max),
    'Halton (Dakota)': x_full_dakota[:n_max,:d_max]
```

(continues on next page)

(continued from previous page)

```
}
```

```
nrows = len(ds)
ncols = len(kinds_func)*len(kinds_coeff)
print('logging')
fig,ax = pyplot.subplots(nrows=nrows,ncols=ncols,figsize=(ncols*5,nrows*5),sharey=True,
↪sharex=True)
ax = ax.reshape(nrows,ncols)
colors = pyplot.rcParams['axes.prop_cycle'].by_key()['color'] + ["indigo"]
for v,(name,x_full) in enumerate(pts.items()):
    print('%20s d: '%name,end='',flush=True)
    for j,d in enumerate(ds):
        print('%d, '%d,end='',flush=True)
        for i1,kind_func in enumerate(kinds_func):
            for i2,kind_coeff in enumerate(kinds_coeff):
                i = len(kinds_coeff)*i1+i2
                tag = '%s.%d'%(kind_func,kind_coeff)
                genz = Genz(IIDStdUniform(d),kind_func=kind_func,kind_coeff=kind_coeff)
                y_full = genz.f(x_full[:,:d])
                mu_hats = array([y_full[:n].mean() for n in ns],dtype=float)
                error = abs(mu_hats-ref_sols.loc[d,tag])
                ax[j,i].plot(ns,error,label=name, color=colors[v])
                if v==(len(pts)-1): ax[j,i].legend(loc='lower left')
                if v>0: continue
                ax[j,i].set_xscale('log',base=2)
                ax[j,i].set_yscale('log',base=10)
                if i==0: ax[j,i].set_ylabel(r'$d=%d$\\$\varepsilon = \lvert \mu - \hat{\
↪mu} \rvert$'%d)
                if j==0: ax[j,i].set_title(tag)
                if j==(len(ds)-1):
                    ax[j,i].set_xlabel(r'$n$')
                    ax[j,i].set_xticks(ns)
                    ax[j,i].set_xlim([ns.min(),ns.max()])
    print()
```

```
logging
IID Standard Uniform d: 1, 2, 4, 8, 16, 32, 64, 128,
Lattice (random shift) d: 1, 2, 4, 8, 16, 32, 64, 128,
Digital Net (random scramble + shift) d: 1, 2, 4, 8, 16, 32, 64, 128,
Halton (not random, not general) d: 1, 2, 4, 8, 16, 32, 64, 128,
Halton (not random, general) d: 1, 2, 4, 8, 16, 32, 64, 128,
Halton (random, not general) d: 1, 2, 4, 8, 16, 32, 64, 128,
Halton (random, general) d: 1, 2, 4, 8, 16, 32, 64, 128,
     Halton (Dakota) d: 1, 2, 4, 8, 16, 32, 64, 128,
```


oscilatory.1
oscilatory.2
oscilatory.3
corner-peak.1
corner-peak.2
corner-peak.3
IID Standard Uniform
Lattice (random shift)
Digital Net (random scramble + shift)
Halton (not random, not general)
Halton (not random, general)
Halton (random, not general)
Halton (random, general)
Halton (Dakota)
n

5.27 Monte Carlo for Vector Functions of Integrals

Demo Accompanying Aleksei Sorokin's PyData Chicago 2023 Talk

5.27.1 Monte Carlo Problem

$$\text{True Mean} = \mu = \mathbb{E}[g(T)] = \mathbb{E}[f(X)] = \int_{[0,1]^d} f(x)\mathrm{d}x \approx \frac{1}{n}\sum_{i=0}^{n-1} f(X_i) = \hat{\mu} = \text{Sample Mean}$$

- T, original measure on $\mathcal{T}$
- $g : \mathcal{T} \to \mathbb{R}$, original integrand
- $X \sim \mathcal{U}[0,1]^d$, transformed measure
- $f : [0,1]^d \to \mathbb{R}$, transformed integrand

5.27.2 Python Setup

```
import qmcpy as qp
import numpy as np
import scipy.stats
import pandas as pd
import time
from matplotlib import pyplot
pyplot.style.use('../qmcpy/qmcpy.mplstyle')
colors = pyplot.rcParams['axes.prop_cycle'].by_key()['color']
%matplotlib inline
```

5.27.3 Discrete Distribution

Generate sampling locations $X_0, \dots, X_{n-1} \sim \mathcal{U}[0,1]^d$

Independent Identically Distributed (IID) Points for Crude Monte Carlo (CMC)

```
iid = qp.IIDStdUniform(dimension=3)
iid.gen_samples(n=4)
```

```
array([[0.33212887, 0.77514762, 0.32281907],
       [0.63606431, 0.87304412, 0.160779  ],
       [0.86648199, 0.16583253, 0.28605698],
       [0.33916281, 0.40836749, 0.59704801]])
```

```
iid.gen_samples(4)
```

```
array([[0.52964844, 0.93287387, 0.92878954],
       [0.52281122, 0.70201421, 0.23376703],
       [0.92408974, 0.69777308, 0.15770565],
       [0.33577149, 0.68206595, 0.97291222]])
```

```
iid
```

```
IIDStdUniform (DiscreteDistribution Object)
    d               3
    entropy         260382129008356013626800297715809294905
    spawn_key       ()
```

Low Discrepancy (LD) Points for Quasi-Monte Carlo (QMC)

```
ld_lattice = qp.Lattice(3)
ld_lattice.gen_samples(4)
```

```
array([[0.53421508, 0.29262606, 0.39547365],
       [0.03421508, 0.79262606, 0.89547365],
       [0.78421508, 0.04262606, 0.14547365],
       [0.28421508, 0.54262606, 0.64547365]])
```

```
ld_lattice.gen_samples(4)
```

```
array([[0.53421508, 0.29262606, 0.39547365],
       [0.03421508, 0.79262606, 0.89547365],
       [0.78421508, 0.04262606, 0.14547365],
       [0.28421508, 0.54262606, 0.64547365]])
```

```
ld_lattice.gen_samples(n_min=2,n_max=4)
```

```
array([[0.78421508, 0.04262606, 0.14547365],
       [0.28421508, 0.54262606, 0.64547365]])
```

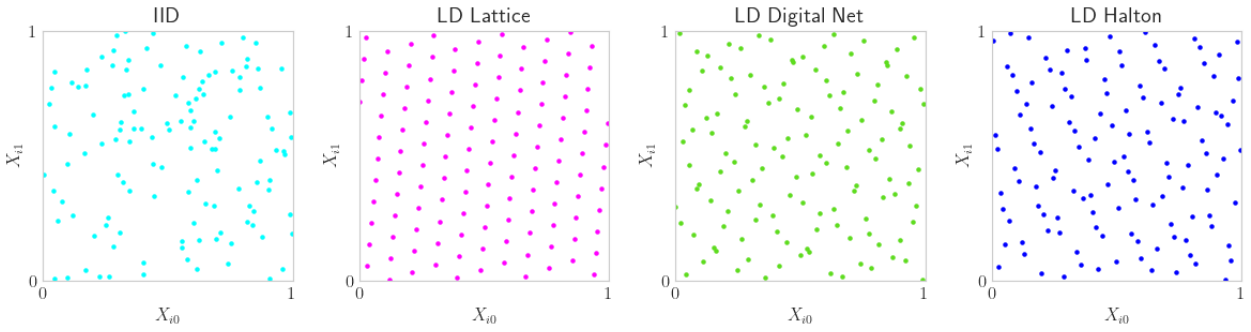
```
ld_lattice
```

```
Lattice (DiscreteDistribution Object)
    d               3
    dvec            [0 1 2]
    randomize       1
    order           natural
    gen_vec         [     1 182667 469891]
    entropy         258997323036294594641435801210344083177
    spawn_key       ()
```

Visuals

IID vs LD Points

```
n = 2**7 # Lattice and Digital Net prefer powers of 2 sample sizes
discrete_distribs = {
    'IID': qp.IIDStdUniform(2),
    'LD Lattice': qp.Lattice(2),
    'LD Digital Net': qp.DigitalNetB2(2),
    'LD Halton': qp.Halton(2)}
fig,ax = pyplot.subplots(nrows=1,ncols=len(discrete_distribs),figsize=(3*len(discrete_
↪distribs),3))
ax = np.atleast_1d(ax)
for i,(name,discrete_distrib) in enumerate(discrete_distribs.items()):
    x = discrete_distrib.gen_samples(n)
    ax[i].scatter(x[:,0],x[:,1],s=5,color=colors[i])
    ax[i].set_title(name)
    ax[i].set_aspect(1)
    ax[i].set_xlabel(r'$X_{i0}$'); ax[i].set_ylabel(r'$X_{i1}$')
    ax[i].set_xlim([0,1]); ax[i].set_ylim([0,1])
    ax[i].set_xticks([0,1]); ax[i].set_yticks([0,1])
```



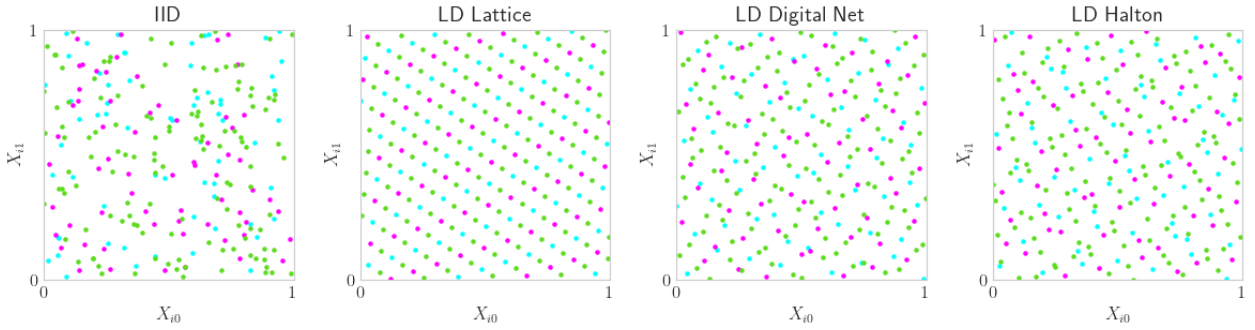
LD Space Filling Extensibility

```
m_min,m_max = 6,8
fig,ax = pyplot.subplots(nrows=1,ncols=len(discrete_distribs),figsize=(3*len(discrete_
↪distribs),3))
ax = np.atleast_1d(ax)
for i,(name,discrete_distrib) in enumerate(discrete_distribs.items()):
    x = discrete_distrib.gen_samples(2**m_max)
    n_min = 0
    for m in range(m_min,m_max+1):
        n_max = 2**m
        ax[i].scatter(x[n_min:n_max,0],x[n_min:n_max,1],s=5,color=colors[m-m_min],label=
↪'n_min = %d, n_max = %d'%(n_min,n_max))
        n_min = 2**m
    ax[i].set_title(name)
    ax[i].set_aspect(1)
    ax[i].set_xlabel(r'$X_{i0}$'); ax[i].set_ylabel(r'$X_{i1}$')
```

(continues on next page)

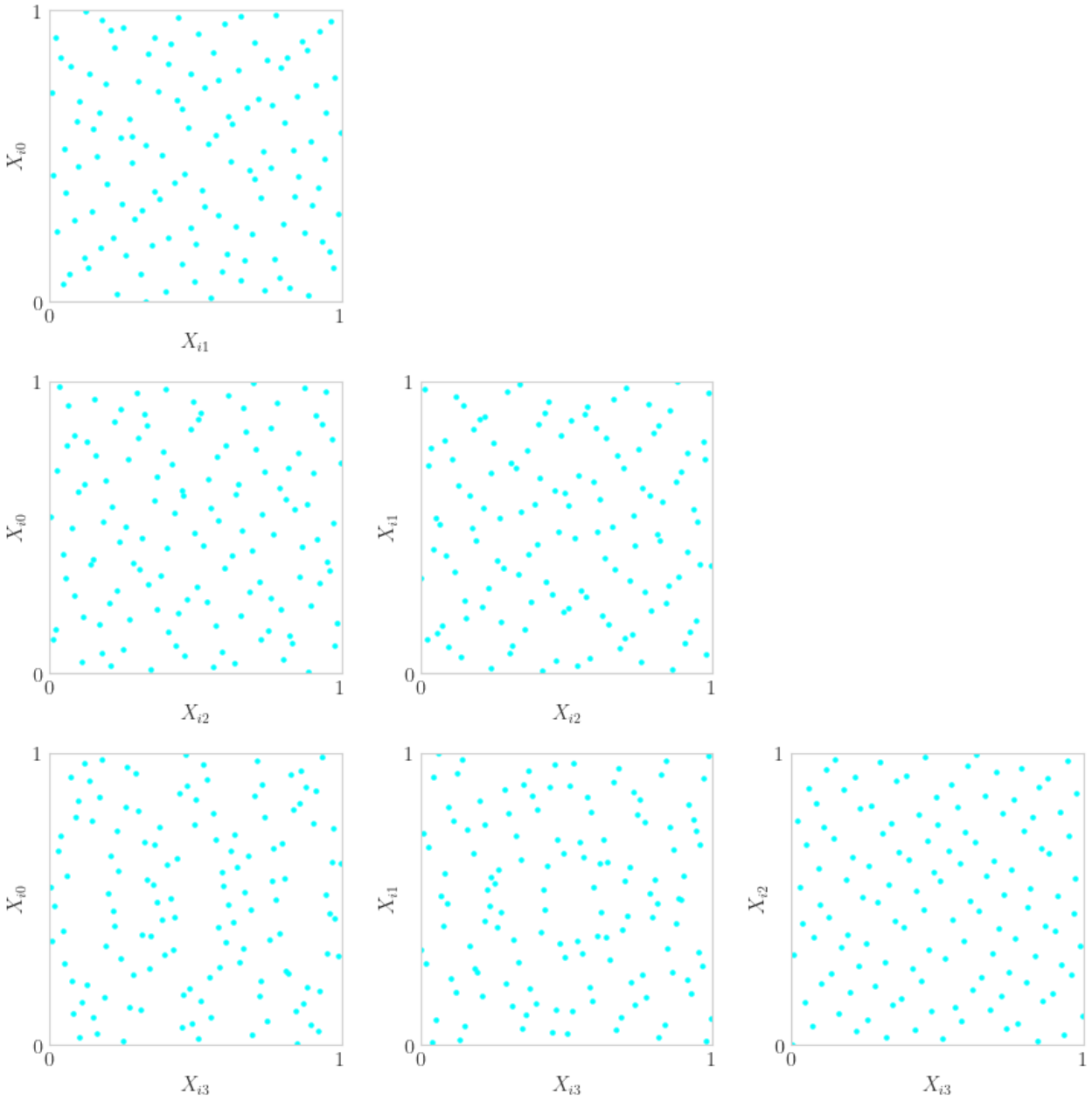
(continued from previous page)

```
    ax[i].set_xlim([0,1]); ax[i].set_ylim([0,1])
    ax[i].set_xticks([0,1]); ax[i].set_yticks([0,1])
```



High Dimensional Pairs Plotting

```
discrete_distrib = qp.DigitalNetB2(4)
x = discrete_distrib(2**7)
d = discrete_distrib.d
assert d>=2
fig,ax = pyplot.subplots(nrows=d,ncols=d,figsize=(3*d,3*d))
for i in range(d):
    fig.delaxes(ax[i,i])
    for j in range(i):
        ax[i,j].scatter(x[:,i],x[:,j],s=5)
        fig.delaxes(ax[j,i])
        ax[i,j].set_aspect(1)
        ax[i,j].set_xlabel(r'$X_{i%d}$'%i); ax[i,j].set_ylabel(r'$X_{i%d}$'%j)
        ax[i,j].set_xlim([0,1]); ax[i,j].set_ylim([0,1])
        ax[i,j].set_xticks([0,1]); ax[i,j].set_yticks([0,1])
```



5.27.4 True Measure

Define T, facilitate transform from original integrand g to transformed integrand f

```
discrete_distrib = qp.Halton(3)
true_measure = qp.Gaussian(discrete_distrib,mean=[1,2,3],covariance=[4,5,6])
true_measure.gen_samples(4)
```

```
array([[ 0.40678465,  1.60773559,  0.34020435],
       [ 5.25849108,  3.60610932,  3.86999995],
       [ 1.42111148, -0.90377725,  2.30796197],
       [-0.8050952 ,  2.23293569,  5.98842354]])
```

```
true_measure.gen_samples(n_min=2,n_max=4)
```

```
array([[ 1.42111148, -0.90377725,  2.30796197],
       [-0.8050952 ,  2.23293569,  5.98842354]])
```

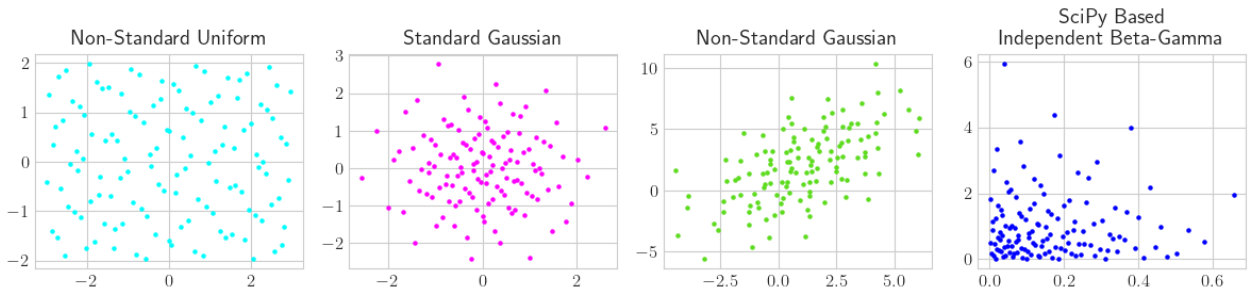
```
true_measure
```

```
Gaussian (TrueMeasure Object)
    mean            [1 2 3]
    covariance      [4 5 6]
    decomp_type     PCA
```

Visuals

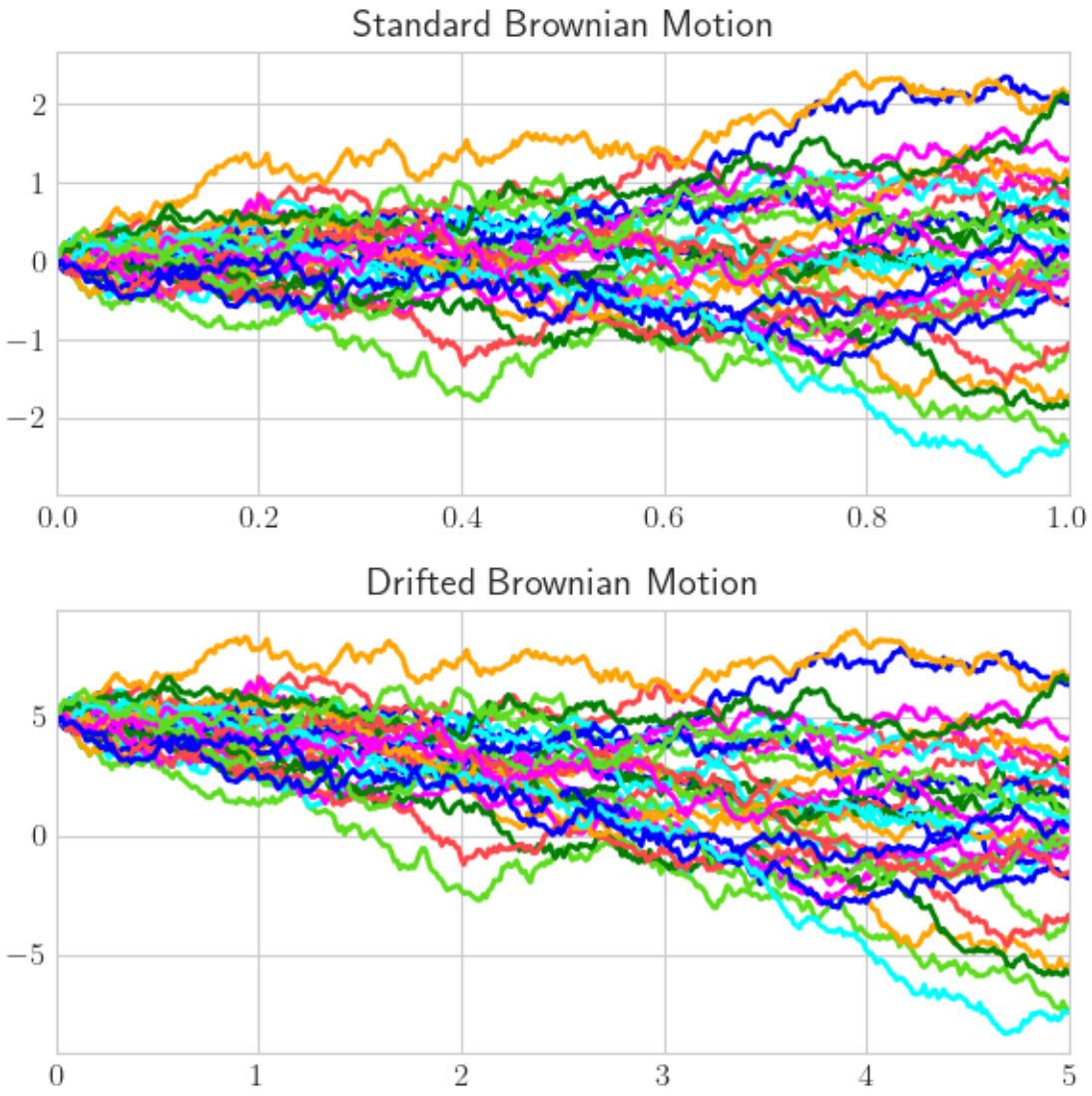
Some True Measure Samplings

```
n = 2**7
discrete_distrib = qp.DigitalNetB2(2)
true_measures = {
    'Non-Standard Uniform': qp.Uniform(discrete_distrib,lower_bound=[-3,-2],upper_
↪bound=[3,2]),
    'Standard Gaussian': qp.Gaussian(discrete_distrib),
    'Non-Standard Gaussian': qp.Gaussian(discrete_distrib,mean=[1,2],covariance=[[5,4],
↪[4,9]]),
    'SciPy Based\nIndependent Beta-Gamma': qp.SciPyWrapper(discrete_distrib,[scipy.stats.
↪beta(a=1,b=5),scipy.stats.gamma(a=1)])}
fig,ax = pyplot.subplots(nrows=1,ncols=len(true_measures),figsize=(3*len(true_measures),
↪3))
ax = np.atleast_1d(ax)
for i,(name,true_measure) in enumerate(true_measures.items()):
    t = true_measure.gen_samples(n)
    ax[i].scatter(t[:,0],t[:,1],s=5,color=colors[i])
    ax[i].set_title(name)
```



Brownian Motion

```
n = 32
discrete_distrib = qp.Lattice(365)
brownian_motions = {
    'Standard Brownian Motion': qp.BrownianMotion(discrete_distrib),
    'Drifted Brownian Motion': qp.BrownianMotion(discrete_distrib,t_final=5,initial_
↪value=5,drift=-1,diffusion=2)}
fig,ax = pyplot.subplots(nrows=len(brownian_motions),ncols=1,figsize=(6,3*len(brownian_
↪motions)))
ax = np.atleast_1d(ax)
for i,(name,brownian_motion) in enumerate(brownian_motions.items()):
    t = brownian_motion.gen_samples(n)
    t_w_init = np.hstack([brownian_motion.initial_value*np.ones((n,1)),t])
    tvec_w_0 = np.hstack([0,brownian_motion.time_vec])
    ax[i].plot(tvec_w_0,t_w_init.T)
    ax[i].set_xlim([tvec_w_0[0],tvec_w_0[-1]])
    ax[i].set_title(name)
```



5.27.5 Integrand

Define original integrand g, store transformed integrand f

Wrap your Function into QMCPy

Our simple example

$$g(T) = T_0 + T_1 + \cdots + T_{d-1}, \qquad T \sim \mathcal{N}(0, I_d)$$

$$f(X) = g(\Phi^{-1}(X)), \qquad \Phi \text{ standard normal CDF}$$

$$\mathbb{E}[f(X)] = \mathbb{E}[g(T)] = 0$$

```
def myfun(t): # define g, the ORIGINAL integrand
    # t an (n,d) shaped np.ndarray of sample from the ORIGINAL (true) measure
    y = t.sum(1)
    return y # an (n,) shaped np.ndarray
true_measure = qp.Gaussian(qp.Halton(5)) # LD Halton discrete distrib for QMC problem
qp_myfun = qp.CustomFun(true_measure,myfun,parallel=False)
```

Evalute the Automatically Transformed Integrand

```
x = qp_myfun.discrete_distrib.gen_samples(4) # samples from the TRANSFORMED measure
y = qp_myfun.f(x) # evaluate the TRANSFORMED integrand at the TRANSFORMED samples
y
```

```
array([[-3.28135198],
       [ 0.58308562],
       [-3.74555828],
       [ 3.35850654]])
```

Manual QMC Approximation

Note that when doing importance sampling the below doesn't work. In that case we need to take a specially weighted sum instead instead of the equally weighted sum as done below.

```
x = qp_myfun.discrete_distrib.gen_samples(2**16) # samples from the TRANSFORMED measure
y = qp_myfun.f(x) # evaluate the TRANSFORMED integrand at the TRANSFORMED samples
mu_hat = y.mean()
mu_hat
```

```
-1.8119973887083317e-05
```

Predefined Integrands

Many more integrands detailed at https://qmcpy.readthedocs.io/en/master/algorithms.html#integrand-class

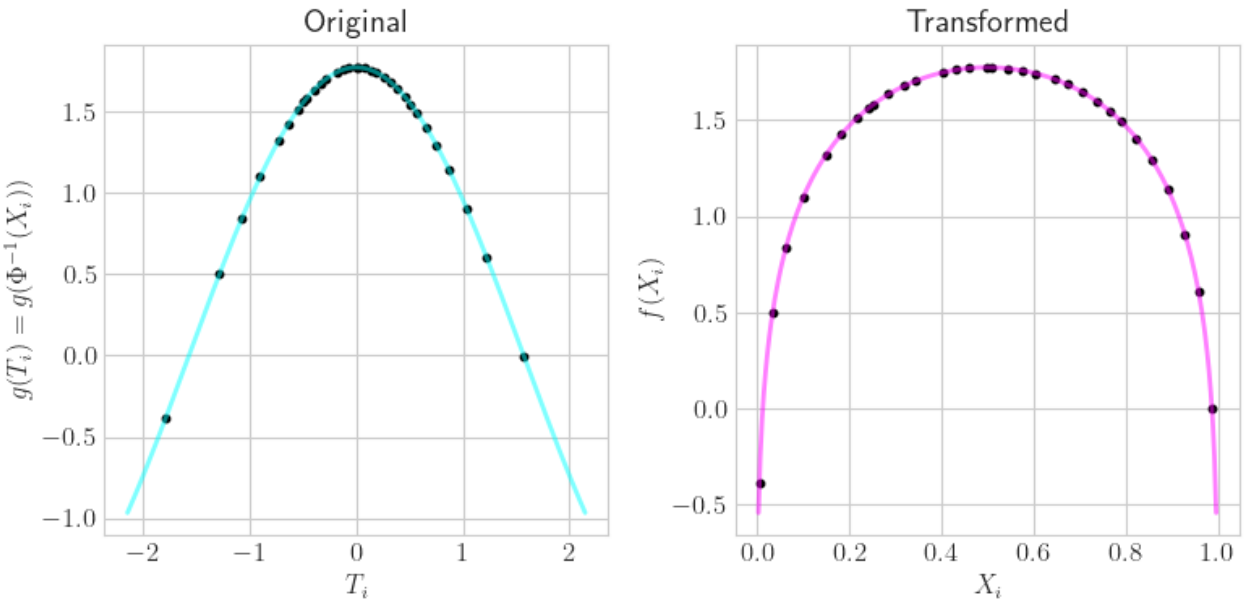
Integrands contain their true measure definition, so the user only needs to pass in a sampler. Samplers are often just discrete distributions.

```
asian_option = qp.AsianOption(
    sampler = qp.DigitalNetB2(52),
    volatility = 1/2,
    start_price = 30,
    strike_price = 35,
    interest_rate = 0.001,
    t_final = 1,
    call_put = 'call',
    mean_type = 'arithmetic')
x = asian_option.discrete_distrib.gen_samples(2**16)
y = asian_option.f(x)
mu_hat = y.mean()
mu_hat
```

```
1.7888072890178057
```

Visual Transformation

```
n = 32
keister = qp.Keister(qp.DigitalNetB2(1))
fig,ax = pyplot.subplots(nrows=1,ncols=2,figsize=(8,4))
x = keister.discrete_distrib.gen_samples(n)
t = keister.true_measure.gen_samples(n)
f_of_x = keister.f(x).squeeze()
g_of_t = keister.g(t).squeeze()
assert (f_of_x==g_of_t).all()
x_fine = np.linspace(0,1,257)[1:-1,None]
f_of_xfine = keister.f(x_fine).squeeze()
lb = 1.2*max(abs(t.min()),abs(t.max()))
t_fine = np.linspace(-lb,lb,257)[:,None]
g_of_tfine = keister.g(t_fine).squeeze()
ax[0].set_title(r'Original')
ax[0].set_xlabel(r'$T_i$'); ax[0].set_ylabel(r'$g(T_i) = g(\Phi^{-1}(X_i))$')
ax[0].plot(t_fine.squeeze(),g_of_tfine,color=colors[0],alpha=.5)
ax[0].scatter(t.squeeze(),f_of_x,s=10,color='k')
ax[1].set_title(r'Transformed')
ax[1].set_xlabel(r'$X_i$'); ax[1].set_ylabel(r'$f(X_i)$')
ax[1].scatter(x.squeeze(),f_of_x,s=10,color='k')
ax[1].plot(x_fine.squeeze(),f_of_xfine,color=colors[1],alpha=.5);
```



5.27.6 Stopping Criterion

Adaptively increase n until $|\mu - \hat{\mu}| < \varepsilon$ where ε is a user defined tolerance.

The stopping criterion should match the discrete distribution e.g. IID CMC stopping criterion for IID points, QMC Lattice stopping criterion for LD Lattice points, QMC digital net stopping criterion for LD digital net points, etc.

IID CMC Algorithm

```
problem_cmc = qp.AsianOption(qp.IIDStdUniform(52))
cmc_stop_crit = qp.CubMCG(problem_cmc,abs_tol=0.025)
approx_cmc,data_cmc = cmc_stop_crit.integrate()
data_cmc
```

```
MeanVarData (AccumulateData Object)
    solution        1.770
    error_bound     0.025
    n_total         445001
    n               443977
    levels          1
    time_integrate  2.678
CubMCG (StoppingCriterion Object)
    abs_tol         0.025
    rel_tol         0
    n_init          2^(10)
    n_max           10000000000
    inflate         1.200
    alpha           0.010
AsianOption (Integrand Object)
    volatility      2^(-1)
    call_put        call
```

(continues on next page)

(continued from previous page)

```
    start_price     30
    strike_price    35
    interest_rate   0
    mean_type       arithmetic
    dim_frac        0
BrownianMotion (TrueMeasure Object)
    time_vec        [0.019 0.038 0.058 ... 0.962 0.981 1.   ]
    drift           0
    mean            [0. 0. 0. ... 0. 0. 0.]
    covariance      [[0.019 0.019 0.019 ... 0.019 0.019 0.019]
                    [0.019 0.038 0.038 ... 0.038 0.038 0.038]
                    [0.019 0.038 0.058 ... 0.058 0.058 0.058]
                    ...
                    [0.019 0.038 0.058 ... 0.962 0.962 0.962]
                    [0.019 0.038 0.058 ... 0.962 0.981 0.981]
                    [0.019 0.038 0.058 ... 0.962 0.981 1.   ]]
    decomp_type     PCA
IIDStdUniform (DiscreteDistribution Object)
    d               52
    entropy         616843586472568791386361078521518534 77
    spawn_key       ()
```

LD QMC Algorithm

```
problem_qmc = qp.AsianOption(qp.DigitalNetB2(52))
qmc_stop_crit = qp.CubQMCNetG(problem_qmc,abs_tol=0.025)
approx_qmc,data_qmc = qmc_stop_crit.integrate()
data_qmc
```

```
LDTransformData (AccumulateData Object)
    solution        1.784
    comb_bound_low  1.759
    comb_bound_high 1.809
    comb_flags      1
    n_total         2^(10)
    n               2^(10)
    time_integrate  0.008
CubQMCNetG (StoppingCriterion Object)
    abs_tol         0.025
    rel_tol         0
    n_init          2^(10)
    n_max           2^(35)
AsianOption (Integrand Object)
    volatility      2^(-1)
    call_put        call
    start_price     30
    strike_price    35
    interest_rate   0
    mean_type       arithmetic
    dim_frac        0
```

(continues on next page)

(continued from previous page)

```
BrownianMotion (TrueMeasure Object)
    time_vec        [0.019 0.038 0.058 ... 0.962 0.981 1.   ]
    drift           0
    mean            [0. 0. 0. ... 0. 0. 0.]
    covariance      [[0.019 0.019 0.019 ... 0.019 0.019 0.019]
                    [0.019 0.038 0.038 ... 0.038 0.038 0.038]
                    [0.019 0.038 0.058 ... 0.058 0.058 0.058]
                    ...
                    [0.019 0.038 0.058 ... 0.962 0.962 0.962]
                    [0.019 0.038 0.058 ... 0.962 0.981 0.981]
                    [0.019 0.038 0.058 ... 0.962 0.981 1.   ]]
    decomp_type     PCA
DigitalNetB2 (DiscreteDistribution Object)
    d               52
    dvec            [ 0  1  2 ... 49 50 51]
    randomize       LMS_DS
    graycode        0
    entropy         190424263390524682978874000220828251392
    spawn_key       ()
```

```
print('QMC took %.2f%% the time and %.2f%% the samples compared to CMC'%(
      100*data_qmc.time_integrate/data_cmc.time_integrate,100*data_qmc.n_total/data_cmc.
↪n_total))
```

```
QMC took 0.29% the time and 0.23% the samples compared to CMC
```

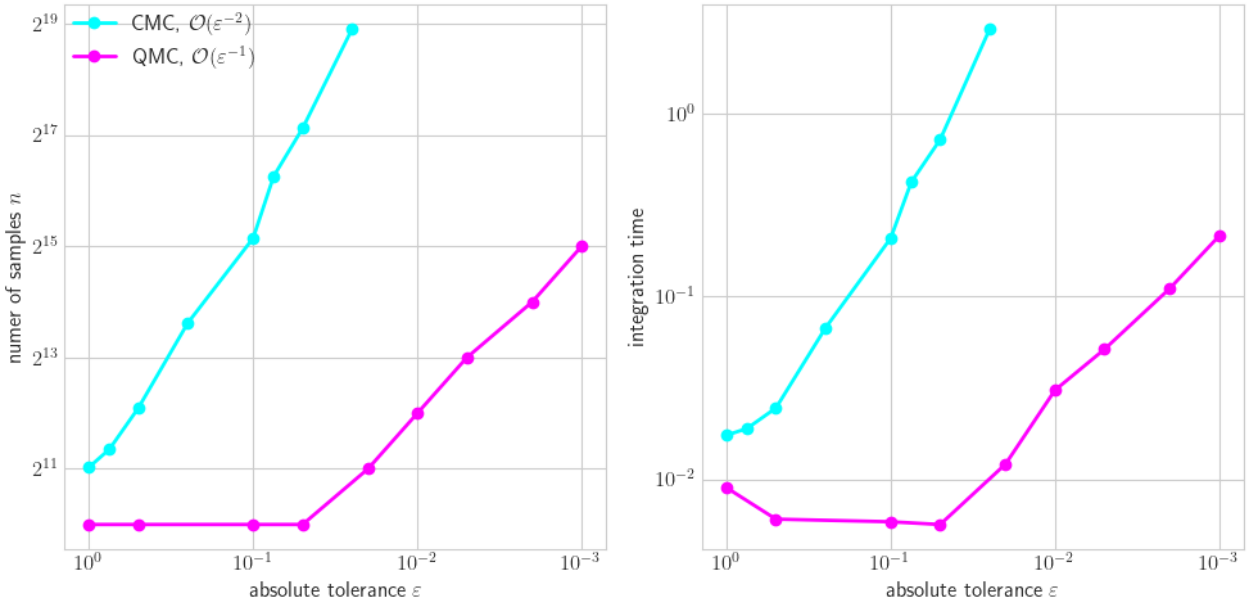
Visual CMC vs LD

```
cmc_tols = [1,.75,.5,.25,.1,.075,.05,.025]
qmc_tols = [1,.5,.1,.05,.02,.01,.005,.002,.001]
fig,ax = pyplot.subplots(nrows=1,ncols=2,figsize=(10,5))
n_cmc,time_cmc = np.zeros_like(cmc_tols),np.zeros_like(cmc_tols)
for i,cmc_tol in enumerate(cmc_tols):
    cmc_stop_crit = qp.CubMCG( qp.AsianOption(qp.IIDStdUniform(52)),abs_tol=cmc_tol)
    approx_cmc,data_cmc = cmc_stop_crit.integrate()
    n_cmc[i],time_cmc[i] = data_cmc.n_total,data_cmc.time_integrate
ax[0].plot(cmc_tols,n_cmc,'-o',color=colors[0],label=r'CMC, $\mathcal{O}(\varepsilon^{-2}
↪)$')
ax[1].plot(cmc_tols,time_cmc,'-o',color=colors[0])
n_qmc,time_qmc = np.zeros_like(qmc_tols),np.zeros_like(qmc_tols)
for i,qmc_tol in enumerate(qmc_tols):
    qmc_stop_crit = qp.CubQMCNetG(qp.AsianOption(qp.DigitalNetB2(52)),abs_tol=qmc_tol)
    approx_qmc,data_qmc = qmc_stop_crit.integrate()
    n_qmc[i],time_qmc[i] = data_qmc.n_total,data_qmc.time_integrate
ax[0].plot(qmc_tols,n_qmc,'-o',color=colors[1],label=r'QMC, $\mathcal{O}(\varepsilon^{-1}
↪)$')
ax[1].plot(qmc_tols,time_qmc,'-o',color=colors[1])
ax[0].set_xscale('log',base=10); ax[0].set_yscale('log',base=2)
ax[1].set_xscale('log',base=10); ax[1].set_yscale('log',base=10)
ax[0].invert_xaxis(); ax[1].invert_xaxis()
```

(continues on next page)

(continued from previous page)

```
ax[0].set_xlabel(r'absolute tolerance $\varepsilon$'); ax[1].set_xlabel(r'absolute␣
↪tolerance $\varepsilon$')
ax[0].set_ylabel(r'numer of samples $n$'); ax[1].set_ylabel('integration time')
ax[0].legend(loc='upper left');
```



5.27.7 Vectorized Stopping Criterion

Many more examples available at https://github.com/QMCSoftware/QMCSoftware/blob/master/demos/vectorized_qmc.ipynb

Vector of Expectations

As a simple example, lets compute $\mathbb{E}[\cos(T_0)\cdots\cos(T_{d-1})]$ and $\mathbb{E}[\sin(T_0)\cdots\sin(T_{d-1})]$ where $T \sim \mathcal{U}[0,\pi]^d$

```
qmc_stop_crit = qp.CubQMCCLT(
    integrand = qp.CustomFun(
        true_measure = qp.Uniform(sampler=qp.Halton(3),lower_bound=0,upper_bound=np.pi),
        g = lambda t,compute_flags: np.vstack([np.cos(t).prod(1),np.sin(t).prod(1)]).T,
        dimension_indv = 2),
    abs_tol=.0001)
approx,data = qmc_stop_crit.integrate()
data
```

```
MeanVarDataRep (AccumulateData Object)
    solution        [2.534e-05 2.580e-01]
    comb_bound_low  [-6.766e-05  2.579e-01]
    comb_bound_high [1.183e-04 2.581e-01]
    comb_flags      [ True  True]
    n_total         2^(18)
    n               [262144.  65536.]
```

(continues on next page)

(continued from previous page)

```
    n_rep           [16384.  4096.]
    time_integrate  0.194
CubQMCCLT (StoppingCriterion Object)
    inflate         1.200
    alpha           0.010
    abs_tol         1.00e-04
    rel_tol         0
    n_init          2^(8)
    n_max           2^(30)
    replications    2^(4)
CustomFun (Integrand Object)
Uniform (TrueMeasure Object)
    lower_bound     0
    upper_bound     3.142
Halton (DiscreteDistribution Object)
    d               3
    dvec            [0 1 2]
    randomize       QRNG
    generalize      1
    entropy         103520389532066966624097310655355259890
    spawn_key       ()
```

Covariance

In a simple example, let $T \sim \mathcal{N}(1, I_d)$ and compute the covariance of $P = T_0 \cdots T_{d-1}$ and $S = T_0 + \cdots + T_{d-1}$ so that

$$\mathrm{Cov}[P, S] = \mathbb{E}[PS] - \mathbb{E}[P]\mathbb{E}[S] = \mu_0 - \mu_1\mu_2$$

Theoretically we have $\mathrm{Cov}[P, S] = 2d - (1)(d) = d$

```
class CovIntegrand(qp.integrand.Integrand):
    def __init__(self,sampler):
        self.sampler = sampler
        self.true_measure = qp.Gaussian(sampler,mean=1)
        super(CovIntegrand,self).__init__(dimension_indv=3,dimension_comb=1,
↪parallel=False)
    def g(self, t, compute_flags):
        y = np.zeros((len(t),3))
        y[:,1] = t.prod(1) # P
        y[:,2] = t.sum(1) # S
        y[:,0] = y[:,1]*y[:,2] #PS
        return y
    def _spawn(self, level, sampler):
        return CovFun(sampler)
    def bound_fun(self, low, high):
        comb_low = low[0]-max(low[1]*low[2],low[1]*high[2],high[1]*low[2],
↪high[1]*high[2])
        comb_high = high[0]-min(low[1]*low[2],low[1]*high[2],high[1]*low[2],
↪high[1]*high[2])
        return comb_low,comb_high
```

(continues on next page)

(continued from previous page)

```
    def dependency(self, comb_flag):
        return np.tile(comb_flag,3)
approx,data = qp.CubQMCLatticeG(CovIntegrand(qp.Lattice(10)),rel_tol=.025).integrate()
data
```

```
LDTransformData (AccumulateData Object)
    solution        9.889
    comb_bound_low  9.697
    comb_bound_high 10.090
    comb_flags      1
    n_total         2^(20)
    n               [1048576. 1048576. 1048576.]
    time_integrate  3.316
CubQMCLatticeG (StoppingCriterion Object)
    abs_tol         0.010
    rel_tol         0.025
    n_init          2^(10)
    n_max           2^(35)
CovIntegrand (Integrand Object)
Gaussian (TrueMeasure Object)
    mean            1
    covariance      1
    decomp_type     PCA
Lattice (DiscreteDistribution Object)
    d               10
    dvec            [0 1 2 3 4 5 6 7 8 9]
    randomize       1
    order           natural
    gen_vec         [     1 182667 469891 498753 110745 446247 250185 118627 245333␣
↪283199]
    entropy         333426925650737605782635567565068085620
    spawn_key       ()
```

Sensitiviy Indices

See Appendix A of Art Owen's Monte Carlo Book

In the following example, we fit a neural network to Iris flower features and try to classify the Iris species. For each set of features, the classifier provides a probability of belonging to each species, a length 3 vector. We quantify the sensitiviy of this classificaiton probability to Iris features, assuming features are uniformly distributed throughout the feature domain.

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPClassifier
data = load_iris()
og_feature_names = data["feature_names"]
feature_names = [fn.replace('sepal ','S')\
    .replace('length ','L')\
    .replace('petal ','P')\
    .replace('width ','W')\
```

(continues on next page)

(continued from previous page)

```
    .replace('(cm)','') for fn in og_feature_names]
target_names = data["target_names"]
xt,xv,yt,yv = train_test_split(data["data"],data["target"],
    test_size = 1/3,
    random_state = 7)
pd.DataFrame(np.hstack([data['data'],data['target'][:,None]]),columns=og_feature_names+[
↪'species']).iloc[[0,1,90,91,140,141]]
```

```
mlpc = MLPClassifier(random_state=7,max_iter=1024).fit(xt,yt)
yhat = mlpc.predict(xv)
print("accuracy: %.1f%%"%(100*(yv==yhat).mean()))
# accuracy: 98.0%
sampler = qp.DigitalNetB2(4,seed=7)
true_measure =  qp.Uniform(sampler,
    lower_bound = xt.min(0),
    upper_bound = xt.max(0))
fun = qp.CustomFun(
    true_measure = true_measure,
    g = lambda x,compute_flags: mlpc.predict_proba(x),
    dimension_indv = 3)
si_fun = qp.SensitivityIndices(fun,indices="all")
qmc_algo = qp.CubQMCNetG(si_fun,abs_tol=.005)
nn_sis,nn_sis_data = qmc_algo.integrate()
```

```
accuracy: 98.0%
```

```
#print(nn_sis_data.flags_indv.shape)
#print(nn_sis_data.flags_comb.shape)
print('samples: 2^(%d)'%np.log2(nn_sis_data.n_total))
print('time: %.1e'%nn_sis_data.time_integrate)
print('indices:',nn_sis_data.integrand.indices)

import pandas as pd

df_closed = pd.DataFrame(nn_sis[0],columns=target_names,index=[str(idx) for idx in nn_
↪sis_data.integrand.indices])
print('\nClosed Indices')
print(df_closed)
df_total = pd.DataFrame(nn_sis[1],columns=target_names,index=[str(idx) for idx in nn_sis_
↪data.integrand.indices])
print('\nTotal Indices')
print(df_total)
df_closed_singletons = df_closed.T.iloc[:,:4]
df_closed_singletons['sum singletons'] = df_closed_singletons[['[%d]'%i for i in␣
↪range(4)]].sum(1)
df_closed_singletons.columns = data['feature_names']+['sum']
df_closed_singletons = df_closed_singletons*100
```

```
samples: 2^(15)
time: 1.6e+00
indices: [[0], [1], [2], [3], [0, 1], [0, 2], [0, 3], [1, 2], [1, 3], [2, 3], [0, 1, 2],␣
↪[0, 1, 3], [0, 2, 3], [1, 2, 3]]
```

(continues on next page)

(continued from previous page)

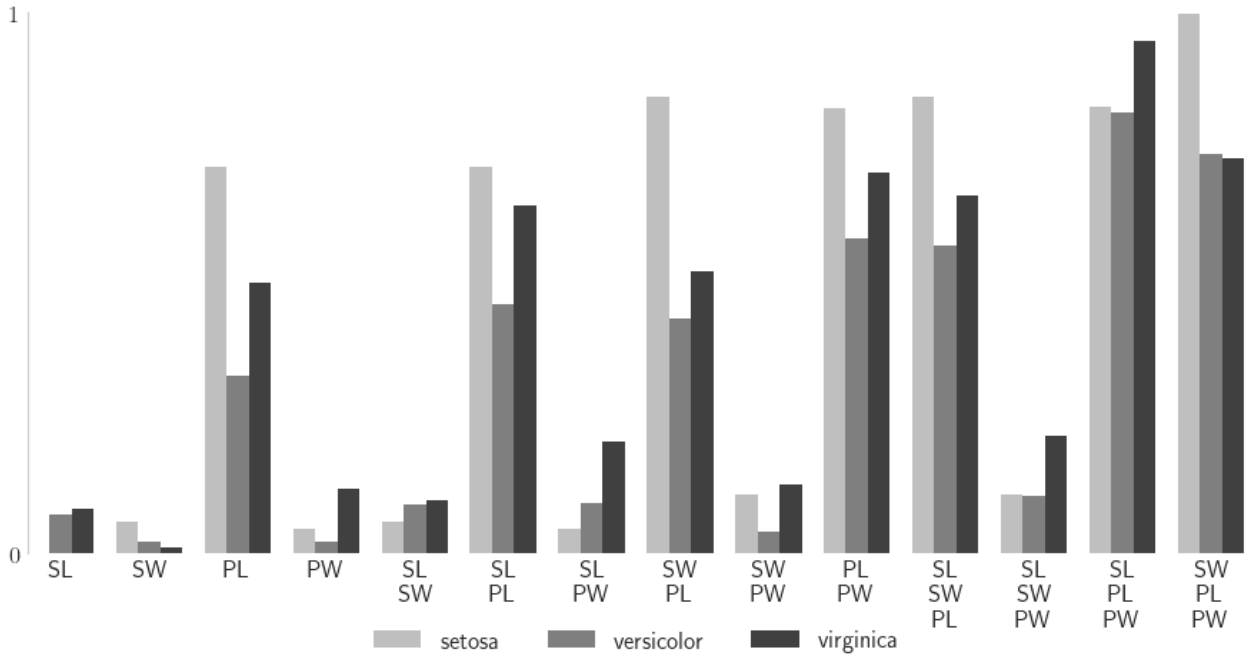
```
Closed Indices
             setosa  versicolor  virginica
[0]        0.001504    0.071122   0.081736
[1]        0.058743    0.022073   0.010373
[2]        0.713777    0.328313   0.500059
[3]        0.046053    0.021077   0.120233
[0, 1]     0.059178    0.091764   0.098233
[0, 2]     0.715117    0.460138   0.642551
[0, 3]     0.046859    0.092322   0.207724
[1, 2]     0.843241    0.434629   0.520469
[1, 3]     0.108872    0.039572   0.127844
[2, 3]     0.823394    0.582389   0.705354
[0, 1, 2]  0.845359    0.570022   0.661100
[0, 1, 3]  0.108503    0.106081   0.218971
[0, 2, 3]  0.825389    0.814286   0.948331
[1, 2, 3]  0.996483    0.738213   0.729940

Total Indices
             setosa  versicolor  virginica
[0]        0.003199    0.259762   0.265085
[1]        0.172391    0.183159   0.045582
[2]        0.889677    0.896874   0.780377
[3]        0.157794    0.433342   0.340092
[0, 1]     0.174905    0.414246   0.289565
[0, 2]     0.890477    0.966238   0.871992
[0, 3]     0.159744    0.566187   0.478082
[1, 2]     0.949190    0.907994   0.790445
[1, 3]     0.283542    0.540486   0.355714
[2, 3]     0.941651    0.919005   0.902203
[0, 1, 2]  0.949431    0.980367   0.880283
[0, 1, 3]  0.284118    0.674406   0.497364
[0, 2, 3]  0.942185    0.986186   0.989555
[1, 2, 3]  0.996057    0.933342   0.913711
```

```
nindices = len(nn_sis_data.integrand.indices)
fig,ax = pyplot.subplots(figsize=(9,5))
ticks = np.arange(nindices)
width = .25
for i,(alpha,species) in enumerate(zip([.25,.5,.75],data['target_names'])):
    cvals = df_closed[species].to_numpy()
    tvals = df_total[species].to_numpy()
    ticks_i = ticks+i*width
    ax.bar(ticks_i,cvals,width=width,align='edge',color='k',alpha=alpha,label=species)
    #ax.bar(ticks_i,np.flip(tvals),width=width,align='edge',bottom=1-np.flip(tvals),
↪color=color,alpha=.1)
ax.set_xlim([0,13+3*width])
ax.set_xticks(ticks+1.5*width)

# closed_labels = [r'$\underline{s}_{\{%s\}}$'%(','.join([r'\text{%s}'%feature_names[i] for␣
↪i in idx])) for idx in nn_sis_data.integrand.indices]
closed_labels = ['\n'.join([feature_names[i] for i in idx]) for idx in nn_sis_data.
↪integrand.indices]
```

(continues on next page)

(continued from previous page)

```
ax.set_xticklabels(closed_labels,rotation=0)
ax.set_ylim([0,1]); ax.set_yticks([0,1])
ax.grid(False)
for spine in ['top','right','bottom']: ax.spines[spine].set_visible(False)
ax.legend(frameon=False,loc='lower center',bbox_to_anchor=(.5,-.2),ncol=3);
```



5.28 Probability of Failure Estimation with Gaussian Processes

```
!pip install botorch -U
```

```
Requirement already satisfied: botorch in /Users/alegresor/miniconda3/envs/qmcpy/lib/
↪python3.9/site-packages (0.9.2)
Requirement already satisfied: linear-operator==0.5.1 in /Users/alegresor/miniconda3/
↪envs/qmcpy/lib/python3.9/site-packages (from botorch) (0.5.1)
Requirement already satisfied: multipledispatch in /Users/alegresor/miniconda3/envs/
↪qmcpy/lib/python3.9/site-packages (from botorch) (1.0.0)
Requirement already satisfied: gpytorch==1.11 in /Users/alegresor/miniconda3/envs/qmcpy/
↪lib/python3.9/site-packages (from botorch) (1.11)
Requirement already satisfied: pyro-ppl>=1.8.4 in /Users/alegresor/miniconda3/envs/qmcpy/
↪lib/python3.9/site-packages (from botorch) (1.8.6)
Requirement already satisfied: torch>=1.13.1 in /Users/alegresor/miniconda3/envs/qmcpy/
↪lib/python3.9/site-packages (from botorch) (2.0.1)
Requirement already satisfied: scipy in /Users/alegresor/miniconda3/envs/qmcpy/lib/
↪python3.9/site-packages (from botorch) (1.9.3)
Requirement already satisfied: scikit-learn in /Users/alegresor/miniconda3/envs/qmcpy/
↪lib/python3.9/site-packages (from gpytorch==1.11->botorch) (1.3.0)
Requirement already satisfied: typeguard~=2.13.3 in /Users/alegresor/miniconda3/envs/
↪qmcpy/lib/python3.9/site-packages (from linear-operator==0.5.1->botorch) (2.13.3)
```

(continues on next page)

(continued from previous page)

```
Requirement already satisfied: jaxtyping>=0.2.9 in /Users/alegresor/miniconda3/envs/
↪qmcpy/lib/python3.9/site-packages (from linear-operator==0.5.1->botorch) (0.2.20)
Requirement already satisfied: numpy>=1.7 in /Users/alegresor/miniconda3/envs/qmcpy/lib/
↪python3.9/site-packages (from pyro-ppl>=1.8.4->botorch) (1.23.4)
Requirement already satisfied: pyro-api>=0.1.1 in /Users/alegresor/miniconda3/envs/qmcpy/
↪lib/python3.9/site-packages (from pyro-ppl>=1.8.4->botorch) (0.1.2)
Requirement already satisfied: opt-einsum>=2.3.2 in /Users/alegresor/miniconda3/envs/
↪qmcpy/lib/python3.9/site-packages (from pyro-ppl>=1.8.4->botorch) (3.3.0)
Requirement already satisfied: tqdm>=4.36 in /Users/alegresor/miniconda3/envs/qmcpy/lib/
↪python3.9/site-packages (from pyro-ppl>=1.8.4->botorch) (4.66.1)
Requirement already satisfied: networkx in /Users/alegresor/miniconda3/envs/qmcpy/lib/
↪python3.9/site-packages (from torch>=1.13.1->botorch) (3.1)
Requirement already satisfied: typing-extensions in /Users/alegresor/miniconda3/envs/
↪qmcpy/lib/python3.9/site-packages (from torch>=1.13.1->botorch) (4.7.1)
Requirement already satisfied: filelock in /Users/alegresor/miniconda3/envs/qmcpy/lib/
↪python3.9/site-packages (from torch>=1.13.1->botorch) (3.12.2)
Requirement already satisfied: jinja2 in /Users/alegresor/miniconda3/envs/qmcpy/lib/
↪python3.9/site-packages (from torch>=1.13.1->botorch) (3.1.2)
Requirement already satisfied: sympy in /Users/alegresor/miniconda3/envs/qmcpy/lib/
↪python3.9/site-packages (from torch>=1.13.1->botorch) (1.12)
Requirement already satisfied: MarkupSafe>=2.0 in /Users/alegresor/miniconda3/envs/qmcpy/
↪lib/python3.9/site-packages (from jinja2->torch>=1.13.1->botorch) (2.1.3)
Requirement already satisfied: joblib>=1.1.1 in /Users/alegresor/miniconda3/envs/qmcpy/
↪lib/python3.9/site-packages (from scikit-learn->gpytorch==1.11->botorch) (1.3.1)
Requirement already satisfied: threadpoolctl>=2.0.0 in /Users/alegresor/miniconda3/envs/
↪qmcpy/lib/python3.9/site-packages (from scikit-learn->gpytorch==1.11->botorch) (3.2.0)
Requirement already satisfied: mpmath>=0.19 in /Users/alegresor/miniconda3/envs/qmcpy/
↪lib/python3.9/site-packages (from sympy->torch>=1.13.1->botorch) (1.3.0)
```

```
import qmcpy as qp
import gpytorch
import torch
import os
import warnings
import pandas as pd
from gpytorch.utils.warnings import NumericalWarning
warnings.filterwarnings("ignore")
pd.set_option(
    'display.max_rows', None,
    'display.max_columns', None,
    'display.width', 1000,
    'display.colheader_justify', 'center',
    'display.precision',2,
    'display.float_format',lambda x:'%.1e'%x)
from matplotlib import pyplot
pyplot.style.use("../qmcpy/qmcpy.mplstyle")
```

```
!pip install botorch -U --quiet
```

```
import qmcpy as qp
import pandas as pd
```

(continues on next page)

(continued from previous page)

```
pd.set_option(
    'display.max_rows', None,
    'display.max_columns', None,
    'display.width', 1000,
    'display.colheader_justify', 'center',
    'display.precision',2,
    'display.float_format',lambda x:'%.1e'%x)
from matplotlib import pyplot
pyplot.style.use("../qmcpy/qmcpy.mplstyle")
```

```
gpytorch_use_gpu = torch.cuda.is_available()
gpytorch_use_gpu
```

```
False
```

5.28.1 Sin 1d Problem

```
mcispfgp = qp.PFGPCI(
    integrand = qp.Sin1d(qp.DigitalNetB2(1,seed=17),k=3),
    failure_threshold = 0,
    failure_above_threshold=True,
    abs_tol = 1e-2,
    alpha = 1e-1,
    n_init = 4,
    init_samples = None,
    batch_sampler = qp.PFSampleErrorDensityAR(verbose=True),
    n_batch = 4,
    n_max = 20,
    n_approx = 2**18,
    gpytorch_prior_mean = gpytorch.means.ZeroMean(),
    gpytorch_prior_cov = gpytorch.kernels.ScaleKernel(
        gpytorch.kernels.MaternKernel(nu=2.5,
            lengthscale_constraint = gpytorch.constraints.Interval(.01,.1)
            ),
        outputscale_constraint = gpytorch.constraints.Interval(1e-3,10)
        ),
    gpytorch_likelihood = gpytorch.likelihoods.GaussianLikelihood(noise_constraint =␣
↪gpytorch.constraints.Interval(1e-12,1e-8)),
    gpytorch_marginal_log_likelihood_func = lambda likelihood,gpyt_model: gpytorch.mlls.
↪ExactMarginalLogLikelihood(likelihood,gpyt_model),
    torch_optimizer_func = lambda gpyt_model: torch.optim.Adam(gpyt_model.parameters(),
↪lr=0.1),
    gpytorch_train_iter = 100,
    gpytorch_use_gpu = False,
    verbose = 50,
    n_ref_approx = 2**22,
    seed_ref_approx = None)
solution,data = mcispfgp.integrate(seed=7,refit=False)
print(data)
df = pd.DataFrame(data.get_results_dict())
```

(continues on next page)

(continued from previous page)

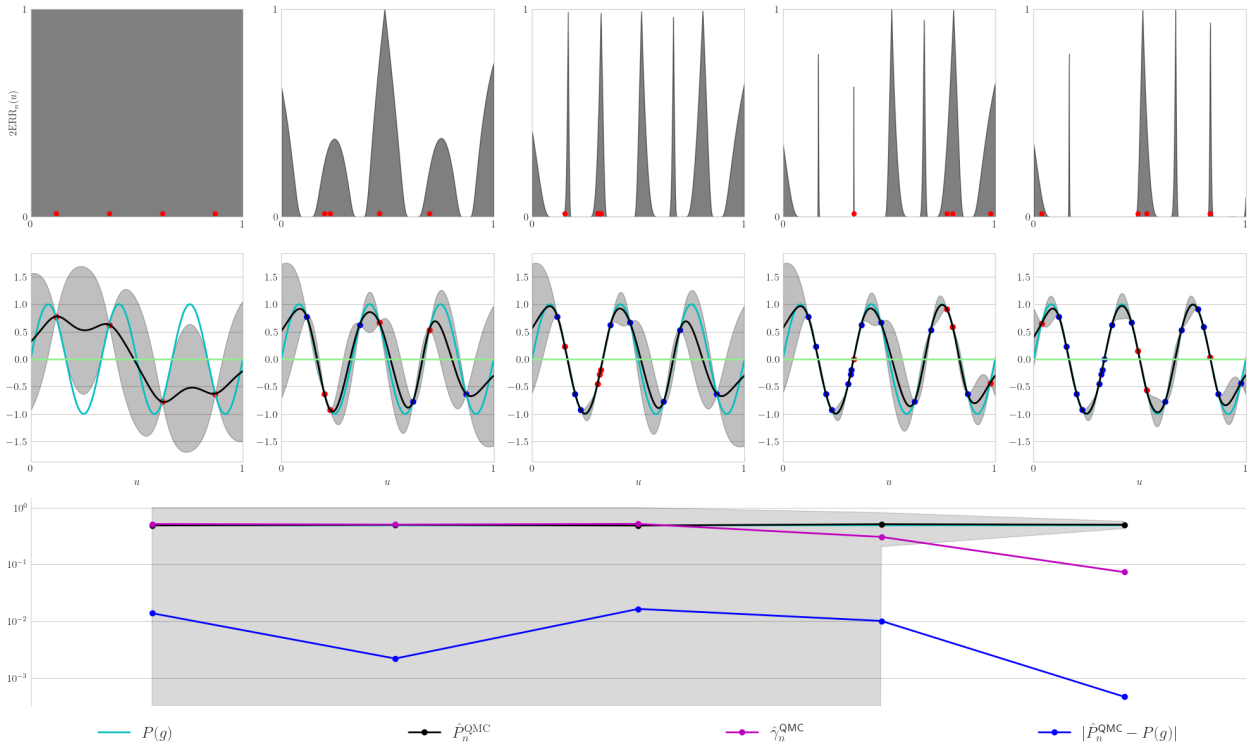
```
print("\nIteration Summary")
print(df)
data.plot();
```

```
reference approximation with d=1: 0.5000002384185791
batch 0
    gpytorch model fitting
            iter 50  of 100
                    likelihood.noise_covar.raw_noise.................. -8.63e-02
                    covar_module.raw_outputscale...................... -3.04e+00
                    covar_module.base_kernel.raw_lengthscale.......... 5.04e+00
            iter 100 of 100
                    likelihood.noise_covar.raw_noise.................. -1.01e-01
                    covar_module.raw_outputscale...................... -2.96e+00
                    covar_module.base_kernel.raw_lengthscale.......... 6.29e+00
batch 1
    AR sampling with efficiency 2.6e-01, expect 15 draws: 12, 15, 18, 21, 24,
batch 2
    AR sampling with efficiency 1.7e-01, expect 23 draws: 16, 20,
batch 3
    AR sampling with efficiency 1.3e-01, expect 30 draws: 20, 25,
batch 4
    AR sampling with efficiency 6.1e-02, expect 65 draws: 48, 72,
PFGPCIData (AccumulateData Object)
    solution        0.500
    error_bound     0.073
    bound_low       0.426
    bound_high      0.573
    n_total         20
    time_integrate  0.644
PFGPCI (StoppingCriterion Object)
Sin1d (Integrand Object)
Uniform (TrueMeasure Object)
    lower_bound     0
    upper_bound     18.850
DigitalNetB2 (DiscreteDistribution Object)
    d               1
    dvec            0
    randomize       LMS_DS
    graycode        0
    entropy         17
    spawn_key       ()

Iteration Summary
   n_sum  n_batch  error_bounds  ci_low  ci_high  solutions  solutions_ref  error_ref ␣
↪in_ci
0    4       4       5.1e-01    0.0e+00 1.0e+00   4.9e-01      5.0e-01      1.4e-02   ␣
↪True
1    8       4       5.0e-01    0.0e+00 1.0e+00   5.0e-01      5.0e-01      2.2e-03   ␣
↪True
2   12       4       5.2e-01    0.0e+00 1.0e+00   4.8e-01      5.0e-01      1.6e-02   ␣
↪True
```

(continues on next page)

(continued from previous page)

```
3    16        4        3.0e-01    2.1e-01 8.1e-01    5.1e-01        5.0e-01        1.0e-02    ␣
↪True
4    20        4        7.3e-02    4.3e-01 5.7e-01    5.0e-01        5.0e-01        4.7e-04    ␣
↪True
```



5.28.2 Multimodal 2d Problem

```
mcispfgp = qp.PFGPCI(
    integrand = qp.Multimodal2d(qp.DigitalNetB2(2,seed=17)),
    failure_threshold = 0,
    failure_above_threshold=True,
    abs_tol = 1e-2,
    alpha = 1e-1,
    n_init = 64,
    init_samples = None,
    batch_sampler = qp.PFSampleErrorDensityAR(verbose=True),
    n_batch = 16,
    n_max = 128,
    n_approx = 2**18,
    gpytorch_prior_mean = gpytorch.means.ZeroMean(),
    gpytorch_prior_cov = gpytorch.kernels.ScaleKernel(
        gpytorch.kernels.MaternKernel(nu=1.5,
            lengthscale_constraint = gpytorch.constraints.Interval(.1,1)
            ),
        outputscale_constraint = gpytorch.constraints.Interval(1e-3,.5)
        ),
```

(continues on next page)

(continued from previous page)

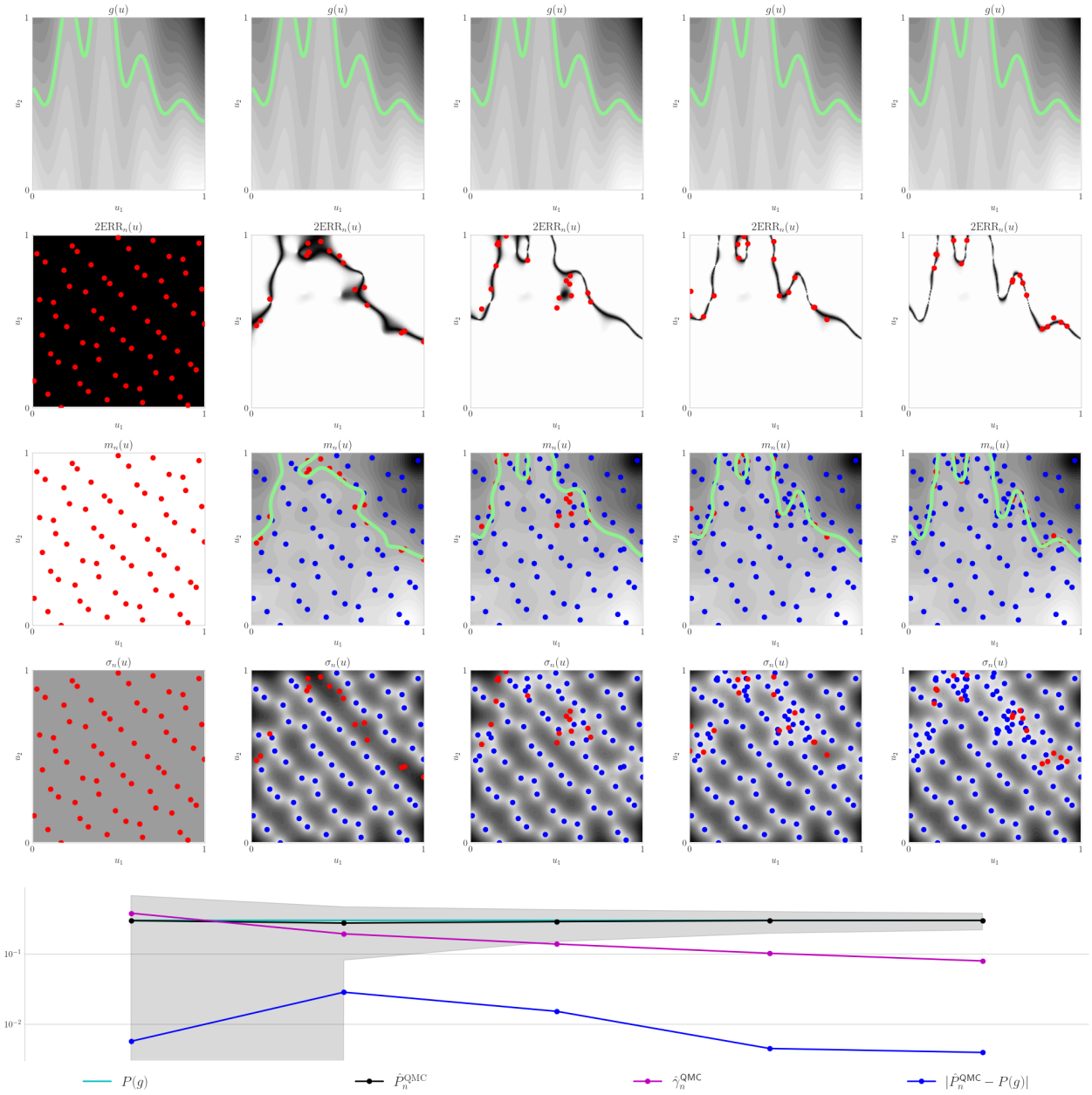
```
    gpytorch_likelihood = gpytorch.likelihoods.GaussianLikelihood(noise_constraint =␣
↪gpytorch.constraints.Interval(1e-12,1e-8)),
    gpytorch_marginal_log_likelihood_func = lambda likelihood,gpyt_model: gpytorch.mlls.
↪ExactMarginalLogLikelihood(likelihood,gpyt_model),
    torch_optimizer_func = lambda gpyt_model: torch.optim.Adam(gpyt_model.parameters(),
↪lr=0.1),
    gpytorch_train_iter = 800,
    gpytorch_use_gpu = gpytorch_use_gpu,
    verbose = 200,
    n_ref_approx = 2**22,
    seed_ref_approx = None)
solution,data = mcispfgp.integrate(seed=7,refit=False)
print(data)
df = pd.DataFrame(data.get_results_dict())
print("\nIteration Summary")
print(df)
data.plot();
```

```
reference approximation with d=2: 0.3020772933959961
batch 0
    gpytorch model fitting
            iter 200 of 800
                    likelihood.noise_covar.raw_noise..... 1.45e+00
                    covar_module.raw_outputscale..... 2.92e+00
                    covar_module.base_kernel.raw_lengthscale..... -3.02e+00
            iter 400 of 800
                    likelihood.noise_covar.raw_noise..... 1.49e+00
                    covar_module.raw_outputscale..... 3.70e+00
                    covar_module.base_kernel.raw_lengthscale..... -3.32e+00
            iter 600 of 800
                    likelihood.noise_covar.raw_noise..... 1.53e+00
                    covar_module.raw_outputscale..... 4.24e+00
                    covar_module.base_kernel.raw_lengthscale..... -3.43e+00
            iter 800 of 800
                    likelihood.noise_covar.raw_noise..... 1.58e+00
                    covar_module.raw_outputscale..... 4.66e+00
                    covar_module.base_kernel.raw_lengthscale..... -3.48e+00
batch 1
    AR sampling with efficiency 7.6e-02, expect 211 draws: 144, 198,
batch 2
    AR sampling with efficiency 3.8e-02, expect 416 draws: 288,
batch 3
    AR sampling with efficiency 2.8e-02, expect 581 draws: 400, 500, 525,
batch 4
    AR sampling with efficiency 2.0e-02, expect 785 draws: 544, 748, 816, 884,
PFGPCIData (AccumulateData Object)
    solution        0.298
    error_bound     0.079
    bound_low       0.219
    bound_high      0.377
    n_total         128
    time_integrate  4.295
```

(continues on next page)

(continued from previous page)

```
PFGPCI (StoppingCriterion Object)
Multimodal2d (Integrand Object)
Uniform (TrueMeasure Object)
    lower_bound     [-4 -3]
    upper_bound     [7 8]
DigitalNetB2 (DiscreteDistribution Object)
    d               2^(1)
    dvec            [0 1]
    randomize       LMS_DS
    graycode        0
    entropy         17
    spawn_key       ()

Iteration Summary
   n_sum  n_batch  error_bounds  ci_low  ci_high  solutions  solutions_ref  error_ref ␣
↪in_ci
0    64      64       3.8e-01    0.0e+00 6.7e-01   3.0e-01      3.0e-01      5.7e-03   ␣
↪True
1    80      16       1.9e-01    8.1e-02 4.7e-01   2.7e-01      3.0e-01      2.8e-02   ␣
↪True
2    96      16       1.4e-01    1.5e-01 4.2e-01   2.9e-01      3.0e-01      1.5e-02   ␣
↪True
3   112      16       1.0e-01    2.0e-01 4.0e-01   3.0e-01      3.0e-01      4.5e-03   ␣
↪True
4   128      16       7.9e-02    2.2e-01 3.8e-01   3.0e-01      3.0e-01      4.0e-03   ␣
↪True
```



5.28.3 Four Branch 2d Problem

```
mcispfgp = qp.PFGPCI(
    integrand = qp.FourBranch2d(qp.DigitalNetB2(2,seed=17)),
    failure_threshold = 0,
    failure_above_threshold=True,
    abs_tol = 1e-2,
    alpha = 1e-1,
    n_init = 64,
    init_samples = None,
    batch_sampler = qp.PFSampleErrorDensityAR(verbose=True),
```

(continues on next page)

(continued from previous page)

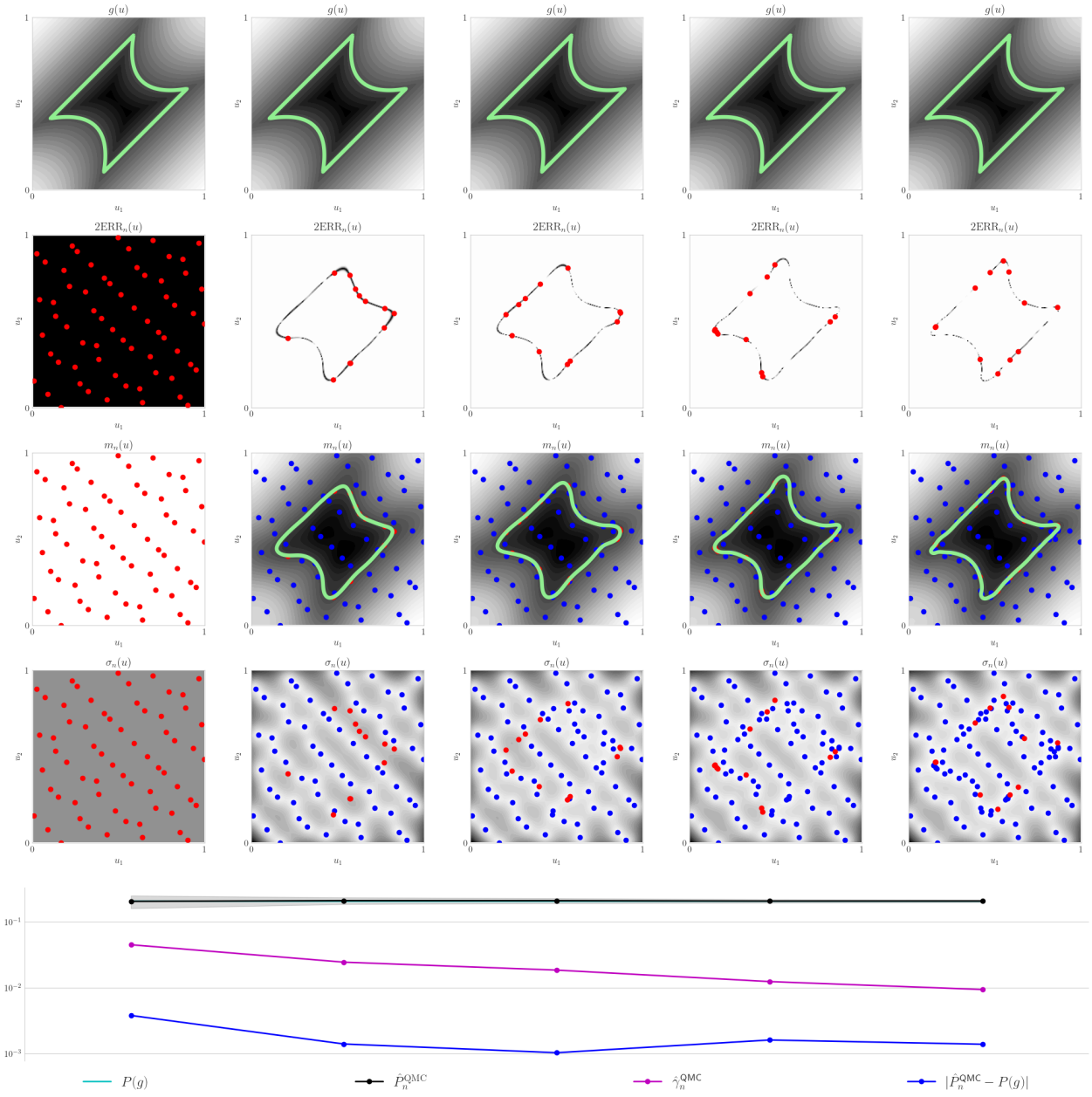
```
    n_batch = 12,
    n_max = 200,
    n_approx = 2**18,
    gpytorch_prior_mean = gpytorch.means.ZeroMean(),
    gpytorch_prior_cov = gpytorch.kernels.ScaleKernel(
        gpytorch.kernels.MaternKernel(nu=1.5,
            lengthscale_constraint = gpytorch.constraints.Interval(.5,1)
            ),
        outputscale_constraint = gpytorch.constraints.Interval(1e-8,.5)
        ),
    gpytorch_likelihood = gpytorch.likelihoods.GaussianLikelihood(noise_constraint =␣
↪gpytorch.constraints.Interval(1e-12,1e-8)),
    gpytorch_marginal_log_likelihood_func = lambda likelihood,gpyt_model: gpytorch.mlls.
↪ExactMarginalLogLikelihood(likelihood,gpyt_model),
    torch_optimizer_func = lambda gpyt_model: torch.optim.Adam(gpyt_model.parameters(),
↪lr=0.1),
    gpytorch_train_iter = 800,
    gpytorch_use_gpu = gpytorch_use_gpu,
    verbose = 200,
    n_ref_approx = 2**22,
    seed_ref_approx = None)
solution,data = mcispfgp.integrate(seed=7,refit=False)
print(data)
df = pd.DataFrame(data.get_results_dict())
print("\nIteration Summary")
print(df)
data.plot();
```

```
reference approximation with d=2: 0.20872807502746582
batch 0
    gpytorch model fitting
            iter 200 of 800
                    likelihood.noise_covar.raw_noise..... 2.89e+00
                    covar_module.raw_outputscale..... 3.88e+00
                    covar_module.base_kernel.raw_lengthscale..... -4.50e+00
            iter 400 of 800
                    likelihood.noise_covar.raw_noise..... 3.63e+00
                    covar_module.raw_outputscale..... 4.78e+00
                    covar_module.base_kernel.raw_lengthscale..... -5.44e+00
            iter 600 of 800
                    likelihood.noise_covar.raw_noise..... 4.17e+00
                    covar_module.raw_outputscale..... 5.36e+00
                    covar_module.base_kernel.raw_lengthscale..... -6.04e+00
            iter 800 of 800
                    likelihood.noise_covar.raw_noise..... 4.59e+00
                    covar_module.raw_outputscale..... 5.80e+00
                    covar_module.base_kernel.raw_lengthscale..... -6.48e+00
batch 1
    AR sampling with efficiency 9.0e-03, expect 1337 draws: 924, 1232,
batch 2
    AR sampling with efficiency 4.9e-03, expect 2461 draws: 1704, 2556,
batch 3
```

(continues on next page)

(continued from previous page)

```
    AR sampling with efficiency 3.7e-03, expect 3250 draws: 2256,
batch 4
    AR sampling with efficiency 2.5e-03, expect 4858 draws: 3372, 3653, 3934,
PFGPCIData (AccumulateData Object)
    solution        0.207
    error_bound     0.009
    bound_low       0.198
    bound_high      0.217
    n_total         112
    time_integrate  4.376
PFGPCI (StoppingCriterion Object)
FourBranch2d (Integrand Object)
Uniform (TrueMeasure Object)
    lower_bound     -8
    upper_bound     2^(3)
DigitalNetB2 (DiscreteDistribution Object)
    d               2^(1)
    dvec            [0 1]
    randomize       LMS_DS
    graycode        0
    entropy         17
    spawn_key       ()

Iteration Summary
   n_sum  n_batch  error_bounds  ci_low  ci_high  solutions  solutions_ref  error_ref␣
↪in_ci
0     64      64        4.5e-02    1.6e-01 2.5e-01   2.0e-01       2.1e-01       3.8e-03   ␣
↪True
1     76      12        2.4e-02    1.9e-01 2.3e-01   2.1e-01       2.1e-01       1.4e-03   ␣
↪True
2     88      12        1.8e-02    1.9e-01 2.3e-01   2.1e-01       2.1e-01       1.0e-03   ␣
↪True
3    100      12        1.2e-02    1.9e-01 2.2e-01   2.1e-01       2.1e-01       1.6e-03   ␣
↪True
4    112      12        9.4e-03    2.0e-01 2.2e-01   2.1e-01       2.1e-01       1.4e-03   ␣
↪True
```



5.28.4 Ishigami 3d Problem

```
mcispfgp = qp.PFGPCI(
    integrand = qp.Ishigami(qp.DigitalNetB2(3,seed=17)),
    failure_threshold = 0,
    failure_above_threshold=False,
    abs_tol = 1e-2,
    alpha = 1e-1,
    n_init = 128,
    init_samples = None,
    batch_sampler = qp.PFSampleErrorDensityAR(verbose=True),
```

(continues on next page)

(continued from previous page)

```
    n_batch = 16,
    n_max = 256,
    n_approx = 2**18,
    gpytorch_prior_mean = gpytorch.means.ZeroMean(),
    gpytorch_prior_cov = gpytorch.kernels.ScaleKernel(
        gpytorch.kernels.MaternKernel(nu=2.5,
            lengthscale_constraint = gpytorch.constraints.Interval(.5,1)
            ),
        outputscale_constraint = gpytorch.constraints.Interval(1e-8,.5)
        ),
    gpytorch_likelihood = gpytorch.likelihoods.GaussianLikelihood(noise_constraint =␣
↪gpytorch.constraints.Interval(1e-12,1e-8)),
    gpytorch_marginal_log_likelihood_func = lambda likelihood,gpyt_model: gpytorch.mlls.
↪ExactMarginalLogLikelihood(likelihood,gpyt_model),
    torch_optimizer_func = lambda gpyt_model: torch.optim.Adam(gpyt_model.parameters(),
↪lr=0.1),
    gpytorch_train_iter = 800,
    gpytorch_use_gpu = gpytorch_use_gpu,
    verbose = 200,
    n_ref_approx = 2**22,
    seed_ref_approx = None)
solution,data = mcispfgp.integrate(seed=7,refit=False)
print(data)
df = pd.DataFrame(data.get_results_dict())
print("\nIteration Summary")
print(df)
data.plot();
```

```
reference approximation with d=3: 0.16239547729492188
batch 0
    gpytorch model fitting
            iter 200 of 800
                    likelihood.noise_covar.raw_noise..... 2.18e+00
                    covar_module.raw_outputscale..... 3.43e+00
                    covar_module.base_kernel.raw_lengthscale..... -4.09e+00
            iter 400 of 800
                    likelihood.noise_covar.raw_noise..... 2.66e+00
                    covar_module.raw_outputscale..... 4.26e+00
                    covar_module.base_kernel.raw_lengthscale..... -4.99e+00
            iter 600 of 800
                    likelihood.noise_covar.raw_noise..... 3.08e+00
                    covar_module.raw_outputscale..... 4.81e+00
                    covar_module.base_kernel.raw_lengthscale..... -5.56e+00
            iter 800 of 800
                    likelihood.noise_covar.raw_noise..... 3.43e+00
                    covar_module.raw_outputscale..... 5.24e+00
                    covar_module.base_kernel.raw_lengthscale..... -6.00e+00
batch 1
    AR sampling with efficiency 6.1e-03, expect 2626 draws: 1824,
batch 2
    AR sampling with efficiency 5.3e-03, expect 3019 draws: 2096, 3013, 3668, 3930, 4061,
↪ 4192, 4323, 4454, 4585,
```

(continues on next page)

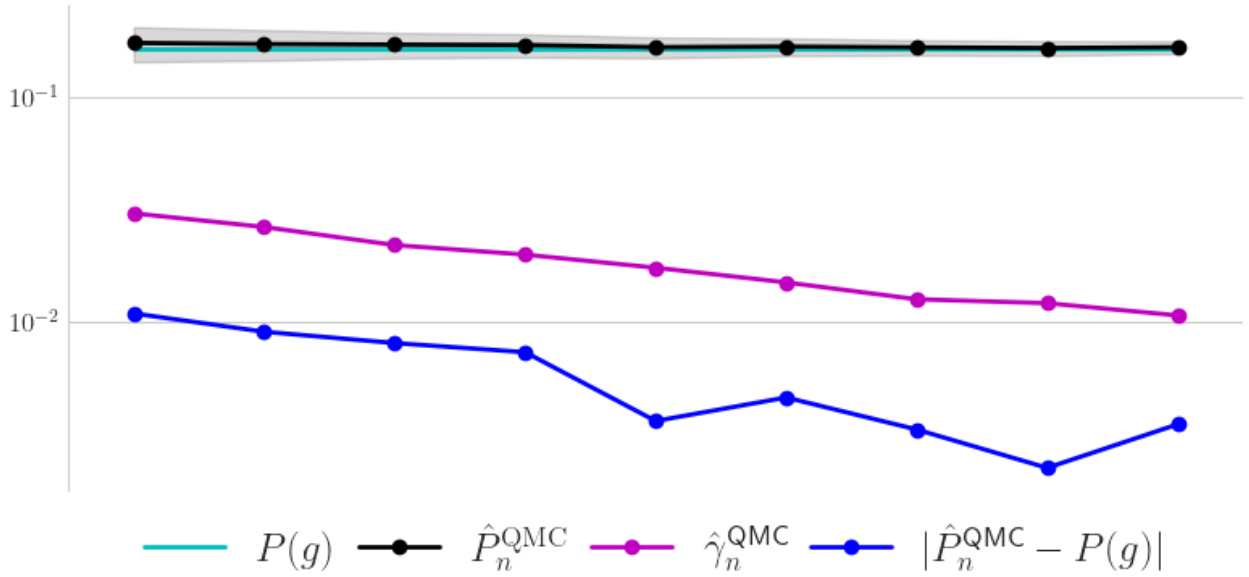
(continued from previous page)

```
batch 3
    AR sampling with efficiency 4.4e-03, expect 3631 draws: 2512, 4082, 4710, 4867, 5024,
batch 4
    AR sampling with efficiency 4.0e-03, expect 4003 draws: 2784,
batch 5
    AR sampling with efficiency 3.5e-03, expect 4576 draws: 3168, 4158, 4554, 4950,
batch 6
    AR sampling with efficiency 3.0e-03, expect 5315 draws: 3680, 4600, 5060, 5520,
batch 7
    AR sampling with efficiency 2.5e-03, expect 6313 draws: 4384,
batch 8
    AR sampling with efficiency 2.4e-03, expect 6570 draws: 4560, 5985, 6270, 6555, 6840,
↪ 7125, 7410,
PFGPCIData (AccumulateData Object)
    solution        0.166
    error_bound     0.011
    bound_low       0.155
    bound_high      0.177
    n_total         256
    time_integrate  14.351
PFGPCI (StoppingCriterion Object)
Ishigami (Integrand Object)
Uniform (TrueMeasure Object)
    lower_bound     -3.142
    upper_bound     3.142
DigitalNetB2 (DiscreteDistribution Object)
    d               3
    dvec            [0 1 2]
    randomize       LMS_DS
    graycode        0
    entropy         17
    spawn_key       ()

Iteration Summary
   n_sum  n_batch  error_bounds  ci_low  ci_high  solutions  solutions_ref  error_ref ␣
↪in_ci
0   128     128      3.0e-02    1.4e-01 2.0e-01   1.7e-01      1.6e-01      1.1e-02   ␣
↪True
1   144      16      2.6e-02    1.4e-01 2.0e-01   1.7e-01      1.6e-01      9.1e-03   ␣
↪True
2   160      16      2.2e-02    1.5e-01 1.9e-01   1.7e-01      1.6e-01      8.1e-03   ␣
↪True
3   176      16      2.0e-02    1.5e-01 1.9e-01   1.7e-01      1.6e-01      7.4e-03   ␣
↪True
4   192      16      1.7e-02    1.5e-01 1.8e-01   1.7e-01      1.6e-01      3.7e-03   ␣
↪True
5   208      16      1.5e-02    1.5e-01 1.8e-01   1.7e-01      1.6e-01      4.7e-03   ␣
↪True
6   224      16      1.3e-02    1.5e-01 1.8e-01   1.7e-01      1.6e-01      3.4e-03   ␣
↪True
7   240      16      1.2e-02    1.5e-01 1.8e-01   1.6e-01      1.6e-01      2.3e-03   ␣
↪True
```

(continues on next page)

(continued from previous page)

```
8   256      16      1.1e-02    1.6e-01 1.8e-01   1.7e-01      1.6e-01      3.5e-03   ␣
↪True
```



5.28.5 Hartmann 6d Problem

```
mcispfgp = qp.PFGPCI(
    integrand = qp.Hartmann6d(qp.DigitalNetB2(6,seed=17)),
    failure_threshold = -2,
    failure_above_threshold=False,
    abs_tol = 2.5e-3,
    alpha = 1e-1,
    n_init = 512,
    init_samples = None,
    batch_sampler = qp.PFSampleErrorDensityAR(verbose=True),
    n_batch = 64,
    n_max = 2500,
    n_approx = 2**18,
    gpytorch_prior_mean = gpytorch.means.ZeroMean(),
    gpytorch_prior_cov = gpytorch.kernels.ScaleKernel(gpytorch.kernels.MaternKernel(nu=2.
↪5)),
    gpytorch_likelihood = gpytorch.likelihoods.GaussianLikelihood(noise_constraint =␣
↪gpytorch.constraints.Interval(1e-12,1e-8)),
    gpytorch_marginal_log_likelihood_func = lambda likelihood,gpyt_model: gpytorch.mlls.
↪ExactMarginalLogLikelihood(likelihood,gpyt_model),
    torch_optimizer_func = lambda gpyt_model: torch.optim.Adam(gpyt_model.parameters(),
↪lr=0.1),
    gpytorch_train_iter = 150,
    gpytorch_use_gpu = gpytorch_use_gpu,
    verbose = 50,
    n_ref_approx = 2**23,
    seed_ref_approx = None)
```

(continues on next page)

(continued from previous page)

```
solution,data = mcispfgp.integrate(seed=7,refit=False)
print(data)
df = pd.DataFrame(data.get_results_dict())
print("\nIteration Summary")
print(df)
data.plot();
```

```
reference approximation with d=6: 0.007387995719909668
batch 0
    gpytorch model fitting
            iter 50  of 150
                    likelihood.noise_covar.raw_noise.................. 1.65e+00
                    covar_module.raw_outputscale...................... 5.53e-01
                    covar_module.base_kernel.raw_lengthscale.......... 2.79e-01
            iter 100 of 150
                    likelihood.noise_covar.raw_noise.................. 2.80e+00
                    covar_module.raw_outputscale...................... 5.10e-01
                    covar_module.base_kernel.raw_lengthscale.......... 2.67e-01
            iter 150 of 150
                    likelihood.noise_covar.raw_noise.................. 3.38e+00
                    covar_module.raw_outputscale...................... 5.14e-01
                    covar_module.base_kernel.raw_lengthscale.......... 2.68e-01
batch 1
    AR sampling with efficiency 1.4e-03, expect 45089 draws: 31232, 35624,
batch 2
    AR sampling with efficiency 2.1e-03, expect 30750 draws: 21312, 27306, 29304, 30969,␣
↪31635, 32301,
batch 3
    AR sampling with efficiency 1.8e-03, expect 35103 draws: 24320, 32680, 36860, 38000,
batch 4
    AR sampling with efficiency 1.6e-03, expect 41262 draws: 28608, 40230, 41571,
batch 5
    AR sampling with efficiency 1.4e-03, expect 46939 draws: 32576, 43265, 47846, 50900,
batch 6
    AR sampling with efficiency 1.2e-03, expect 54986 draws: 38144, 51852, 53044, 54236,␣
↪54832,
batch 7
    AR sampling with efficiency 1.0e-03, expect 61209 draws: 42432, 59670, 66300, 68289,␣
↪69615, 70278,
batch 8
    AR sampling with efficiency 9.2e-04, expect 69388 draws: 48128, 63168, 65424, 66176,␣
↪66928, 67680, 68432,
batch 9
    AR sampling with efficiency 8.4e-04, expect 75933 draws: 52672, 66663, 71601, 73247,
batch 10
    AR sampling with efficiency 7.6e-04, expect 84744 draws: 58752, 80784, 89046,
batch 11
    AR sampling with efficiency 6.9e-04, expect 93317 draws: 64704, 93012, 99078, 101100,
↪ 103122, 105144,
batch 12
    AR sampling with efficiency 6.3e-04, expect 101312 draws: 70208, 95439, 103118,␣
↪105312, 107506,
```

(continues on next page)

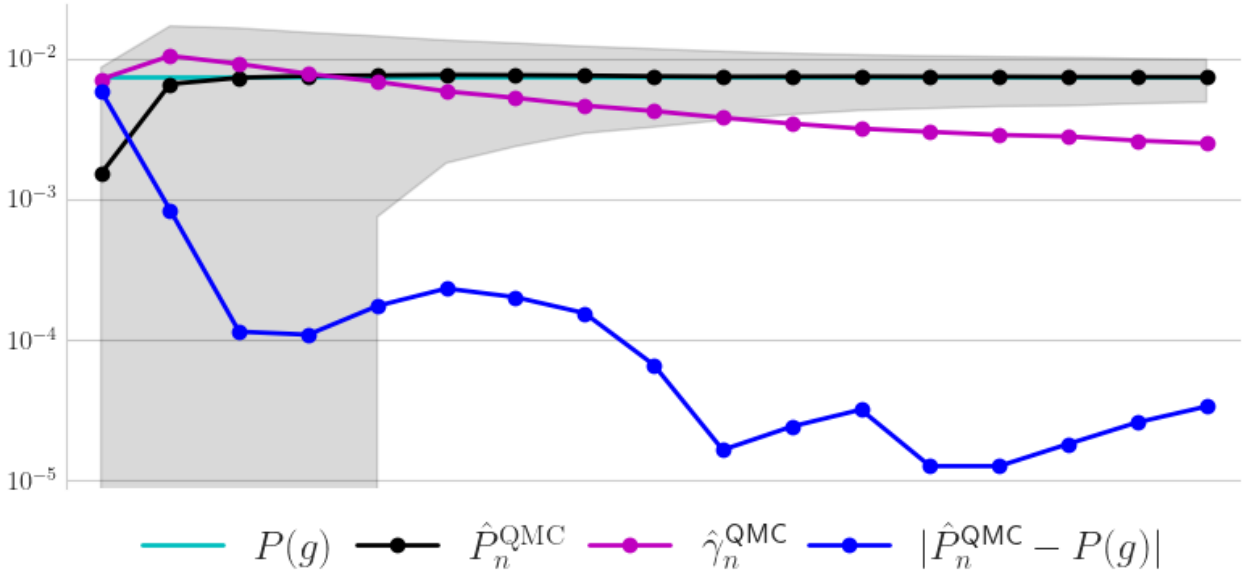
(continued from previous page)

```
batch 13
    AR sampling with efficiency 6.0e-04, expect 106776 draws: 74048, 87932, 92560,
batch 14
    AR sampling with efficiency 5.7e-04, expect 112270 draws: 77824, 98496, 100928,␣
↪102144,
batch 15
    AR sampling with efficiency 5.6e-04, expect 115250 draws: 79872, 113568, 124800,␣
↪126048, 127296, 128544, 129792, 131040, 132288, 133536, 134784,
batch 16
    AR sampling with efficiency 5.2e-04, expect 123202 draws: 85376, 88044, 89378, 90712,
PFGPCIData (AccumulateData Object)
    solution        0.007
    error_bound     0.002
    bound_low       0.005
    bound_high      0.010
    n_total         1536
    time_integrate  250.998
PFGPCI (StoppingCriterion Object)
Hartmann6d (Integrand Object)
Uniform (TrueMeasure Object)
    lower_bound     0
    upper_bound     1
DigitalNetB2 (DiscreteDistribution Object)
    d               6
    dvec            [0 1 2 3 4 5]
    randomize       LMS_DS
    graycode        0
    entropy         17
    spawn_key       ()

Iteration Summary
    n_sum  n_batch  error_bounds  ci_low  ci_high  solutions  solutions_ref  error_ref ␣
↪in_ci
0    512     512      7.1e-03    0.0e+00 8.6e-03   1.5e-03      7.4e-03      5.9e-03   ␣
↪True
1    576      64      1.0e-02    0.0e+00 1.7e-02   6.6e-03      7.4e-03      8.3e-04   ␣
↪True
2    640      64      9.1e-03    0.0e+00 1.6e-02   7.3e-03      7.4e-03      1.1e-04   ␣
↪True
3    704      64      7.8e-03    0.0e+00 1.5e-02   7.5e-03      7.4e-03      1.1e-04   ␣
↪True
4    768      64      6.8e-03    7.4e-04 1.4e-02   7.6e-03      7.4e-03      1.7e-04   ␣
↪True
5    832      64      5.8e-03    1.8e-03 1.3e-02   7.6e-03      7.4e-03      2.3e-04   ␣
↪True
6    896      64      5.2e-03    2.4e-03 1.3e-02   7.6e-03      7.4e-03      2.0e-04   ␣
↪True
7    960      64      4.6e-03    2.9e-03 1.2e-02   7.5e-03      7.4e-03      1.5e-04   ␣
↪True
8   1024      64      4.2e-03    3.2e-03 1.2e-02   7.5e-03      7.4e-03      6.6e-05   ␣
↪True
9   1088      64      3.8e-03    3.6e-03 1.1e-02   7.4e-03      7.4e-03      1.6e-05   ␣
↪True
```

(continues on next page)

(continued from previous page)

```
10  1152      64      3.4e-03    4.0e-03 1.1e-02   7.4e-03      7.4e-03      2.4e-05  ␣
↪True
11  1216      64      3.2e-03    4.3e-03 1.1e-02   7.4e-03      7.4e-03      3.2e-05  ␣
↪True
12  1280      64      3.0e-03    4.4e-03 1.0e-02   7.4e-03      7.4e-03      1.3e-05  ␣
↪True
13  1344      64      2.9e-03    4.6e-03 1.0e-02   7.4e-03      7.4e-03      1.3e-05  ␣
↪True
14  1408      64      2.8e-03    4.6e-03 1.0e-02   7.4e-03      7.4e-03      1.8e-05  ␣
↪True
15  1472      64      2.6e-03    4.8e-03 1.0e-02   7.4e-03      7.4e-03      2.6e-05  ␣
↪True
16  1536      64      2.5e-03    4.9e-03 9.8e-03   7.4e-03      7.4e-03      3.3e-05  ␣
↪True
```



CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

CHAPTER

SEVEN

SPONSORS

7.1 Illinois Tech

7.2 Kamakura Corporation

7.3 SigOpt

PYTHON MODULE INDEX

q

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

P

Q

S

T

U