Metadata of the chapter that will be visualized in SpringerLink

Book Title	Monte Carlo and Quasi-Monte Carlo Methods	
Series Title		
Chapter Title	Quasi-Monte Carlo Software	
Copyright Year	2022	
Copyright HolderName	The Author(s), under exclusive license to Springer Nature Switzerland AG	

Author	Family Name	**Choi**
	Particle	
	Given Name	**Sou-Cheng T.**
	Prefix	
	Suffix	
	Role	
	Division	Department of Applied Mathematics
	Organization	Illinois Institute of Technology
	Address	RE 220, 10 W. 32nd St., Chicago, IL, 60616, USA
	Division	
	Organization	Kamakura Corporation
	Address	2222 Kalakaua Ave, Suite 1400, Honolulu, HI, 96815, USA
	Email	schoi32@iit.edu
Corresponding Author	Family Name	**Hickernell**
	Particle	
	Given Name	**Fred J.**
	Prefix	
	Suffix	
	Role	
	Division	Department of Applied Mathematics
	Organization	Illinois Institute of Technology
	Address	RE 220, 10 W. 32nd St., Chicago, IL, 60616, USA
	Division	Center for Interdisciplinary Scientific Computation and Department of Applied Mathematics
	Organization	Illinois Institute of Technology
	Address	RE 220, 10 W. 32nd St., Chicago, IL, 60616, USA
	Email	hickernell@iit.edu
Author	Family Name	**Jagadeeswaran**
	Particle	
	Given Name	**Rathinavel**
	Prefix	
	Suffix	
	Role	
	Division	Department of Applied Mathematics
	Organization	Illinois Institute of Technology
	Address	RE 220, 10 W. 32nd St., Chicago, IL, 60616, USA

	Division	
	Organization	Wi-Tronix LLC
	Address	631 E Boughton Rd, Suite 240, Bolingbrook, IL, 60440, USA
	Email	jrathin1@iit.edu
Author	Family Name	**McCourt**
	Particle	
	Given Name	**Michael J.**
	Prefix	
	Suffix	
	Role	
	Division	
	Organization	SigOpt, an Intel company
	Address	100 Bush St., Suite 1100, San Francisco, CA, 94104, USA
	Email	mccourt@sigopt.com
Author	Family Name	**Sorokin**
	Particle	
	Given Name	**Aleksei G.**
	Prefix	
	Suffix	
	Role	
	Division	Department of Applied Mathematics
	Organization	Illinois Institute of Technology
	Address	RE 220, 10 W. 32nd St., Chicago, IL, 60616, USA
	Email	asorokin@hawk.iit.edu

Abstract	Practitioners wishing to experience the efficiency gains from using low discrepancy sequences need correct, robust, well-written software. This article, based on our MCQMC 2020 tutorial, describes some of the better quasi-Monte Carlo (QMC) software available. We highlight the key software components required by QMC to approximate multivariate integrals or expectations of functions of vector random variables. We have combined these components in QMCPy, a Python open-source library, which we hope will draw the support of the QMC community. Here we introduce QMCPy.

Keywords (separated by '-')	■■■

Quasi-Monte Carlo Software

Sou-Cheng T. Choi, Fred J. Hickernell, Rathinavel Jagadeeswaran, Michael J. McCourt, and Aleksei G. Sorokin

1 **Abstract** Practitioners wishing to experience the efficiency gains from using low
2 discrepancy sequences need correct, robust, well-written software. This article, based
3 on our MCQMC 2020 tutorial, describes some of the better quasi-Monte Carlo
4 (QMC) software available. We highlight the key software components required by
5 QMC to approximate multivariate integrals or expectations of functions of vector
6 random variables. We have combined these components in QMCPy, a Python open-
7 source library, which we hope will draw the support of the QMC community. Here
8 we introduce QMCPy.

9 **Keywords** ∎∎∎

S.-C. T. Choi · F. J. Hickernell (✉) · R. Jagadeeswaran · A. G. Sorokin
Department of Applied Mathematics, Illinois Institute of Technology, RE 220, 10 W. 32nd St.,
Chicago, IL 60616, USA
e-mail: hickernell@iit.edu

S.-C. T. Choi
e-mail: schoi32@iit.edu

R. Jagadeeswaran
e-mail: jrathin1@iit.edu

A. G. Sorokin
e-mail: asorokin@hawk.iit.edu

S.-C. T. Choi
Kamakura Corporation, 2222 Kalakaua Ave, Suite 1400, Honolulu, HI 96815, USA

F. J. Hickernell
Center for Interdisciplinary Scientific Computation and Department of Applied Mathematics,
Illinois Institute of Technology, RE 220, 10 W. 32nd St., Chicago, IL 60616, USA

R. Jagadeeswaran
Wi-Tronix LLC, 631 E Boughton Rd, Suite 240, Bolingbrook, IL 60440, USA

M. J. McCourt
SigOpt, an Intel company, 100 Bush St., Suite 1100, San Francisco, CA 94104, USA
e-mail: mccourt@sigopt.com

1

1 Introduction

Quasi-Monte Carlo (QMC) methods promise great efficiency gains over independent and identically distributed (IID) Monte Carlo (MC) methods. In some cases, QMC achieves one hundredth of the error of IID MC in the same amount of time (see Fig. 6). Often, these efficiency gains are obtained simply by replacing IID sampling with low discrepancy (LD) sampling, which is the heart of QMC.

Practitioners might wish to test whether QMC would speed up their computation. Access to the best QMC algorithms available would make that easier. Theoreticians or algorithm developers might want to demonstrate their ideas on various use cases to show their practical value.

This tutorial points to some of the best QMC software available. Then we describe QMCPy [6],[1] which is crafted to be a community-owned Python library that combines the best QMC algorithms and interesting use cases from various authors under a common user interface.

The model problem for QMC is approximating a multivariate integral,

$$\mu := \int_{\mathcal{T}} g(\boldsymbol{t})\,\lambda(\boldsymbol{t})\,\mathrm{d}\boldsymbol{t}, \tag{1}$$

where g is the integrand, and λ is a non-negative weight. If λ is a probability distribution (PDF) for the random variable $\boldsymbol{T}$, then μ is the mean of $g(\boldsymbol{T})$. Regardless, we perform a suitable variable transformation to interpret this integral as the mean of a function of a multivariate, standard uniform random variable:

$$\mu = \mathbb{E}[f(\boldsymbol{X})] = \int_{[0,1]^d} f(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}, \qquad \boldsymbol{X} \sim \mathcal{U}[0,1]^d. \tag{2}$$

QMC approximates the population mean, μ, by a sample mean,

$$\widehat{\mu} := \frac{1}{n}\sum_{i=0}^{n-1} f(\boldsymbol{X}_i), \qquad \boldsymbol{X}_0, \boldsymbol{X}_1, \ldots \overset{\mathrm{M}}{\sim} \mathcal{U}[0,1]^d. \tag{3}$$

The choice of the sequence $\{\boldsymbol{X}_i\}_{i=0}^{\infty}$ and the choice of n to satisfy the prescribed error requirement,

$$|\mu - \widehat{\mu}| \leq \varepsilon \qquad \text{absolutely or with high probability,} \tag{4}$$

are important decisions, which QMC software helps the user make.

---

[1] QMCPy is in active development. This article is based on version 1.2 on PyPI.
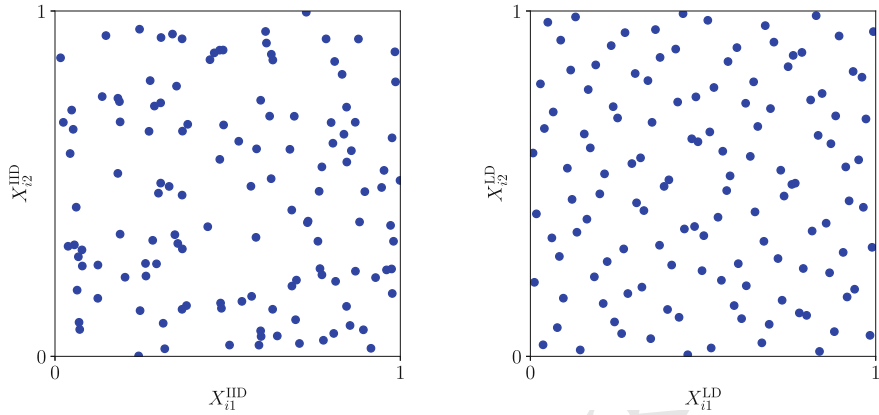
Fig. 1 IID points (left) contrasted with LD points (right). The LD points cover the square more evenly

Here, the notation $\overset{\mathrm{M}}{\sim}$ means that the sequence mimics the specified, target distribution, but not necessarily in a probabilistic way. We use this notation in two forms: $\overset{\mathrm{IID}}{\sim}$ and $\overset{\mathrm{LD}}{\sim}$.

IID sequences must be random. The position of any point is not influenced by any other, so clusters and gaps occur. A randomly chosen subsequence of an IID sequence is also IID. When we say that $X_0, X_1, \ldots \overset{\mathrm{IID}}{\sim} F$ for some distribution F, we mean that for any positive integer n, the multivariate probability distribution of $X_0, \ldots, X_{n-1}$ is the product of the marginals, specifically,

$$F_n(x_0, \ldots, x_{n-1}) = F(x_0) \cdots F(x_{n-1}).$$

When IID points are used to approximate μ by the sample mean, the root mean squared error is $O(n^{-1/2})$. Figure 1 displays IID uniform points, $X_0^{\mathrm{IID}}, X_1^{\mathrm{IID}}, \ldots \overset{\mathrm{IID}}{\sim} \mathcal{U}[0, 1]^2$, i.e., the target distribution is $F_{\mathrm{unif}}(x) = x_1 x_2$.

LD sequences may be deterministic or random, but each point is carefully coordinated with the others so that they fill the domain well. Subsequences of LD sequences are generally not LD. When we say that $X_0, X_1, \ldots \overset{\mathrm{LD}}{\sim} \mathcal{U}[0, 1]^d$, we mean that for any positive integer n, the *empirical distribution* of $X_0, \ldots, X_{n-1}$, denoted $F_{\{X_i\}_{i=0}^{n-1}}$, approximates the uniform distribution, F_{unif}, well (relative to n). (The empirical distribution of a set assigns equal probability to each point.).

A measure of the difference between the empirical distribution of a set of points and the uniform distribution is called a *discrepancy* and is denoted $D(\{X_i\}_{i=0}^{n-1})$ [10, 17, 18, 38]. This is the origin of the term "low discrepancy" points or sequences. LD points by definition have a smaller discrepancy than IID points. Figure 1 contrasts IID uniform points with LD points, $X_0^{\mathrm{LD}}, X_1^{\mathrm{LD}} \ldots \overset{\mathrm{LD}}{\sim} \mathcal{U}[0, 1]^2$, in this case, linearly scrambled and digitally shifted Sobol' points.

$_{61}$ The error in using the sample mean to approximate the integral can be bounded
$_{62}$ according to the Koksma-Hlawka inequality and its extensions [10, 17, 18, 38] as
$_{63}$ the product of the discrepancy of the sampling sequence and the variation of the
$_{64}$ integrand, denoted $V(\cdot)$:

$$_{65} \qquad |\mu - \widehat{\mu}| = \left| \int_{[0,1]^d} f(\boldsymbol{x}) \, \mathrm{d}\big(F_{\mathrm{unif}} - F_{\{X_i\}_{i=0}^{n-1}}\big)(\boldsymbol{x}) \right| \le D\big(\{X_i\}_{i=0}^{n-1}\big) V(f). \qquad (5)$$

$_{66}$ The variation is a (semi-) norm of the integrand in a suitable Banach space. The
$_{67}$ discrepancy corresponds to the norm of the error functional for that Banach space.
$_{68}$ For typical Banach spaces, the discrepancy of LD points is $O(n^{-1+\epsilon})$, a higher con-
$_{69}$ vergence order than for IID points. For details, readers may refer to the references.

$_{70}$ Here, we expect the reader to see in Fig. 1 that the LD points cover the integration
$_{71}$ domain more evenly than IID points. LD sampling can be thought of as a more even
$_{72}$ distribution of the sampling sites than IID. LD sampling is similar to stratified sam-
$_{73}$ pling. In the examples below, the reader will see the demonstrably smaller cubature
$_{74}$ errors arising from using LD points.

$_{75}$ In the sections that follow, we first overview available QMC software. We next
$_{76}$ describe an architecture for good QMC software, i.e., what the key components are
$_{77}$ and how they should interact. We then describe how we have implemented this archi-
$_{78}$ tecture in QMCPy. Finally, we summarize further directions that we hope QMCPy
$_{79}$ and other QMC software projects will take. Those interested in following or con-
$_{80}$ tributing to the development of QMCPy are urged to visit the GitHub repository at
$_{81}$ https://github.com/QMCSoftware/QMCSoftware.

$_{82}$ We have endeavored to be as accurate as possible at the time of writing this article.
$_{83}$ We hope that progress in QMC software development will make this article happily
$_{84}$ obsolete in the coming years.

2 Available Software for QMC

$_{86}$ QMC software spans LD sequence generators, cubatures, and applications. Here we
$_{87}$ review the better-known software, recognizing that some software overlaps multiple
$_{88}$ categories. Whenever applicable, we state each library's accessibility in QMCPy,
$_{89}$ or contrast its functionalities with QMCPy's—where we lag behind in QMCPy, we
$_{90}$ strive to catch up in the near future.

$_{91}$ Software focusing on generating high-quality LD sequences and their generators
$_{92}$ includes the following, listed in alphabetical order:

$_{93}$ **BRODA**   Commerical and non-commercial software developed jointly with I.
$_{94}$ M. Sobol' in C++, MATLAB, and Excel [28]. BRODA can generate Sobol'
$_{95}$ sequences up to 65,536 dimensions. In comparison, QMCPy supports Sobol'
$_{96}$ sequences up to 21,201 dimensions.

97 **Burkhardt** Various QMC software for generating van der Corput, Faure, Halton,
98 Hammersley, Niederreiter, or Sobol' sequences in C, C++, Fortran, MATLAB,
99 or Python [4]. In QMCPy, we have implemented digital net, lattice, and Halton
100 generators.

101 **LatNet Builder** The successor to *Lattice Builder* [34], this is a C++ library with
102 Python and Java interfaces (in SSJ below) for generating vectors or matrices
103 for lattices and digital nets [9, 33]. QMCPy contains a module for parsing the
104 resultant vectors or matrices from LatNet Builder for compatibility with our LD
105 point generators.

106 **MATLAB** Commercial software for scientific computing [51], which contains
107 Sobol' and Halton sequences in the Statistics and Machine Learning Toolbox.
108 Both generators can be applied jointly with the Parallel Computing Toolbox
109 to accelerate their execution speed. The dimension of the Sobol' sequences is
110 restricted to 1,111, which is relatively small, yet sufficient for most applications.

111 **MPS** Magic Point Shop contains lattices and Sobol' sequences in C++, Python,
112 and MATLAB [39]. QMCPy started with MPS for developing LD generators.

113 **Owen** Owen's randomized Halton sequences with dimensions up to 1,000 [43] and
114 scrambled Sobol' sequences with dimensions up to 21,021 [41, 44] in R. QMCPy
115 supports Owen's Halton randomization method, and we plan to implement Owen's
116 nested uniform scrambling for digital nets in the near future.

117 **PyTorch** Open-source Python library for deep learning, with unscrambled or
118 scrambled Sobol' sequences [46, 47]. PyTorch enables seamless utilization of
119 Graphics Processing Units (GPUs) or Field Programmable Gate Arrays (FPGAs).

120 **QMC.jl** LD Sequences in Julia [48]. Julia [3] is an interpreted language similar
121 to Python and R in terms of ease of use, but is designed to run much faster.

122 **qrng** Randomized Sobol', Halton, and Korobov sequences in R [23]. The default
123 Halton randomization in QMCPy utilizes the methods from qrng.

124 **SciPy** Scientific computing library in Python with Latin hypercube, Halton, and
125 Sobol' generators [52].

126 **TF Quant Finance** Google's Tensorflow deep-learning library [1] specialized for
127 financial modeling [12]. It contains lattice and Sobol' generators alongside with
128 algorithms sped up with GPUs, FGPAs, or Tensor Processing Units (TPUs).

129 Software focusing on QMC cubatures and applications includes the following:

130 **GAIL** The Guaranteed Automatic Integration Library contains automatic (Q)MC
131 stopping criteria in MATLAB [5, 14]. These are iterative procedures for one- or
132 high-dimensional integration that take a user's input error tolerance(s) and deter-
133 mine the number of (Q)MC sampling points necessary to achieve user-desired
134 accuracy (almost surely). Most of GAIL's (Q)MC functions, some with enhance-
135 ments, are implemented in Python in QMCPy.

136 **ML(Q)MC** Multi-Level (Q)MC routines in C, C++, MATLAB, Python, and
137 R [11]. We have ported ML(Q)MC functions to QMCPy.

138 **MultilevelEstimators.jl** ML(Q)MC methods in Julia [49]. The author, Pieterjan
139 Robbe, has contributed cubature algorithms and use cases to QMCPy.

140 **OpenTURNS**   Open source initiative for the Treatment of Uncertainties, Risks'N
141   Statistics [40] written in C++ and Python, leveraging R statistical packages, as
142   well as LAPACK and BLAS for numerical linear algebra.
143 **QMC4PDE**   QMC for elliptic PDEs with random diffusion coefficients in
144   Python [30].
145 **SSJ**   Stochastic Simulation with the `hups` package in Java [32].
146 **UQLab**   Framework for Uncertainty Quantification in MATLAB [36]. The core of
147   UQLab is closed source, but a large portion of the library is open source. Recently,
148   UQ[py]Lab, the beta release of UQLab with Python bindings, is available as
149   Software as a Service (SaaS) via UQCloud [31].

150   The sections that follow describe QMCPy [6], which is our attempt to establish a
151 framework for QMC software and to combine the best of the software under a
152 common user interface written in Python 3. The choice of language was determined
153 by the desire to make QMC software accessible to a broad audience, especially the
154 technology industry.

3   Components of QMC Software

156 QMC cubature can be summarized as follows. We want to approximate the expecta-
157 tion, μ, well by the sample mean, $\widehat{\mu}$, where (1), (2), and (3) combine to give

$$\mu := \int_{\mathcal{T}} g(t)\,\lambda(t)\,\mathrm{d}t = \mathbb{E}[f(X)] = \int_{[0,1]^d} f(x)\,\mathrm{d}x \approx \frac{1}{n}\sum_{i=0}^{n-1} f(X_i) =: \widehat{\mu},$$

$$X \sim \mathcal{U}[0,1]^d,\ X_0, X_1, \ldots \overset{\mathrm{M}}{\sim} \mathcal{U}[0,1]^d. \tag{6}$$

161 Moreover, we want to satisfy the error requirement in (4). This requires four com-
162 ponents, which we implement as QMCPy classes.

163 **Discrete Distribution**   produces the sequence $X_0, X_1, \ldots$ that mimics $\mathcal{U}[0,1]^d$;
164 **True Measure** $t \mapsto \lambda(t)\mathrm{d}t$ defines the original measure, e.g., Gaussian or
165   Lebesgue;
166 **Integrand** g defines the original integrand, and f defines the transformed version
167   to fit the `DiscreteDistribution`; and
168 **Stopping Criterion**   determines how large n should be to ensure that $|\mu - \widehat{\mu}| \le \varepsilon$
169   as in (4).

170   The software libraries referenced in Sect. 2 provide one or more of these com-
171 ponents. QMCPy combines multiple examples of all these components under an
172 object-oriented framework. Each example is implemented as a concrete class that
173 realizes the properties and methods required by the abstract class for that component.
174 The following sections detail descriptions and specific examples for each component.

175 Thorough documentation of all QMCPy classes is available in [7]. Demonstrations
176 of how QMCPy works are given in Google Colab notebooks [15, 16]. The project
177 may be installed from PyPI into a Python 3 environment via the command `pip`
178 `install qmcpy`. In the code snippets that follow, we assume QMCPy has been
179 imported alongside NumPy [13] via the following commands in a Python Console:

```
180 >>> import qmcpy as qp
181 >>> import numpy as np
```

4 Discrete Distributions

183 LD sequences typically mimic $\mathcal{U}[0, 1]^d$. Good sequences mimicking other distribu-
184 tions are obtained by transformations as described in the next section. We denote by
185 `DiscreteDistribution` the abstract class containing LD sequence generators.
186 In most cases that have been implemented, the points $X_0, \ldots, X_{n-1}$ have an empirical
187 (discrete) distribution that closely approximates the uniform distribution, say, in the
188 sense of discrepancy. We also envision future possible `DiscreteDistribution`
189 objects that assign unequal weights to the sampling points. In any case, the term
190 "discrete" refers to the fact that these are sequences of points (and weights), not
191 continuous distributions.

192 QMCPy implements *extensible* LD sequences, i.e., those that allow practitioners
193 to obtain and use $X_n, X_{n+1}, \ldots$ without discarding $X_0, \ldots, X_{n-1}$. Halton sequences
194 do not have preferred sample sizes n, but extensible integration lattices and digital
195 sequences in base b prefer n to be a power of b. For integration lattices and digital
196 sequences, we have focused on base $b = 2$ since this is a popular choice and for
197 convenience in generating extensible sequences.

198 Integration lattices and digital sequences in base 2 have an elegant group structure,
199 which we summarize in Table 1. The addition operator[2] is $\oplus$, and its inverse is $\ominus$. The
200 unshifted sequence is $Z_0, Z_1, \ldots$ and the randomly shifted sequence is $X_0, X_1, \ldots$
201 We illustrate lattice and Sobol' sequences using QMCPy. First, we create an
202 instance of a $d = 2$ dimensional `Lattice` object of the `DiscreteDistribution`
203 abstract class. Then we generate the first eight (non-randomized) points in this
204 lattice.

---

[2] The operator $\oplus$ is commonly used to denote exclusive-or, which is its meaning for digital sequences in base 2. However, we are using it here in a more general sense.

Table 1 Properties of lattices and digital net sequences. Note that they share group properties but also have distinctives

Define …
$\mathbf{Z}_1, \mathbf{Z}_2, \mathbf{Z}_4, \ldots \in [0, 1)^d$ chosen well
$\mathbf{Z}_i := i_0 \mathbf{Z}_1 \oplus i_1 \mathbf{Z}_2 \oplus i_2 \mathbf{Z}_4 \oplus i_3 \mathbf{Z}_8 \oplus \cdots \quad$ for $i = i_0 + i_1 2 + i_2 4 + i_3 8 + \cdots, \ i_\ell \in \{0, 1\}$
$\mathbf{X}_i := \mathbf{Z}_i \oplus \mathbf{\Delta}, \qquad$ where $\mathbf{\Delta} \overset{\text{IID}}{\sim} [0, 1)^d$

Rank-1 Integration Lattices	Digital Nets
$t \oplus x := (t + x) \bmod \mathbf{1}$	$t \oplus x :=$ binary digitwise addition, $\oplus_{\text{dig}}$
require $\mathbf{Z}_{2^m} \oplus \mathbf{Z}_{2^m} = \mathbf{Z}_{\lfloor 2^{m-1} \rfloor} \quad \forall m \in \mathbb{N}_0$	

Then it follows that …
$\mathcal{P}_m := \{\mathbf{Z}_0, \ldots, \mathbf{Z}_{2^m - 1}\}, \quad \mathbf{Z}_i \oplus \mathbf{Z}_j \in \mathcal{P}_m \left.\right\} \quad \forall i, j, k \in \{0, \ldots, 2^m - 1\}$
$\mathcal{P}_{\mathbf{\Delta},m} := \{\mathbf{X}_0, \ldots, \mathbf{X}_{2^m - 1}\}, \quad \mathbf{X}_i \oplus \mathbf{X}_j \ominus \mathbf{X}_k \in \mathcal{P}_{\mathbf{\Delta},m} \left.\right\} \quad \forall m \in \mathbb{N}_0$

```
205  >>> lattice = qp.Lattice(dimension=2, randomize=False)
206  >>> lattice.gen_samples(n=8)
207  "ParameterWarning:
208    Non-randomized lattice sequence includes the origin"
209  array([[0.  , 0.  ],
210         [0.5 , 0.5 ],
211         [0.25 , 0.75 ],
212         [0.75 , 0.25 ],
213         [0.125, 0.375],
214         [0.625, 0.875],
215         [0.375, 0.125],
216         [0.875, 0.625]])
```

The first three generators for this lattice are $\mathbf{Z}_1 = (0.5, 0.5)$, $\mathbf{Z}_2 = (0.25, 0.75)$, and $\mathbf{Z}_4 = (0.125, 0.375)$. One can check that $(\mathbf{Z}_2 + \mathbf{Z}_4) \bmod \mathbf{1} = (0.375, 0.125) = \mathbf{Z}_6$, as Table 1 specifies.

The random shift has been turned off above to illuminate the group structure. We normally include the randomization to ensure that there are no points on the boundary of $[0, 1]^d$. Then, when points are transformed to mimic distributions such as the Gaussian, no LD points will be transformed to infinity. Turning off the randomization generates a warning when the gen_samples method is called.

Now, we generate Sobol' points using a similar process as we did for lattice points. Sobol' sequences are one of the most popular example of digital sequences.

```
227  >>> sobol = qp.Sobol(2, randomize=False)
228  >>> sobol.gen_samples(8, warn=False)
229  array([[0.  , 0.  ],
230         [0.5 , 0.5 ],
231         [0.25 , 0.75 ],
232         [0.75 , 0.25 ],
233         [0.125, 0.625],
234         [0.625, 0.125],
235         [0.375, 0.375],
236         [0.875, 0.875]])
```

237 Here, Z_4 differs from that for lattices, but more importantly, addition for digital
238 sequences differs from that for lattices. Using digitwise addition for digital sequences,
239 we can confirm that according to Table 1,

$$Z_2 \oplus_{\text{dig}} Z_4 = (0.25, 0.75) \oplus_{\text{dig}} (0.125, 0.625)$$

$$= (_20.010, {}_20.110) \oplus_{\text{dig}} (_20.001, {}_20.101) = (_20.011, {}_20.011)$$

$$= (0.375, 0.375) = Z_6.$$

244 By contrast, if we construct a digital sequence using the generators for the lattice
245 above with $Z_2 = (0.25, 0.75)$, and $Z_4 = (0.125, 0.375)$, we would obtain

$$Z_6 = Z_2 \oplus_{\text{dig}} Z_4 = (_20.010, {}_20.110) \oplus_{\text{dig}} (_20.001, {}_20.011)$$

$$= (_20.011, {}_20.101) = (0.375, 0.625),$$

249 which differs from the $Z_6 = (0.375, 0.125)$ constructed for lattices. To emphasize,
250 lattices and digital sequences are different, even if they share the same generators,
251 $Z_1, Z_2, Z_4, \ldots$.

252 The examples of `qp.Lattice` and `qp.Sobol` illustrate how QMCPy LD gen-
253 erators share a common user interface. The dimension is specified when the instance is
254 constructed, and the number of points is specified when the `gen_samples` method
255 is called. Following Python practice, parameters can be input without specifying
256 their names if input in the prescribed order. QMCPy also includes Halton sequences
257 and IID sequences, again deferring details to the QMCPy documentation [7].

258 A crucial difference between IID generators and LD generators is reflected in the
259 behavior of generating n points. For an IID generator, asking for n points repeatedly
260 gives different points each time because they are meant to be random and indepen-
261 dent.

```
262 >>> iid = qp.IIDStdUniform(2); iid.gen_samples(1)
263 array([[0.40538109, 0.12255759]])
264 >>> iid.gen_samples(1)
265 array([[0.50913741, 0.11312201]])
```

266 Your output may look different depending on the seed used to generate these random
267 numbers.

268 On the other hand, for an LD generator, asking for n points repeatedly gives *the
269 same* points each time because they are meant to be the first n points of a specific
270 LD sequence.

```
271 >>> lattice = qp.Lattice(2); lattice.gen_samples(1)
272 array([[0.2827584 , 0.36731649]])
273 >>> lattice.gen_samples(1)
274 array([[0.2827584 , 0.36731649]])
```

275 Here we allow the randomization so that the first point in the sequence is not the
276 origin. To obtain the *next n* points, one may specify the start and ending indices of
277 the sequence.
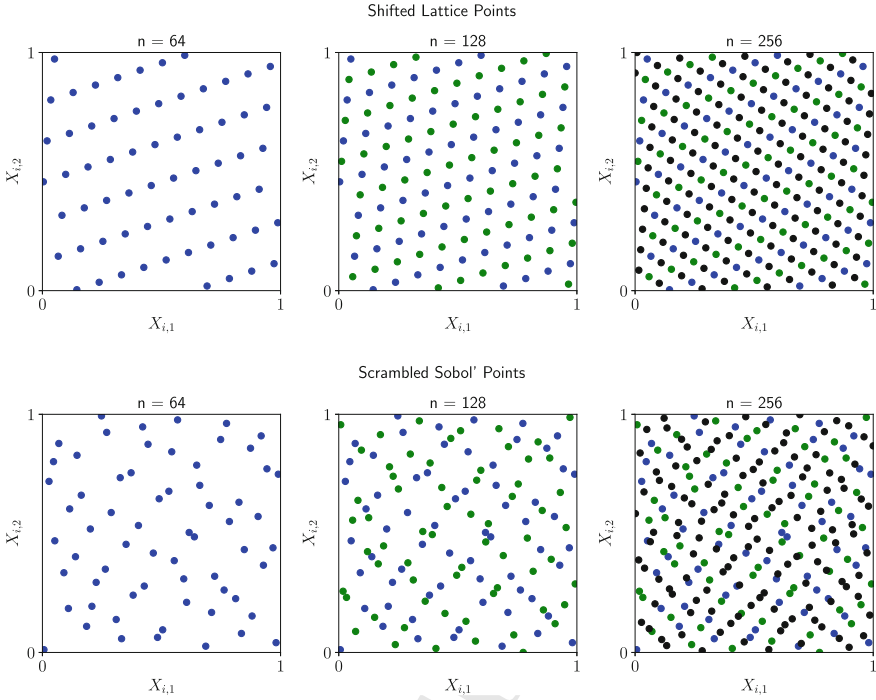
Fig. 2 Randomized lattice and Sobol' points mimicking a $\mathcal{U}[0, 1]^2$ measure for $n = 64$, 128, and 256. Note how increasing the number of points evenly fills in the gaps between the points

```
278  >>> lattice.gen_samples(2)
279  array([[0.2827584 , 0.36731649],
280         [0.7827584 , 0.86731649]])
281  >>> lattice.gen_samples(n_min=1, n_max=2)
282  array([[0.7827584 , 0.86731649]])
```

283   Figure 2 shows how increasing the number of lattice and Sobol' LD points through
284 powers of two fills in the gaps in an even way.

5 True Measures

286 The LD sequences implemented as `DiscreteDistribution` objects mimic the
287 $\mathcal{U}[0, 1]^d$ distribution. However, we may need sequences to mimic other distributions.
288 This is implemented via variable transformations, $\boldsymbol{\Psi}$. In general, if $X \overset{\text{M}}{\sim} \mathcal{U}[0, 1]^d$,
289 then

$$\boldsymbol{T} = \boldsymbol{\Psi}(X) := \boldsymbol{a} + (\boldsymbol{b} - \boldsymbol{a}) \odot X \overset{\mathrm{M}}{\sim} \mathcal{U}[\boldsymbol{a}, \boldsymbol{b}], \tag{7a}$$

$$\boldsymbol{T} = \boldsymbol{\Psi}(X) := \boldsymbol{a} + \mathsf{A}\boldsymbol{\Phi}^{-1}(X) \overset{\mathrm{M}}{\sim} \mathcal{N}(\boldsymbol{a}, \Sigma), \tag{7b}$$

$$\text{where } \boldsymbol{\Phi}^{-1}(X) := \begin{pmatrix} \Phi^{-1}(X_1) \\ \vdots \\ \Phi^{-1}(X_d) \end{pmatrix}, \qquad \Sigma = \mathsf{A}\mathsf{A}^T,$$

and $\odot$ denotes term-by-term (Hadamard) multiplication. Here, $\boldsymbol{a}$ and $\boldsymbol{b}$ are assumed to be finite, and Φ is the standard Gaussian distribution function. Again we use $\overset{\mathrm{M}}{\sim}$ to denote mimicry, not necessarily in a probabilistic sense.

Figure 3 displays LD sequences transformed as described above to mimic a uniform and a Gaussian distribution. The code to generate these points takes the following form for uniform points based on a Halton sequence:

```
>>> u = qp.Uniform(
...     sampler = qp.Halton(2),
...     lower_bound = [-2, 0],
...     upper_bound = [2, 4])
>>> u.gen_samples(4)
array([[ 1.80379772, 3.51293599],
       [-0.19620228, 0.84626932],
       [ 0.80379772, 2.17960265],
       [-1.19620228, 3.95738043]])
```

whereas for Gaussian points based on a lattice sequence, we have:

```
>>> g = qp.Gaussian(qp.Lattice(2),
...     mean = [3, 2],
...     covariance = [[9, 5],
...     [5, 4]])
>>> g.gen_samples(4)
array([[-0.00920667, 0.29392389],
       [ 5.02422481, 1.45408341],
       [ 6.53688109, 5.14785917],
       [ 2.58403124, 1.16941969]])
```

Here the covariance decomposition $\Sigma = \mathsf{A}\mathsf{A}^T$ is done using principal component analysis. The Cholesky decomposition is also available.

The Brownian motion distribution arises often in financial risk applications. Here the d components of the variable $\boldsymbol{T}$ correspond to the discretized Brownian motion at times $\tau/d, 2\tau/d, \ldots, \tau$, where τ is the time horizon. The distribution is a special case of the Gaussian with covariance

$$\Sigma = (\tau/d)\big(\min(j, k)\big)_{j,k=1}^d \tag{8}$$

and mean $\boldsymbol{a}$, which is proportional to the times $(\tau/d)(1, 2, \ldots, d)^T$. The code for generating a Brownian motion is
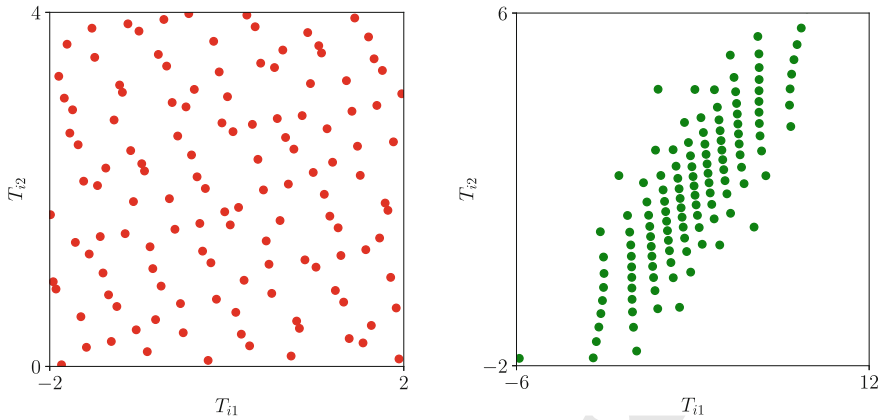
Fig. 3 Halton samples transformed to mimic a uniform $\mathcal{U}\left(\begin{bmatrix}\begin{pmatrix}-2\\0\end{pmatrix}\end{bmatrix}, \begin{bmatrix}\begin{pmatrix}2\\4\end{pmatrix}\end{bmatrix}\right)$ distribution (left) and lattice samples transformed to mimic a Gaussian $\mathcal{N}\left(\begin{bmatrix}\begin{pmatrix}3\\2\end{pmatrix}\end{bmatrix}, \begin{bmatrix}\begin{pmatrix}9&5\\5&4\end{pmatrix}\end{bmatrix}\right)$ distribution (right)
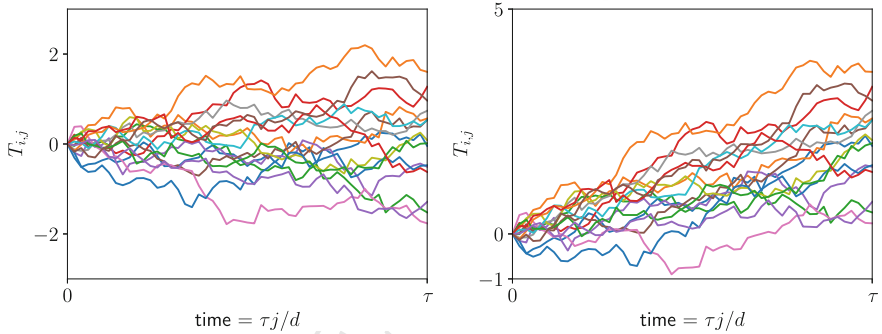
Fig. 4 Sobol' samples transformed to mimic a 52-dimensional Brownian Motion without drift (left) and with drift coefficient 2 (right)

```
328  >>> bm = qp.BrownianMotion(qp.Sobol(4), drift=2)
329  >>> bm.gen_samples(2)
330  array([[-0.28902829, 0.48582198, 1.81113976, 2.2376372 ],
331         [ 1.41552651, 1.93926124, 1.27619672, 1.47496128]])
```

Figure 4 displays a Brownian motion based on Sobol' sequence with and without a drift.

6 Integrands

Let's return to the integration problem in (1), which we must rewrite as (2). We choose a transformation of variables defined as $\boldsymbol{t} = \boldsymbol{\Psi}(\boldsymbol{x})$ where $\boldsymbol{\Psi} : [0,1]^d \to \mathcal{T}$. This leads to

$$338 \qquad \mu = \int_{\mathcal{T}} g(\boldsymbol{t})\,\lambda(\boldsymbol{t})\,\mathrm{d}\boldsymbol{t} = \int_{[0,1]^d} g\big(\boldsymbol{\Psi}(\boldsymbol{x})\big)\,\lambda\big(\boldsymbol{\Psi}(\boldsymbol{x})\big)\,|\boldsymbol{\Psi}'(\boldsymbol{x})|\,\mathrm{d}\boldsymbol{x} = \int_{[0,1]^d} f(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x},$$

$$339 \atop 340 \qquad\qquad \text{where } f(\boldsymbol{x}) = g\big(\boldsymbol{\Psi}(\boldsymbol{x})\big)\,\lambda\big(\boldsymbol{\Psi}(\boldsymbol{x})\big)\,|\boldsymbol{\Psi}'(\boldsymbol{x})|, \qquad\qquad (9)$$

341 and $|\boldsymbol{\Psi}'(\boldsymbol{x})| := |\partial\boldsymbol{\Psi}/\partial\boldsymbol{x}|$ represents the Jacobian of the variable transformation.
342 The abstract class `Integrand` provides f based on the user's input of g and
343 the `TrueMeasure` instance, which defines λ and the transformation $\boldsymbol{\Psi}$. Differ-
344 ent choices of $\boldsymbol{\Psi}$ lead to different f, which may give different rates of convergence
345 of the cubature, $\widehat{\mu}$ to μ.

346   We illustrate the `Integrand` class via an example of Keister [27]:

$$347 \qquad \mu = \int_{\mathbb{R}^d} \cos(\|\boldsymbol{t}\|)\exp(-\boldsymbol{t}^T\boldsymbol{t})\,\mathrm{d}\boldsymbol{t} = \int_{\mathbb{R}^d} \underbrace{\pi^{d/2}\cos(\|\boldsymbol{t}\|)}_{g(\boldsymbol{t})}\,\underbrace{\pi^{-d/2}\exp(-\boldsymbol{t}^T\boldsymbol{t})}_{\lambda(\boldsymbol{t})}\,\mathrm{d}\boldsymbol{t}. \qquad (10)$$

348 Since λ is the density for $\mathcal{N}(\boldsymbol{0}, \mathsf{I}/2)$, it is natural to choose $\boldsymbol{\Psi}$ according to (7b) with
349 $\mathsf{A} = \sqrt{1/2}\,\mathsf{I}$, in which case $\lambda(\boldsymbol{\Psi}(\boldsymbol{x}))|\boldsymbol{\Psi}'(\boldsymbol{x})| = 1$, and so

$$350 \qquad \mu = \int_{[0,1]^d} \underbrace{\pi^{d/2}\cos(\|\boldsymbol{\Psi}(\boldsymbol{x})\|)}_{f(\boldsymbol{x})}\,\mathrm{d}\boldsymbol{x}, \qquad \boldsymbol{\Psi}(\boldsymbol{x}) := \sqrt{1/2}\,\Phi^{-1}(\boldsymbol{x}).$$

351   The code below sets up an `Integrand` instance using QMCPy's `CustomFun`
352 wrapper to tie a user-defined function g into the QMCPy framework. Then we eval-
353 uate the sample mean of $n = 1000$ f values obtained by sampling at transformed
354 Halton points. Notice how a two-dimensional Halton generator is used to construct
355 a Gaussian true measure, which is applied alongside the `my_Keister` function to
356 instantiate a customized, QMCPy-compatible integrand for this problem.

```
357 >>> def my_Keister(t):
358 ...  d = t.shape[1] # t is an (n x d) array
359 ...  norm = np.sqrt((t**2).sum(1))
360 ...  out = np.pi**(d/2)*np.cos(norm)
361 ...  return out # size n vector
362 ...
363 >>> gauss = qp.Gaussian(qp.Halton(2), covariance=1/2)
364 >>> keister = qp.CustomFun(true_measure=gauss, g=my_Keister)
365 >>> x = keister.discrete_distrib.gen_samples(1000)
366 >>> y = keister.f(x)
367 >>> y.mean()
368 1.809055768468628
```

369 We have no indication yet of how accurate our approximation is. That topic is treated
370 in the next section. Figure 5 visualizes sampling on the original integrand, g, and
371 sampling on the transformed integrand, f.
372   Another way to approximate the Keister integral in (10) is to write it as an integral
373 with respect to the Lebesgue measure:
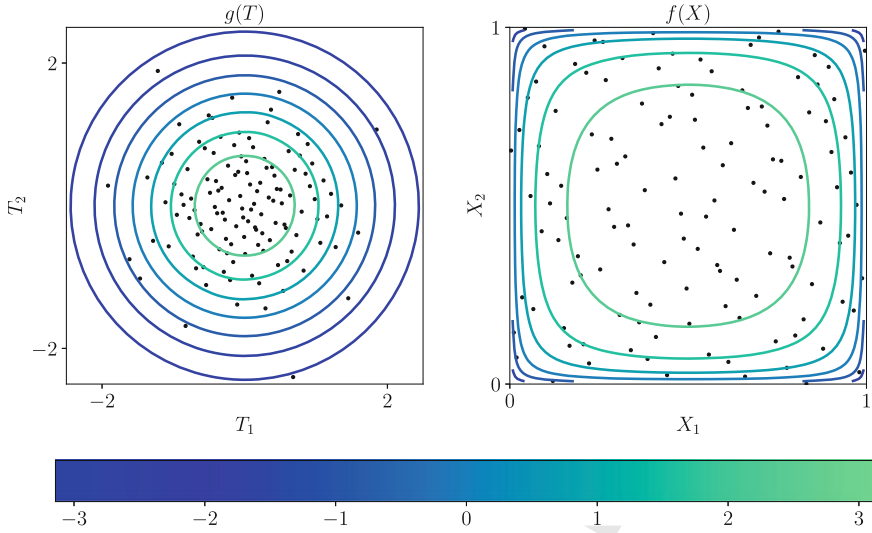
Fig. 5 Right: Sampling the transformed Keister integrand f at Halton points $X_i \overset{\text{LD}}{\sim} \mathcal{U}[0, 1]^2$. Left: Sampling the original Keister integrand g at $T_i = \Psi(X_i) \overset{\text{M}}{\sim} \mathcal{N}(\mathbf{0}, \mathsf{I}/2)$ where Ψ is defined in (7b)

$$\mu = \int\limits_{\mathbb{R}^d} \underbrace{\cos(\|\boldsymbol{t}\|) \exp(-\boldsymbol{t}^T \boldsymbol{t})}_{g(\boldsymbol{t})} \underbrace{1}_{\lambda(\boldsymbol{t})} \, \mathrm{d}\boldsymbol{t}$$

$$= \int\limits_{[0,1]^d} \underbrace{\cos(\|\Psi(\boldsymbol{x})\|) \exp(-\Psi^T(\boldsymbol{x})\Psi(\boldsymbol{x}))|\Psi'(\boldsymbol{x})|}_{f(\boldsymbol{x})} \, \mathrm{d}\boldsymbol{x},$$

where Ψ is any transformation from $[0, 1]^d$ to $\mathbb{R}^d$. Now λ is *not* a PDF. QMCPy can perform the cubature this way as well.

```
>>> def my_L_Keister(t):
...     norm_sq = (t**2).sum(1)
...     out = np.cos(np.sqrt(norm_sq))*np.exp(-norm_sq)
...     return out
...
>>> lebesgue_gauss = qp.Lebesgue(qp.Gaussian(qp.Halton(2)))
>>> keister = qp.CustomFun(lebesgue_gauss, my_L_Keister)
>>> x = keister.discrete_distrib.gen_samples(1000)
>>> y = keister.f(x)
>>> y.mean()
1.8056340581961572
```

The Ψ chosen when transforming uniform sequences on the unit cube to fill $\mathbb{R}^d$ is given by (7b) with $\mathsf{A} = \mathsf{I}$.

In the examples above, one must input the correct g into CustomFun along with the correct TrueMeasure λ to define the integration problem. The Keister

394 integrand included in the QMCPy library takes a more flexible approach to defin-
395 ing the integration problem μ in (10). Selecting a different `sampler` Ψ performs
396 *importance sampling*, which leaves μ unchanged.

```
397 >>> # default transform
398 >>> keister = qp.Keister(qp.Halton(2))
399 >>> x = keister.discrete_distrib.gen_samples(1e4)
400 >>> keister.f(x).mean()
401 1.8082377673556123
402 >>> # custom transform for importance sampling
403 >>> keister = qp.Keister(sampler=qp.Gaussian(qp.Halton(2)))
404 >>> x = keister.discrete_distrib.gen_samples(1e4)
405 >>> keister.f(x).mean()
406 1.8080555069060817
```

407   In the first case above, the λ in (9) corresponds to the Gaussian density with mean
408 zero and variance $1/2$ by default, and the corresponding variable transformation, Ψ,
409 is chosen to make $\lambda\big(\Psi(x)\big)|\Psi'(x)| = 1$ and $f(x) = g\left(\Psi(x)\right)$. In the second case,
410 we choose an importance sampling density λ_{IS}, corresponding to standard Gaussian,
411 and the variable transformation Ψ_{IS} makes $\lambda_{\text{IS}}\big(\Psi_{\text{IS}}(x)\big)|\Psi'_{\text{IS}}(x)| = 1$. Then

$$
\mu = \int_{\mathcal{T}} g(t)\,\lambda(t)\,\mathrm{d}t = \int_{\mathcal{T}} g(t)\,\frac{\lambda(t)}{\lambda_{\text{IS}}(t)}\,\lambda_{\text{IS}}(t)\mathrm{d}t
$$

$$
= \int_{[0,1]^d} g\big(\Psi_{\text{IS}}(x)\big)\,\frac{\lambda\big(\Psi_{\text{IS}}(x)\big)}{\lambda_{\text{IS}}\big(\Psi_{\text{IS}}(x)\big)}\lambda_{\text{IS}}\big(\Psi_{\text{IS}}(x)\big)\,|\Psi'_{\text{IS}}(x)|\,\mathrm{d}x
$$

$$
= \int_{[0,1]^d} f_{\text{IS}}(x)\,\mathrm{d}x
$$

$$
\text{where } f_{\text{IS}}(x) = g\big(\Psi_{\text{IS}}(x)\big)\,\frac{\lambda\big(\Psi_{\text{IS}}(x)\big)}{\lambda_{\text{IS}}\big(\Psi_{\text{IS}}(x)\big)}. \tag{11}
$$

417 Because LD samples mimic $\mathcal{U}[0,1]^d$, choosing a different `sampler` is equivalent
418 to choosing a different variable transform.

7 Stopping Criteria

420 The `StoppingCriterion` object determines the number of samples n that are
421 required for the sample mean approximation $\widehat{\mu}$ to be within error tolerance ε of the
422 true mean μ. Several QMC stopping criteria have been implemented in QMCPy,
423 including replications, stopping criteria that track the decay of the Fourier complex
424 exponential or Walsh coefficients of the integrand [20, 21, 26], and stopping criteria
425 based on Bayesian credible intervals [24, 25].

426   The CubQMCSobolG stopping criterion used in the example below assumes the
427 Walsh-Fourier coefficients of the integrand are absolutely convergent. The algorithm
428 iteratively doubles the number of samples used in the integration and estimates the
429 error using the decay of the Walsh coefficients [20]. When the estimated error is below
430 the user-specified tolerance, it finishes the computation and returns the estimated
431 integral.
432   Let us return to the Keister example from the previous section. After setting up a
433 default Keister instance via a Sobol' DiscreteDistribution, we choose a
434 StoppingCriterion object that matches the DiscreteDistribution and
435 input our desired tolerance. Calling the integrate method returns the approximate
436 integral plus some useful information about the computation.

```
437  >>> keister = qp.Keister(qp.Sobol(2))
438  >>> stopping = qp.CubQMCSobolG(keister, abs_tol=1e-3)
439  >>> solution_qmc,data_qmc = stopping.integrate()
440  >>> data_qmc # equivalent to print(data_qmc)
441  LDTransformData (AccumulateData Object)
442      solution 1.808
443      error_bound 6.06e-04
444      n_total 2^(13)
445      time_integrate 0.008
446  CubQMCSobolG (StoppingCriterion Object)
447      abs_tol 0.001
448      rel_tol 0
449      n_init 2^(10)
450      n_max 2^(35)
451  Keister (Integrand Object)
452  Gaussian (TrueMeasure Object)
453      mean 0
454      covariance 2^(-1)
455      decomp_type PCA
456  Sobol (DiscreteDistribution Object)
457      d 2^(1)
458      dvec [0 1]
459      randomize LMS_DS
460      graycode 0
461      entropy 326942311248945520670220938885737472885
462      spawn_key ()
```

463 The second output of the stopping criterion provides helpful diagnostic information.
464 This computation requires $n = 2^{13}$ Sobol' points and 0.008 seconds to complete.
465 The error bound is 0.000606, which falls below the absolute tolerance.

466   QMC, which uses LD sequences, is touted as providing substantially greater
467 computational efficiency compared to IID MC. Figure 6 compares the time and
468 sample sizes needed to compute the 5-dimensional Keister integral (10) using IID
469 sequences and LD lattice sequences. Consistent with what is stated in Section 1, the
470 error of IID MC is $O(n^{-1/2})$, which means that the time and sample size to obtain
471 an absolute error tolerance of ε is $O(\varepsilon^{-2})$. By contrast, the error of QMC using LD
472 sequences is $O(n^{-1+\epsilon})$, which implies $O(\varepsilon^{-1-\epsilon})$ times and sample sizes. We see that
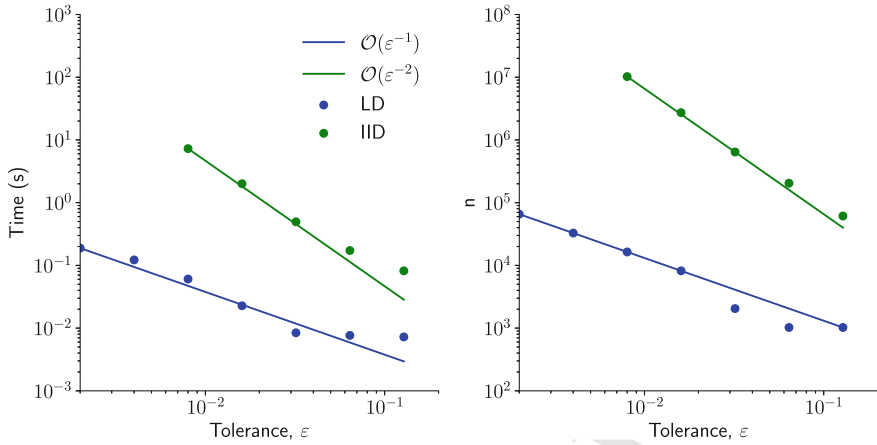
Fig. 6 Comparison of run times and sample sizes for computing the 5-dimensional Keister integral (10) using IID and LD lattice sequences for a variety of absolute error tolerances. The respective stopping criteria are qp.CubMCG [19] and qp.CubQMCLatticeG [26]. The LD sequences provide the desired answer much more efficiently

QMC methods often require orders of magnitude fewer samples than MC methods to achieve the same error tolerance.

For another illustration of QMC cubature, we turn to pricing an Asian arithmetic mean call option. The (continuous-time) Asian option is defined in terms of the average of the stock price, which is written in terms of an integral. The payoff of this option is the positive difference between the strike price, K, averaged over the time horizon:

$$\text{payoff}(\boldsymbol{S}) = \max\left(\frac{1}{2d}\sum_{j=1}^{d}(S_{j-1} + S_j) - K,\ 0\right), \quad \boldsymbol{S} = (S_0, \ldots, S_d).$$

Here S_j denotes the asset price at time $\tau j/d$, and a trapezoidal rule is used for discrete approximation of the integral in time that defines the average. The trapezoidal rule is a more accurate approximation to the integral than a rectangle rule. A basic model for asset prices is a geometric Brownian motion,

$$S_j(\boldsymbol{T}) = S_0 \exp((r - \sigma^2/2)\tau j/d + \sigma T_j), \ j = 1, \ldots, d, \ \boldsymbol{T} = (T_1, \ldots, T_d) \sim \mathcal{N}(\boldsymbol{0}, \Sigma),$$

where Σ is defined in (8), r is the interest rate, σ is the volatility, and S_0 is the initial asset price. The fair price of the option is then the expected value of the discounted payoff, namely,

$$\text{price} = \mu = \mathbb{E}[g(\boldsymbol{T})], \quad \text{where } g(\boldsymbol{t}) = \text{payoff}(\boldsymbol{S}(\boldsymbol{t})) \exp(-r\tau).$$

484   The following code utilizes QMCPy's Asian option `Integrand` object to
485 approximate the value of an Asian call option for a particular choice of the parame-
486 ters.

```
487 >>> payoff = qp.AsianOption(
488 ... qp.Sobol(52), # weekly monitoring
489 ... start_price = 100,
490 ... strike_price = 120,
491 ... volatility = 0.5,
492 ... interest_rate = 0,
493 ... t_final = 1,
494 ... call_put = "call")
495 >>> qmc_stop_crit = qp.CubQMCSobolG(payoff, abs_tol=0.001)
496 >>> price,data = qmc_stop_crit.integrate()
497 >>> print("Option price = $%.3f using %.3f seconds, %.2e samples"
498 ... %(price, data.time_integrate, data.n_total))
499 Option price = $5.194 using 0.587 seconds, 1.31e+05 samples
```

500 Because this `Integrand` object has the built-in Brownian motion `TrueMeasure`,
501 one only need provide the LD sampler.
502   Out of the money option price calculations can be sped up by adding an upward
503 drift to the Brownian motion. The upward drift produces more in the money paths
504 and also reduces the variation or variance of the final integrand, f. This is a form of
505 importance sampling. Using a Brownian motion without drift we get

```
506 >>> payoff = qp.AsianOption(qp.Sobol(52),
507 ... start_price=100, strike_price=200)
508 >>> qmc_stopper = qp.CubQMCSobolG(payoff, abs_tol=0.001)
509 >>> price,data = qmc_stopper.integrate()
510 >>> print("Option price = $%.4f using %.3f seconds, n = %.2e"
511 ... %(price, data.time_integrate, data.n_total))
512 Option price = $0.1757 using 0.583 seconds, n = 1.31e+05
```

513 Adding the upward drift gives us the answer faster:

```
514 >>> payoff = qp.AsianOption(
515 ... sampler = qp.BrownianMotion(qp.Sobol(52), drift=1),
516 ... start_price=100, strike_price=200)
517 >>> qmc_stopper = qp.CubQMCSobolG(payoff, abs_tol=0.001)
518 >>> price,data_drift = qmc_stopper.integrate()
519 >>> print("Option price = $%.4f using %.3f seconds, n = %.2e"
520 ... %(price, data_drift.time_integrate, data_drift.n_total))
521 Option price = $0.1754 using 0.085 seconds, n = 1.64e+04
522 >>> print("Using drift required %.0f%% the time, %.0f%% the n"
523 ... %(100*data_drift.time_integrate / data.time_integrate,
524 ... 100*data_drift.n_total / data.n_total))
525 Using drift required 15% the time, 12% the n
```

526 The choice of a good drift is an art.
527   The improvement in time is less than that in n because the integrand is more
528 expensive to compute when the drift is employed. Referring to (11), in the case of
529 no drift, the λ corresponds to the density for the discrete Brownian motion, and

530 the variable transformation $\boldsymbol{\Psi}$ is chosen so that $f(\boldsymbol{x}) = g(\boldsymbol{\Psi}(\boldsymbol{x}))$. However, in the
531 case of a drift, the integrand becomes $f_{\mathrm{IS}}(\boldsymbol{x}) = g(\boldsymbol{\Psi}_{\mathrm{IS}}(\boldsymbol{x}))\lambda(\boldsymbol{\Psi}_{\mathrm{IS}}(\boldsymbol{x}))/\lambda_{\mathrm{IS}}(\boldsymbol{\Psi}_{\mathrm{IS}}(\boldsymbol{x}))$,
532 which requires more computation time per integrand value.

8 Under the Hood

534 In this section, we look at the inner workings of QMCPy and point out features
535 we hope will benefit the community of QMC researchers and practitioners. We also
536 highlight important nuances of QMC methods and how QMCPy addresses these
537 challenges. For details, readers should refer to the QMCPy documentation [7].

8.1 LD Sequences

539 LD sequences are the backbone of QMC methods. QMCPy provides generators that
540 combine research from across the QMC community to enable advanced features and
541 customization options.

542 Two popular LD sequences are integration lattices and digital nets which we previ-
543 ously outlined in Table 1. These LD generators are comprised of two parts: the static
544 generating vectors $\boldsymbol{Z}_1, \boldsymbol{Z}_2, \boldsymbol{Z}_4, \ldots \in [0, 1)^d$ and the callable generator function. By
545 default, QMCPy provides a number of high-quality generating vectors for users to
546 choose from. For instance, the default ordinary lattice vector was constructed by
547 Cools, Kuo, and Nuyens [8] using component-by-component search, is extensible,
548 has order-2 weights, and supports up to 3600 dimensions and 2^{20} samples. However,
549 users who require more samples but fewer dimensions may switch to a generating
550 vector constructed using LatNet Builder [9, 33] to support 750 dimensions and 2^{24}
551 samples. Moreover, the `qp.Lattice` and `qp.DigitalNet` objects allow users
552 to input their own generating vectors to produce highly customized sequences. To
553 find such vectors, we recommend using LatNet Builder's construction routines as
554 the results can be easily parsed into a QMCPy-compatible format.

555 Along with the selection of a generating vector, QMCPy's low discrepancy
556 sequence routines expose a number of other customization parameters. For instance,
557 the lattice generator extends the Magic Point Shop [39] to support either linear or
558 natural ordering. Digital sequences permit either standard or Gray code ordering
559 and may be randomized via a digital shift optionally combined with a linear scram-
560 bling [37]. Halton sequences may be randomized via the routines of either Owen [43]
561 or Hofert and Lemieux [23].

8.2 A Word of Caution When Using LD Sequences

Although QMCPy's `DiscreteDistributions` have many of the same parameters and methods, users should be careful when swapping IID sequences with LD sequences. While IID node-sets have no preferred sample size, LD sequences often require special sampling ranges to ensure optimal discrepancy. As mentioned earlier, base-2 digital nets and extensible integration lattices show better evenness for sample sizes that are powers of 2. On the other hand, the preferred sample sizes for d-dimensional Halton sequences are $n = \prod_{j=1}^{d} p_j^{m_j}$ where p_j is the jth prime number and $m_j \in \mathbb{N}_0$ for $j = 1, \ldots, d$. Due to the infrequency of such values, Halton sequences are often regarded as not having a preferred sample size.

Users may also run into trouble when trying to generate too many points. Since QMCPy's generators construct sequences in 32-bit precision, generating greater than 2^{32} consecutive samples will cause the sequence to repeat. In the future, we plan to expand our generators to support optional 64-bit precision at the cost of greater computational overhead.

Another subtlety arises when transforming LD sequences to mimic different distributions. As mentioned earlier, unrandomized lattice and digital sequences include the origin, making transformations such as (7b) produce infinite values.

Some popular implementations of LD sequences drop the first point, which is the origin in the absence of randomization. The rationale is to avoid the transformation of the origin to infinity when mimicking a Gaussian or other distribution with an infinite sample space. Unfortunately, dropping the first point destroys some nice properties of the first $n = 2^m$ points of LD sequences, which can degrade the order of convergence for QMC cubature. A careful discussion of this matter is given by [42].

8.3 Transformations

The transformation $\boldsymbol{\Psi}$ connects a `DiscreteDistribution` and `TrueMeasure`. So far, we have assumed the `DiscreteDistribution` mimics a $\mathcal{U}[0, 1]^d$ distribution with PDF $\varrho(\boldsymbol{x}) = 1$. However, it may be advantageous to utilize a `DiscreteDistribution` that mimics a different distribution.

Suppose we have a `DiscreteDistribution` mimicking density ϱ supported on $\mathcal{X}$. Then using the variable transformation $\boldsymbol{\Psi}$,

$$\mu = \int_{\mathcal{T}} g(\boldsymbol{t}) \, \lambda(\boldsymbol{t}) \, \mathrm{d}\boldsymbol{t} = \int_{\mathcal{X}} f(\boldsymbol{x}) \, \varrho(\boldsymbol{x}) \mathrm{d}\boldsymbol{x} \quad \text{for } f(\boldsymbol{x}) = g\big(\boldsymbol{\Psi}(\boldsymbol{x})\big) \frac{\lambda\big(\boldsymbol{\Psi}(\boldsymbol{x})\big)}{\varrho(\boldsymbol{x})} \, |\boldsymbol{\Psi}'(\boldsymbol{x})|,$$

which generalizes (9). QMCPy also includes support for successive changes of measures so users may build complex variable transformations in an intuitive manner. Suppose that the variable transformation is a composition of several transformations: $\boldsymbol{\Psi} = \widehat{\boldsymbol{\Psi}}_L = \boldsymbol{\Psi}_L \circ \boldsymbol{\Psi}_{L-1} \circ \cdots \circ \boldsymbol{\Psi}_1$ as in (11). Here, $\boldsymbol{\Psi}_l : \mathcal{X}_{l-1} \to$

598 $\mathcal{X}_l$, $\mathcal{X}_0 = \mathcal{X}$, and $\mathcal{X}_L = \mathcal{T}$ so that the transformations are compatible with the
599 DiscreteDistribution and TrueMeasure. Let $\widehat{\boldsymbol{\Psi}}_l = \boldsymbol{\Psi}_l \circ \boldsymbol{\Psi}_{l-1} \circ \cdots \circ \boldsymbol{\Psi}_1$
600 denote the composition of the first l transforms and assume that $\widehat{\boldsymbol{\Psi}}_0(\boldsymbol{x}) = \boldsymbol{x}$, the
601 identity transform. Then we may write $\mu = \int_\mathcal{X} f(\boldsymbol{x})\, \varrho(\boldsymbol{x})\mathrm{d}\boldsymbol{x}$ for

$$
602 \qquad f(\boldsymbol{x}) = g\left(\widehat{\boldsymbol{\Psi}}_L(\boldsymbol{x})\right)\frac{\lambda\left(\widehat{\boldsymbol{\Psi}}_L(\boldsymbol{x})\right)}{\varrho(\boldsymbol{x})}\prod_{l=1}^{L}\left|\boldsymbol{\Psi}'_l\left(\widehat{\boldsymbol{\Psi}}_{l-1}(\boldsymbol{x})\right)\right|.
$$

603 It is often the case that $\boldsymbol{\Psi}_l$ is chosen such that $\boldsymbol{\Psi}_l(\boldsymbol{X})$ is stochastically equivalent to
604 a random variable with density λ_l on sample space $\mathcal{X}_l$ when $\boldsymbol{X}$ is a random variable
605 with density ϱ_l on sample space $\mathcal{X}_{l-1}$. This implies $\varrho_l(\boldsymbol{x}) = \lambda_l(\boldsymbol{\Psi}_l(\boldsymbol{x}))|\boldsymbol{\Psi}'_l(\boldsymbol{x})|$ so
606 that

$$
607 \qquad f(\boldsymbol{x}) = g\left(\widehat{\boldsymbol{\Psi}}_L(\boldsymbol{x})\right)\frac{\lambda\left(\widehat{\boldsymbol{\Psi}}_L(\boldsymbol{x})\right)}{\varrho(\boldsymbol{x})}\prod_{l=1}^{L}\frac{\varrho_l(\widehat{\boldsymbol{\Psi}}_{l-1}(\boldsymbol{x}))}{\lambda_l(\widehat{\boldsymbol{\Psi}}_l(\boldsymbol{x}))}.
$$

608 For an example, we return to the Keister integral (10). The following code con-
609 structs three Keister instances: one without importance sampling, one importance
610 sampled by a Gaussian distribution, and one importance sampled by the composition
611 of a Gaussian distribution with a Kumaraswamy distribution [29]. All Integrands
612 use a Sobol' DiscreteDistribution, making $\varrho(\boldsymbol{x}) = 1$ and $\mathcal{X} = [0,1]^d$. The
613 TrueMeasure is $\mathcal{N}(\boldsymbol{0}, \mathsf{I}/2)$ making $\lambda(\boldsymbol{t}) = \pi^{-d/2}\exp(-\boldsymbol{t}^T\boldsymbol{t})$ and $\mathcal{T} = \mathbb{R}^d$.
614 The table below displays the variable transformations and the measures for these
615 three cases. In all cases $\varrho_1(\boldsymbol{x}) = \cdots = \varrho_L(\boldsymbol{x}) = 1$ because the $\boldsymbol{\Psi}_l$ utilize inverse
616 cumulative distributions.

617

Integrand	L	λ_1	$\boldsymbol{\Psi}_1$	λ_2	$\boldsymbol{\Psi}_2$	f
K	1	$\mathcal{N}(\boldsymbol{0}, \mathsf{I}/2)$	(7b)			$g(\boldsymbol{\Psi}_1(\cdot))$
K_gauss	1	$\mathcal{N}(\boldsymbol{0}, 3\mathsf{I}/4)$	(7b)			$g(\boldsymbol{\Psi}_1(\cdot))\dfrac{\lambda(\boldsymbol{\Psi}_1(\cdot))}{\lambda_1(\boldsymbol{\Psi}_1(\cdot))}$
K_gauss_kuma	2	Kum	$\boldsymbol{F}_{\mathrm{Kum}}^{-1}$	$\mathcal{N}(\boldsymbol{0}, \mathsf{I})$	(7b)	$\dfrac{g(\boldsymbol{\Psi}_2(\boldsymbol{\Psi}_1(\cdot)))\lambda(\boldsymbol{\Psi}_2(\boldsymbol{\Psi}_1(\cdot)))}{\lambda_1(\boldsymbol{\Psi}_1(\cdot))\lambda_2(\boldsymbol{\Psi}_2(\boldsymbol{\Psi}_1(\cdot)))}$

618 Here Kum denotes the multivariate Kumaraswamy distribution with independent
619 marginals, and $\boldsymbol{F}_{\mathrm{Kum}}^{-1}$ denotes the element-wise inverse cumulative distribution func-
620 tion. The code below evaluates the Keister integral (10) for $d = 1$ and error tolerance
621 $\varepsilon = 5 \times 10^{-8}$. The timings for each of these different integrands are displayed.

```
622 >>> sobol = qp.Sobol(1)
623 >>> K = qp.Keister(sobol) # keister 0
624 >>> K_gauss = qp.Keister( # keister 1
625 ... qp.Gaussian(sobol, covariance=.75))
626 >>> K_gauss_kuma = qp.Keister( # keister 2
627 ... qp.Gaussian(
628 ... qp.Kumaraswamy(sobol, a=.8, b=.8)))
629 >>> for i,keister in enumerate([K, K_gauss, K_gauss_kuma]):
```

Fig. 7 Keister functions with and without importance sampling (`IS`). Note that the Keister functions using importance sampling are generally less variable and therefore easier to integrate, as evidenced by the faster integration times
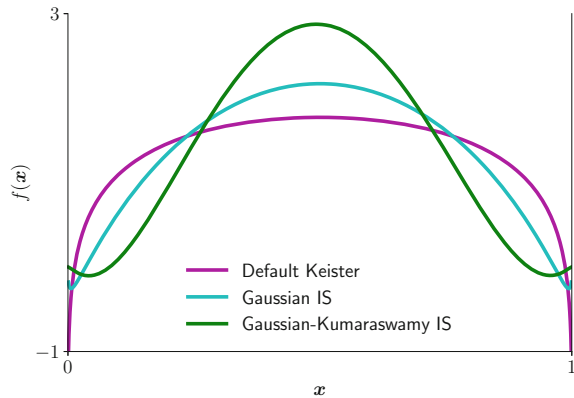

```
630  ... stopper = qp.CubQMCSobolG(keister, abs_tol=5e-8)
631  ... sol,data = stopper.integrate()
632  ... print("keister %d integration time = %.3f seconds"
633  ... %(i, data.time_integrate))
634  keister 0 integration time = 0.815 seconds
635  keister 1 integration time = 0.338 seconds
636  keister 2 integration time = 0.040 seconds
```

637   Successful importance sampling makes the transformed integrand, f, more flat.
638 The shorter cubature times correspond to flatter integrands, as illustrated in Fig. 7.
639 The above example uses $d = 1$ to facilitate the plot in Fig. 7; however, the same
640 example works for arbitrary dimensions.

9 Further Work

642 QMCPy is ripe for growth and development in several areas. We hope that the QMC
643 community will join us in making this a reality.

644   Multi-level (quasi-)Monte Carlo (ML(Q)MC) methods make possible the com-
645 putation of expectations of functionals of stochastic differential equations and partial
646 differential equations with random coefficients. Such use cases appear in quantitative
647 finance and geophysical applications. QMCPy's ML(Q)MC's capability is rudimen-
648 tary, but under active development.

649   We hope to add a greater variety of use cases and are engaging collaborators to
650 help. Sobol' indices, partial differential equations with random coefficients, expected
651 improvement measures for Bayesian optimization, and multivariate probabilities are
652 some of those on our radar.

653   Recently, several QMC experts have focused on developing LD generators for
654 Python. Well-established packages such as SciPy [50] and PyTorch [47] are have
655 developed QMC modules that support numerous LD sequences and related func-

tionalities. We plan to integrate the routines as optional backends for QMCPy's LD generators. Creating ties to these other packages will allow users to call their preferred generators from within the QMCPy framework. Moreover, as features in QMCPy become more common and prove their value, we will try to incorporate them into SciPy and other popular, general-purpose packages.

We also plan to expand our library of digital net generating matrices. We wish to incorporate interlaced digital nets, polynomial lattices, and Niederreiter sequences, among others. By including high-quality defaults in QMCPy, we hope to make these sequences more readily available to the public.

Our `DiscreteDistrution` places equal weights on each support point, X_i. In the future, we might generalize this to unequal weights.

QMCPy already includes importance sampling, but the choice of sampling distribution must be chosen a priori. We would like to see an automatic, adaptive choice following the developments of [2, 35, 45].

Control variates can be useful for QMC as well as for IID MC [22]. These should be incorporated into QMCPy in a seamless way.

We close with an invitation. Try QMCPy. If you find bugs or missing features, please submit an issue to https://github.com/QMCSoftware/QMCSoftware/issues. If you wish to add your great algorithm or use case, please submit a pull request to our GitHub repository at https://github.com/QMCSoftware/QMCSoftware/pulls. We hope that the community will embrace QMCPy.

References

1. Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G.S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., Zheng, X.: TensorFlow: Large-scale machine learning on heterogeneous systems (2015). https://www.tensorflow.org/. Software available from tensorflow.org
2. Asmussen, S., Glynn, P.: Stochastic simulation: algorithms and analysis. In: Stochastic Modelling and Applied Probability, vol. 57. Springer, New York (2007). https://doi.org/10.1007/978-0-387-69033-9
3. Bezanson, J., Karpinski, S., Shah, V.B., Edelman, A.: Julia: A fast dynamic language for technical computing (2012). arXiv:1209.5145
4. Burkhardt, J.: Various Software (2020). http://people.sc.fsu.edu/~jburkardt/
5. Choi, S.C.T., Ding, Y., Hickernell, F.J., Jiang, L., Jiménez Rugama, Ll.A., Li, D., Jagadeeswaran, R., Tong, X., Zhang, K., Zhang, Y., Zhou, X.: GAIL: Guaranteed Automatic Integration Library (versions 1.0–2.3.2). MATLAB software. http://gailgithub.github.io/GAIL_Dev/ (2021). https://doi.org/10.5281/zenodo.4018189

6. Choi, S.C.T., Hickernell, F.J., Jagadeeswaran, R., McCourt, M., Sorokin, A.: QMCPy: A quasi-Monte Carlo Python library (2020). https://doi.org/10.5281/zenodo.3964489, https://qmcsoftware.github.io/QMCSoftware/

7. Choi, S.C.T., Hickernell, F.J., Jagadeeswaran, R., McCourt, M.J., Sorokin, A.: QMCPy Documentation (2020). https://qmcpy.readthedocs.io/en/latest/

8. Cools, R., Kuo, F.Y., Nuyens, D.: Constructing embedded lattice rules for multivariate integration. SIAM J. Sci. Comput. **28**(6), 2162–2188 (2006). https://doi.org/10.1137/06065074X

9. Darmon, Y., Godin, M., L'Ecuyer, P., Jemel, A., Marion, P., Munger, D.: LatNet Builder (2018). https://github.com/umontreal-simul/latnetbuilder

10. Dick, J., Kuo, F., Sloan, I.H.: High dimensional integration—the Quasi-Monte Carlo way. Acta Numer. **22**, 133–288 (2013). https://doi.org/10.1017/S0962492913000044

11. Giles, M.: Multi-level (Quasi-)Monte Carlo software (2020). https://people.maths.ox.ac.uk/gilesm/mlmc/

12. Google Inc.: TF Quant Finance: Tensorflow Based Quant Finance Library (2021). https://github.com/google/tf-quant-finance

13. Harris, C.R., Millman, K.J., van der Walt, S.J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N.J., Kern, R., Picus, M., Hoyer, S., van Kerkwijk, M.H., Brett, M., Haldane, A., del Río, J.F., Wiebe, M., Peterson, P., Gérard-Marchant, P., Sheppard, K., Reddy, T., Weckesser, W., Abbasi, H., Gohlke, C., Oliphant, T.E.: Array programming with NumPy. Nature **585**(7825), 357–362 (2020). https://doi.org/10.1038/s41586-020-2649-2, https://doi.org/10.1038/s41586-020-2649-2

14. Hickernell, F.J., Choi, S.C.T., Jiang, L., Jiménez Rugama, L.A.: Monte Carlo Simulation, Automatic Stopping Criteria for Wiley StatsRef: Statistics Reference Online, pp. 1–7 (2014)

15. Hickernell, F.J., Sorokin, A.: Quasi-Monte Carlo (QMC) software in QMCPy Google Colaboratory Notebook (2020). http://tinyurl.com/QMCPyTutorial

16. Hickernell, F.J., Sorokin, A.: Quasi-Monte Carlo (QMC) software in QMCPy Google Colaboratory Notebook for MCQMC2020 Article (2020). https://tinyurl.com/QMCPyArticle2021

17. Hickernell, F.J.: A generalized discrepancy and quadrature error bound. Math. Comput. **67**, 299–322 (1998). https://doi.org/10.1090/S0025-5718-98-00894-1

18. Hickernell, F.J.: Goodness-of-fit statistics, discrepancies and robust designs. Statist. Prob. Lett. **44**, 73–78 (1999). https://doi.org/10.1016/S0167-7152(98)00293-4

19. Hickernell, F.J., Jiang, L., Liu, Y., Owen, A.B.: Guaranteed conservative fixed width confidence intervals via Monte Carlo sampling. In: Dick, J., Kuo, F.Y., Peters, G.W., Sloan, I.H. (eds.) Monte Carlo and Quasi-Monte Carlo Methods 2012, Springer Proceedings in Mathematics and Statistics, vol. 65, pp. 105–128. Springer, Berlin (2013). https://doi.org/10.1007/978-3-642-41095-6

20. Hickernell, F.J., Jiménez Rugama, Ll.A.: Reliable adaptive cubature using digital sequences. In: Cools, R., Nuyens, D. (eds.) Monte Carlo and Quasi-Monte Carlo Methods: MCQMC, Leuven, Belgium, April 2014, Springer Proceedings in Mathematics and Statistics, vol. 163, pp. 367–383. Springer, Berlin (2016). ArXiv:1410.8615 [math.NA]

21. Hickernell, F.J., Jiménez Rugama, Ll.A., Li, D.: Adaptive quasi-Monte Carlo methods for cubature. In: Dick, J., Kuo, F.Y., Woźniakowski, H. (eds.) Contemporary Computational Mathematics—A Celebration of the 80th Birthday of Ian Sloan, pp. 597–619. Springer, Berlin (2018). https://doi.org/10.1007/978-3-319-72456-0

22. Hickernell, F.J., Lemieux, C., Owen, A.B.: Control variates for quasi-Monte Carlo. Stat. Sci. **20**, 1–31 (2005). https://doi.org/10.1214/088342304000000468

23. Hofert, M., Lemieux, C.: qrng R package (2017). https://cran.r-project.org/web/packages/qrng/qrng.pdf

24. Jagadeeswaran, R., Hickernell, F.J.: Fast automatic Bayesian cubature using lattice sampling. Stat. Comput. **29**, 1215–1229 (2019). https://doi.org/10.1007/s11222-019-09895-9

25. Jagadeeswaran, R., Hickernell, F.J.: Fast automatic Bayesian cubature using Sobol' sampling (2021+). In preparation for submission for publication

26. Jiménez Rugama, Ll.A., Hickernell, F.J.: Adaptive multidimensional integration based on rank-1 lattices. In: Cools, R., Nuyens, D. (eds.) Monte Carlo and Quasi-Monte Carlo Methods:

752   MCQMC, Leuven, Belgium, April 2014, Springer Proceedings in Mathematics and Statistics,
753   vol. 163, pp. 407–422. Springer, Berlin (2016). ArXiv:1411.1966
754   27. Keister, B.D.: Multidimensional quadrature algorithms. Comput. Phys. **10**, 119–122 (1996).
755   https://doi.org/10.1063/1.168565
756   28. Kucherenko, S.: BRODA (2020). https://www.broda.co.uk/index.html
757   29. Kumaraswamy, P.: A generalized probability density function for double-bounded random
758   processes. J. Hydrol. **46**(1), 79–88 (1980). https://doi.org/10.1016/0022-1694(80)90036-0
759   30. Kuo, F.Y., Nuyens, D.: Application of quasi-Monte Carlo methods to elliptic PDEs with random
760   diffusion coefficients—a survey of analysis and implementation. Found. Comput. Math. **16**,
761   1631–1696 (2016). https://people.cs.kuleuven.be/~dirk.nuyens/qmc4pde/
762   31. Lataniotis, C., Marelli, S., Sudret, B.: Uncertainty quantification in the cloud with UQCloud.
763   In: 4th International Conference on Uncertainty Quantification in Computational Sciences and
764   Engineering (UNCECOMP 2021), pp. 209–217 (2021)
765   32. L'Ecuyer, P.: SSJ: Stochastic Simulation in Java (2020). https://github.com/umontreal-simul/
766   ssj
767   33. L'Ecuyer, P., Marion, P., Godin, M., Puchhammer, F.: A tool for custom construction of QMC
768   and RQMC point sets. In: Arnaud, E., Giles, M., Keller, A. (eds.) Monte Carlo and Quasi-Monte
769   Carlo Methods: MCQMC, Oxford 2020 (2021+)
770   34. L'Ecuyer, P., Munger, D.: Algorithm 958: Lattice Builder: a general software tool for con-
771   structing rank-1 latice rules. ACM Trans. Math. Softw. **42**, 1–30 (2016)
772   35. L'Ecuyer, P., Tuffin, B.: Approximate zero-variance simulation. In: Proceedings of the 40th
773   Conference on Winter Simulation, WSC '08, pp. 170–181. Winter Simulation Conference
774   (2008)
775   36. Marelli, S., Sudret, B.: UQLab: A framework for uncertainty quantification in MATLAB.
776   In: The 2nd International Conference on Vulnerability and Risk Analysis and Management
777   (ICVRAM 2014), pp. 2554–2563. ASCE Library (2014). https://www.uqlab.com
778   37. Matoušek, J.: On the L_2-discrepancy for anchored boxes. J. Complex. **14**, 527–556 (1998)
779   38. Niederreiter, H.: Random Number Generation and Quasi-Monte Carlo Methods. CBMS-NSF
780   Regional Conference Series in Applied Mathematics. SIAM, Philadelphia (1992)
781   39. Nuyens, D.: Magic Point Shop (2017). https://people.cs.kuleuven.be/~dirk.nuyens/qmc-
782   generators/
783   40. OpenTURNS Developers: An Open Source Initiative for the Treatment of Uncertainties, Risks
784   'N Statistics (2020). http://www.openturns.org
785   41. Owen, A.B.: Scrambling Sobol' and Niederreiter-Xing points. J. Complex. **14**(4), 466–489
786   (1998)
787   42. Owen, A.B.: On Dropping the First Sobol' Point (2020). ArXiv:2008.08051 [math.NA]
788   43. Owen, A.B.: Randomized Halton Sequences in R (2020). http://statweb.stanford.edu/~owen/
789   code/
790   44. Owen, A.B.: About the R function: rsobol (2021). https://statweb.stanford.edu/~owen/reports/
791   seis.pdf
792   45. Owen, A.B., Zhou, Y.: Safe and effective importance sampling. J. Am. Stat. Assoc. **95**, 135–143
793   (2000)
794   46. Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z.,
795   Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani,
796   A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., Chintala, S.: PyTorch: An imperative style,
797   high-performance deep learning library. In: Wallach, H., Larochelle, H., Beygelzimer, A.,
798   d'Alché-Buc, F., Fox, E., Garnett, R. (eds.) Advances in Neural Information Processing Sys-
799   tems 32, pp. 8024–8035. Curran Associates, Inc. (2019). http://papers.neurips.cc/paper/9015-
800   pytorch-an-imperative-style-high-performance-deep-learning-library.pdf
801   47. PyTorch Developers: PyTorch (2020). https://pytorch.org
802   48. Robbe, P.: Low Discrepancy Sequences in Julia (2020). https://github.com/PieterjanRobbe/
803   QMC.jl
804   49. Robbe, P.: Multilevel Monte Carlo simulations in Julia (2021). https://github.com/
805   PieterjanRobbe/MultilevelEstimators.jl

806   50. SciPy Developers: SciPy Ecosystem (2018). www.scipy.org

807   51. The MathWorks Inc: MATLAB R2021a. Natick, MA (2020)

808   52. Virtanen, P., Gommers, R., Oliphant, T.E., Haberland, M., Reddy, T., Cournapeau, D., Burovski,
809   E., Peterson, P., Weckesser, W., Bright, J., van der Walt, S.J., Brett, M., Wilson, J., Millman,
810   K.J., Mayorov, N., Nelson, A.R.J., Jones, E., Kern, R., Larson, E., Carey, C.J., Polat, İ., Feng, Y.,
811   Moore, E.W., VanderPlas, J., Laxalde, D., Perktold, J., Cimrman, R., Henriksen, I., Quintero,
812   E.A., Harris, C.R., Archibald, A.M., Ribeiro, A.H., Pedregosa, F., van Mulbregt, P., SciPy 1.0
813   Contributors: SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. Nat.
814   Methods **17**, 261–272 (2020). https://doi.org/10.1038/s41592-019-0686-2

Author Queries

Chapter 2

Query Refs.	Details Required	Author's response
AQ1	As keywords are mandatory for this chapter, please provide 3–6 keywords.	

MARKED PROOF

Please correct and return this set

Please use the proof correction marks shown below for all alterations and corrections. If you wish to return your proof by fax you should ensure that all amendments are written clearly in dark ink and are made well within the page margins.

Instruction to printer	Textual mark	Marginal mark
Leave unchanged	• • • under matter to remain	ⓙ
Insert in text the matter indicated in the margin	⋏	New matter followed by ⋏ or ⋏⊘
Delete	/ through single character, rule or underline or ⊢──────┤ through all characters to be deleted	⌿ or ⌿⊘
Substitute character or substitute part of one or more word(s)	/ through letter or ⊢──────┤ through characters	new character / or new characters /
Change to italics	── under matter to be changed	‿
Change to capitals	≡ under matter to be changed	≡
Change to small capitals	= under matter to be changed	=
Change to bold type	⌄⌄⌄ under matter to be changed	⌄⌄⌄
Change to bold italic	≋ under matter to be changed	≋
Change to lower case	Encircle matter to be changed	≢
Change italic to upright type	(As above)	⌄|⌄
Change bold to non-bold type	(As above)	⌿|⌿
Insert 'superior' character	/ through character or ⋏ where required	⋎ or ⋋ under character e.g. ⋎² or ⋋²
Insert 'inferior' character	(As above)	⋏ over character e.g. ⋏₂
Insert full stop	(As above)	⊙
Insert comma	(As above)	,
Insert single quotation marks	(As above)	⋎ or ⋋ and/or ⋎ or ⋋
Insert double quotation marks	(As above)	⋎ or ⋋ and/or ⋎ or ⋋
Insert hyphen	(As above)	⊢⊣
Start new paragraph	⌐	⌐
No new paragraph	⌒	⌒
Transpose	⊔⊓	⊔⊓
Close up	linking ⌒ characters	◯
Insert or substitute space between characters or words	/ through character or ⋏ where required	Y
Reduce space between characters or words	| between characters or words affected	↑