Quasi-Monte Carlo Software

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

Abstract Practitioners wishing to experience the efficiency gains from using low discrepancy sequences need correct, 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 to approximate multivariate integrals or expectations of functions of vector random variables by QMC. 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.

Sou-Cheng T. Choi
Department of Applied Mathematics, Illinois Institute of Technology,
RE 220, 10 W. 32nd St., Chicago, IL 60616; and Kamakura Corporation, 2222 Kalakaua Ave, Suite 1400, Honolulu, HI 96815 e-mail: schoi32@iit.edu

Fred 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 e-mail: hickernell@iit.edu

R. Jagadeeswaran
Department of Applied Mathematics, Illinois Institute of Technology,
RE 220, 10 W. 32nd St., Chicago, IL 60616 e-mail: jaarathin1@iit.edu; and
Wi-Tronix LLC, 631 E Boughton Rd, Suite 240, Bolingbrook, IL 60440

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

Aleksei G. Sorokin
Department of Applied Mathematics, Illinois Institute of Technology,
RE 220, 10 W. 32nd St., Chicago, IL 60616 e-mail: asorokin@hawk.iit.edu
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 Figure 6). Often, these efficiency gains are obtained simply by replacing IID sampling with low discrepancy (LD) sampling, which is the heart of QMC.

As a practitioner, you might wish to test whether QMC would speed up your computation. You would like to easily access the best QMC algorithms available. As a theoretician or an algorithm developer, you might want to demonstrate your best 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 [3], 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,

\[ \mu := \int_T g(t) \lambda(t) \, dt, \]  

where \( g \) is the integrand, and \( \lambda \) is a non-negative weight. If \( \lambda \) is a probability distribution (PDF) for the random variable \( T \), then \( \mu \) is the mean of \( g(T) \).

Regardless, we perform a suitable variable transformation to interpret this integral as the mean of a function of a multivariate, standard uniform random variable:

\[ \mu = \mathbb{E}[f(X)] = \int_{[0,1]^d} f(x) \, dx, \quad X \sim \mathcal{U}[0,1]^d. \]  

QMC approximates the population mean, \( \mu \), by a sample mean,

\[ \hat{\mu} := \frac{1}{n} \sum_{i=0}^{n-1} f(X_i), \quad X_0, X_1, \ldots, X_M \sim \mathcal{U}[0,1]^d. \]  

The choice of this sequence and the choice of \( n \) to satisfy the prescribed error requirement,

\[ |\mu - \hat{\mu}| \leq \varepsilon \quad \text{absolutely or with high probability}, \]  

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

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

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 $X_0, X_1, \ldots \sim F$ 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 $\mu$ by the sample mean, the root mean squared error is $O(n^{-1/2})$. Figure 1 displays IID uniform points, $X_0^{\text{IID}}, X_1^{\text{IID}}, \ldots \sim \mathcal{U}[0,1]^2$, i.e., the target distribution is $F_{\text{unif}}(x) = x_1x_2$.

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 $X_0, X_1, \ldots \sim \mathcal{U}[0,1]^d$, 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, $F_{\text{unif}}$, well (relative to $n$). (The empirical distribution of a set assigns equal probability to each point.) The measure of the difference between the empirical distribution of a set of points and the uniform distribution is called a discrepancy and is denoted $D\{X_i\}_{i=0}^{n-1}$ [7, 12, 13, 30]. 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, $X_0^{\text{LD}}, X_1^{\text{LD}}, \ldots \sim \mathcal{U}[0,1]^2$, in this case linearly scrambled Sobol’ points.

The error in using the sample mean to approximate the integral can be bounded according to the Koksma-Hlawka inequality and its extensions [7, 12, 13, 30] as the product of the discrepancy of the sampling sequence and the variation of the integrand, denoted $V()$:
\[ |\mu - \tilde{\mu}| = \left| \int_{[0,1]^d} f(x) d(F_{\text{unif}} - F_{\{X_i\}_{i=0}^{n-1}})(x) \right| \leq D(\{X_i\}_{i=0}^{n-1}) V(f), \quad (5) \]

The variation is a (semi-) norm of the integrand in a suitable Banach space. The discrepancy corresponds to the norm of the error functional for that Banach space. For typical Banach spaces, the discrepancy of LD points is \( O(n^{-1+\epsilon}) \), which is a higher convergence order than for IID points. For details, the reader is referred to the references. Here, we expect the reader to see in Figure 1 that the LD points cover the integration domain more evenly than IID points. LD sampling can be thought of as highly stratified sampling. In the examples below the reader will see the demonstrably smaller cubature errors arising from using LD points.

In the sections that follow we first overview available QMC software. We next describe an architecture for good QMC software, i.e., what are the key components and how they interact. We then describe how we have implemented this architecture in QMCPy. Finally, we summarize further directions that we hope QMCPy and other QMC software projects will take. Those interested in following or contributing to the development of QMCPy are urged to visit the GitHub repository at https://github.com/QMCSoftware/QMCSofware.

We have endeavored to be as accurate as possible at the time of writing this article. We hope that progress in QMC software development will make this article somewhat obsolete in a good way in the next several years.

2 Available Software for QMC

QMC software spans LD sequence generators, cubatures, and applications. We review the better known software collections, recognizing that some software overlaps multiple categories. Software focusing on generating high quality LD sequences and their generators includes

- **BRODA** Sobol’ sequences in C, MATLAB, and Excel [23],
- **Burkhardt** Various QMC software in C++, Fortran, MATLAB, and Python [1],
- **LatNet Builder** Generating vectors/matrices for lattices and digital nets [26, 27],
- **MATLAB** Sobol’ and Halton sequences, commercial [38],
- **MPS** Magic Point Shop, lattices and Sobol’ sequences [31],
- **Owen** Randomized Halton sequences in R [34],
- **PyTorch** Scrambled Sobol’ sequences [35],
- **QMC.jl** LD Sequences in Julia [36], and
- **qrng** Sobol’, Halton, and Korobov sequences in R [18].

Software focusing on QMC cubatures and applications includes

- **GAIL** Automatic (Q)MC stopping criteria in MATLAB [2],
Quasi-Monte Carlo Software

ML(Q)MC Multi-Level (Q)MC routines in C++, MATLAB, Python, and R [8].
OpenTURNS Open source initiative for the Treatment of Uncertainties, Risks ’N Statistics in Python [32],
QMC4PDE QMC for elliptic PDEs with random diffusion coefficients [24],
SSJ Stochastic Simulation in Java [25], and
UQLab Framework for Uncertainty Quantification in MATLAB [28].

The sections that follow describe QMCPy [3], which is our attempt to combine some of the best of the above software under a common user interface written in Python 3. The choice of language was determined by the desire to make QMC software accessible to a broad audience, especially the tech industry.

3 Components of QMC Software

QMC cubature can be summarized as follows. We want to approximate the expectation, $\mu$, well by the sample mean, $\hat{\mu}$, where (1), (2), and (3) combine to give

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

where $X \sim \mathcal{U}[0,1]^d$, $X_0, X_1, \ldots \sim \mathcal{U}[0,1]^d$. (6)

Moreover, we want to satisfy the error requirement in (4). This requires four components, which we implement as QMCPy classes.

- **Discrete Distribution** produces $X_0, X_1, \ldots$ mimicking $\mathcal{U}[0,1]^d$;
- **True Measure** $t \mapsto \lambda(t)\,dt$ defines the original integral, e.g., Gaussian or Lebesgue;
- **Integrand** $g$ defines the original, and $f$ defines the transformed version to fit the DiscreteDistribution; and
- **Stopping Criterion** determines how large $n$ should be to ensure that $|\mu - \hat{\mu}| \leq \varepsilon$ as in (4).

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

Thorough documentation of all QMCPy classes is available in [4]. Demonstrations of how QMCPy work are given in Google Colab notebooks [10, 11].
The project may be installed from PyPI into your Python 3 environment via the command `pip install qmcpy`. In the code snippets that follow, we assume QMCPy has been imported alongside NumPy [9] via the following commands in a Python Console:

```python
>>> import qmcpy as qp
>>> import numpy as np
```

4 Discrete Distributions

LD sequences typically mimic $\mathcal{U}[0,1]^d$. Good sequences mimicking other distributions are obtained by transformations as described in the next section.

QMCPy implements extensible LD sequences, i.e., those that allow practitioners to obtain and use $X_n, X_{n+1}, \ldots$ without discarding $X_0, \ldots, X_{n-1}$. Halton sequences do not have preferred sample sizes $n$, but integration lattices and digital sequences prefer $n$ to be a power of 2.

These latter two also have an elegant group structure, which we summarize in Table 1. The addition operator is $\oplus$. The unshifted sequence is $Z_0, Z_1, \ldots$ and the randomly shifted sequence is $Z_0, Z_1, \ldots$

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

| Define ... | \(Z_1, Z_2, Z_3, \ldots \in [0,1]^d\) chosen well |
|------------|---------------------------------------------------|
| \(Z_i := i_0 Z_1 \oplus i_1 Z_2 \oplus i_2 Z_3 \oplus i_3 Z_8 \oplus \cdots\) for \(i = i_0 + i_1 2 + i_2 4 + i_3 8 + \cdots\), \(i, \ell \in \{0,1\}\) |
| \(X_i := Z_i \oplus \Delta\), where \(\Delta \sim \mathcal{U}(0,1)^d\) |

| Rank-1 Integration Lattices | Digital Nets |
|-----------------------------|--------------|
| \(t \oplus x := (t + x) \mod 1\) | \(t \oplus x := \text{binary digitwise addition, } \oplus_{\text{dig}}\) |
| require \(Z_{2m} \oplus Z_{2m} = \lfloor Z_{2m} - 1 \rfloor\) \(\forall m \in \mathbb{N}_0\) |

| Then it follows that ... |
|--------------------------|
| \(P_m := \{Z_0, \ldots, Z_{2^m-1}\}, Z_i \oplus Z_j \in P_m\) |
| \(P_{\Delta, m} := \{X_0, \ldots, X_{2^m-1}\}, X_i \oplus X_j \oplus X_k \in P_{\Delta, m}\) |
| \(\forall i, j, k \in \{0,\ldots,2^m-1\}\) |

We illustrate lattice and Sobol’ sequences using QMCPy. First, we create an instance of a $d = 2$ dimensional Lattice object of the DiscreteDistribution abstract class. Then we generate the first eight (non-randomized) points in this lattice.

```python
>>> lattice = qp.Lattice(dimension=2, randomize=False)
```
The first three generators for this lattice are \(Z_1 = (0.5,0.5)\), \(Z_2 = (0.25,0.75)\), and \(Z_4 = (0.125,0.375)\). One can check that \((Z_2 + Z_4) \mod 1 = (0.375,0.125) = Z_6\), 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 \texttt{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.

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

\[
Z_2 \oplus_{\text{dig}} Z_4 = (0.25,0.75) \oplus_{\text{dig}} (0.125,0.625) \\
= (0.010,0.110) \oplus_{\text{dig}} (0.001,0.101) = (0.011,0.011) \\
= (0.375,0.375) = Z_6.
\]

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

\[
Z_6 = Z_2 \oplus_{\text{dig}} Z_4 = (0.010,0.110) \oplus_{\text{dig}} (0.001,0.011) \\
= (0.011,0.101) = (0.375,0.625),
\]
which differs from the $Z_0 = (0.375,0.125)$ constructed for lattices. To emphasize, lattices and digital sequences are different, even if they share the same generators, $Z_1, Z_2, Z_4, \ldots$.

The examples of `qp.Lattice` and `qp.Sobol` illustrate how QMCPy LD generators share a common user interface. The dimension is specified when the instance is constructed, and the number of points is specified when the `gen_samples` method is called. Following Python practice, parameters can be input without specifying their names if they are input in the prescribed order. QMCPy also includes Halton sequences and IID sequences, again deferring details to the QMCPy documentation [4].

A crucial difference between IID generators and LD generators is reflected in the behavior of generating $n$ points. For an IID generator, asking for $n$ points repeatedly gives you different points each time because they are meant to be random and independent.

```python
>>> iid = qp.IIDStdUniform(2); iid.gen_samples(1)
array([[ 0.17496884,  0.56612647]])
>>> iid.gen_samples(1)
array([[ 0.82314635,  0.26919205]])
```

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

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

```python
>>> lattice = qp.Lattice(2); lattice.gen_samples(1)
array([[ 0.15700854,  0.41080809]])
>>> lattice.gen_samples(1)
array([[ 0.15700854,  0.41080809]])
```

Here we allow the randomization so that the first point in the sequence is not the origin. To obtain the next $n$ points one may specify the start and ending indices of the sequence.

```python
>>> lattice.gen_samples(2)
array([[ 0.15700854,  0.41080809],
       [ 0.65700854,  0.91080809]])
>>> lattice.gen_samples(n_min=1, n_max=2)
array([[ 0.65700854,  0.91080809]])
```

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

### 5 True Measures

The LD sequences implemented as `DiscreteDistribution` objects mimic the $U[0,1]^d$ distribution. However, we may need sequences to mimic other
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 points.

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:

```python
>>> u = qp.Uniform(
...   sampler = qp.Halton(2),
...   lower_bound = [-2, 0],
...   upper_bound = [2, 4])
```
Fig. 3 Sobol’ samples transformed to mimic a uniform $U\left(\left[-2, 2\right]\right)$ (left) and Gaussian $N\left(\left[3/2, 9/5\right]\right)$ (right) distribution.

```python
>>> u.gen_samples(4)
array([[ 1.15995575, 3.5553664 ],
       [-0.84004425, 0.88869974],
       [ 0.15995575, 2.22203307],
       [-1.84004425, 3.99981085]])
```

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

```python
>>> g = qp.Gaussian(qp.Lattice(2),
                 mean = [3, 2],
                 covariance = [[9, 5],
                           [5, 4]])
>>> g.gen_samples(4)
array([[2.65147682, 1.86323299],
       [9.06342728, 3.60370135],
       [0.2084947 , 1.13289686],
       [4.91327026, 2.52208387]])
```

Here the covariance decomposition $\Sigma = AA^T$ 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 $T$ correspond to the discretized Brownian motion at times $\tau/d, 2\tau/d, \ldots, \tau$, where $\tau$ is the time horizon. The distribution is a special case of the Gaussian with covariance

$$
\Sigma = (\tau/d) \left( \min(j,k) \right)_{j,k=1}^d
$$

and mean $a$, which corresponds to a drift coefficient times $(\tau/d)(1,2,\ldots,d)^T$. The code for generating a Brownian motion is
Fig. 4 Sobol’ samples transformed to mimic a 52-dimensional Brownian Motion without drift (left) and with drift coefficient 2 (right).

```python
>>> bm = qp.BrownianMotion(qp.Sobol(4), drift=2)
>>> bm.gen_samples(2)
array([[0.24724376, 1.04423332, 0.62553528, 0.87167138],
       [1.33604991, 1.92610875, 3.52091525, 4.34341092]])
```

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 $t = \Psi(x)$ where $\Psi: [0,1]^d \to \mathcal{T}$. This leads to

$$\mu = \int_{\mathcal{T}} g(t) \lambda(t) dt = \int_{[0,1]^d} g(\Psi(x)) \lambda(\Psi(x)) |\Psi'(x)| dx = \int_{[0,1]^d} f(x) dx,$$

where $f(x) = g(\Psi(x)) \lambda(\Psi(x)) |\Psi'(x)|$, (9)

and $|\Psi'(x)| := |\partial \Psi / \partial x|$ represents the Jacobian of the variable transformation. The abstract class `Integrand` provides $f$ based on the user’s input of $g$ and the `TrueMeasure` instance, which defines $\lambda$ and the transformation $\Psi$. Different choices of $\Psi$ lead to different $f$, which may give different rates of convergence of the cubature, $\tilde{\mu}$ to $\mu$.

We illustrate the `Integrand` class via an example of Keister [22]:
\[ \mu = \int_{\mathbb{R}^d} \cos(\|t\|) \exp(-t^T t) \, dt = \int_{\mathbb{R}^d} \frac{\pi^{d/2} \cos(\|t\|) \, \pi^{-d/2} \exp(-t^T t)}{\lambda(t)} \, dt. \quad (10) \]

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

\[ \mu = \int_{[0,1]^d} \frac{\pi^{d/2} \cos(\|\Psi(x)\|) \, dx}{f(x)} \]

\( \Psi(x) := \sqrt{1/2} \Phi^{-1}(x) \).

The code below sets up an Integrand instance using QMCPy’s CustomFun wrapper to tie a user-defined function \( g \) into the QMCPy framework. Then we evaluate the sample mean of \( n = 1000 \) \( f \) values obtained by sampling at transformed Halton points.

```python
>>> def my_Keister(t):
...     d = t.shape[1] # t is an (n x d) array
...     norm = np.sqrt((t**2).sum(1))
...     out = np.pi**(d/2)*np.cos(norm)
...     return out # size n vector
>>> gauss = qp.Gaussian(qp.Halton(2), covariance=1/2)
>>> keister = qp.CustomFun(true_measure=gauss, g=my_Keister)
>>> x = keister.discrete_distrib.gen_samples(1000)
>>> y = keister.f(x)
>>> y.mean()
1.8068921077383129
```

We have no indication yet of how accurate our approximation is. That topic is treated in the next section. Figure 5 visualizes sampling on the original integrand, \( g \), and sampling on the transformed integrand, \( f \).

Another way to approximate the Keister integral in (10) is to write it as an integral with respect to the Lebesgue measure:

\[ \mu = \int_{\mathbb{R}^d} \cos(\|t\|) \exp(-t^T t) \frac{1}{\lambda(t)} \, dt \]

\[ = \int_{[0,1]^d} \frac{\cos(\|\Psi(x)\|) \exp(-\Psi^T(x)\Psi(x)) |\Psi'(x)| \, dx}{f(x)} \]

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

```python
>>> 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)
```
Right: Sampling the transformed integrand $f$ at Halton points $X_i \sim U([0,1]^2)$.

Left: Sampling the original integrand $g$ at $T_i = \Psi(X_i) \sim \mathcal{N}(0,1/2)$ where $\Psi$ is defined in (7b).

```
>>> y.mean()
1.8081374367858
```

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

In the examples above, one must input the correct $g$ into `CustomFun` along with the correct `TrueMeasure` $\lambda$ to define the integration problem. The `Keister` integrand included in the QMCPy library takes a more flexible approach of defining the integration problem $\mu$ in (10). Selecting a different sampler $\Psi$ performs importance sampling which leaves $\mu$ unchanged.

```
>>> # default transform
>>> keister = qp.Keister(qp.Halton(2))
>>> x = keister.discrete_distrib.gen_samples(1e4)
>>> keister.f(x).mean()
1.8082700897855786

>>> # custom transform for importance sampling
>>> keister = qp.Keister(sampler=qp.Gaussian(qp.Halton(2)))
>>> x = keister.discrete_distrib.gen_samples(1e4)
>>> keister.f(x).mean()
1.807929542158969
```

In the first case above, the $\lambda$ in (9) corresponds to the Gaussian density with mean zero and variance $1/2$ by default, and the corresponding variable transformation, $\Psi$, is chosen to make $\lambda(\psi(x))|\psi'(x)| = 1$ and $f(x) = g(\psi(x))$. In the second case, we choose an importance sampling density $\lambda_{IS}$, corresponding to standard Gaussian, and the variable transformation $\psi_{IS}$ makes $\lambda_{IS}(\psi_{IS}(x))|\psi_{IS}'(x)| = 1$. Then
\[ \mu = \int_T g(t) \lambda(t) \, dt = \int_T g(t) \frac{\lambda(t)}{\lambda_{IS}(t)} \lambda_{IS}(t) \, dt \]

\[ = \int_{[0,1]^d} g(\Psi_{IS}(x)) \frac{\lambda(\Psi_{IS}(x))}{\lambda_{IS}(\Psi_{IS}(x))} \lambda_{IS}(\Psi_{IS}(x)) |\Psi_{IS}'(x)| \, dx \]

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

where \( f_{IS}(x) = g(\Psi_{IS}(x)) \frac{\lambda(\Psi_{IS}(x))}{\lambda_{IS}(\Psi_{IS}(x))} \).

(11)

Because LD samples mimic \( U[0,1]^d \), choosing a different sampler is equivalent to choosing a different variable transform.

7 Stopping Criteria

The StoppingCriterion object determines the number of samples \( n \) that are required for the sample mean approximation \( \bar{\mu} \) to be within error tolerance \( \varepsilon \) of the true mean \( \mu \). Several QMC stopping criteria have been implemented in QMCPy, including replications, stopping criteria that track the decay of the Fourier complex exponential or Walsh coefficients of the integrand [15, 16, 21], and stopping criteria based on Bayesian credible intervals [19, 20].

Let us return to the Keister example from the previous section. After setting up a default Keister instance via a Sobol' DiscreteDistribution, we choose a StoppingCriterion object that matches the DiscreteDistribution and input our desired tolerance. Calling the integrate method returns the approximate integral plus some useful information about the computation.

```python
>>> keister = qp.Keister(qp.Sobol(2))
>>> stopping = qp.CubQMCSobolG(keister, abs_tol=1e-3)
>>> solution_qmc, data_qmc = stopping.integrate()
>>> data_qmc  # equivalent to print(data_qmc)
Solution: 1.8083
Keister (Integrand Object)
Sobol (DiscreteDistribution Object)
d 2^(-1)
  randomize 1
  graycode 0
  seed 194061
  mimics StdUniform
  dim0 0
Gaussian (TrueMeasure Object)
  mean 0
  covariance 2^(-1)
  decomp_type pca
CubQMCSobolG (StoppingCriterion Object)
  abs_tol 0.001
  rel_tol 0
```
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 \texttt{qp.CubMCG} \cite{14} and \texttt{qp.CubQMCLatticeG} \cite{21}. The LD sequences provide the desired answer much more efficiently.

\begin{verbatim}
n_init = 2^{10}
n_max = 2^{35}
LDTransformData ( AccumulateData Object )
n_total = 2^{13}
solution = 1.808
error_bound = 6.37e-04
time_integrate = 0.009
\end{verbatim}

The second output of the stopping criterion provides helpful diagnostic information. This computation requires $n = 2^{13}$ Sobol’ points and 0.009 seconds to complete. The error bound is 0.000637, which falls below the absolute tolerance.

QMC, which uses LD sequences, is touted as providing substantially greater computational efficiency compared to IID MC. Figure 6 compares the time and sample sizes needed to compute the 5-dimensional Keister integral (10) using IID sequences and LD lattice sequences. Consistent with what is stated in Section 1, the error of IID MC is $O(n^{-1/2})$, which means that the time and sample size to obtain an absolute error tolerance of $\varepsilon$ is $O(\varepsilon^{-2})$. By contrast, the error of QMC using LD sequences is $O(n^{-1+\varepsilon})$, which implies $O(\varepsilon^{-1-\varepsilon})$ times and sample sizes. We see that 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 payoff of this option is the positive difference between the strike price, $K$, averaged over the time horizon:
payoff($S$) = max \[ \left( \frac{1}{2d} \sum_{j=1}^{d} (S_{j-1} + S_j) - K, 0 \right) \]. \[ S = (S_0, \ldots, S_d). \]

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. A basic model for asset prices is a geometric Brownian motion,

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

where $\Sigma$ is defined in (8), $r$ is the interest rate, $\sigma$ 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,

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

The following code utilizes QMCPy’s Asian option Integrand object to approximate the Asian call option for a particular choice of the parameters.

```python
>>> payoff = qp.AsianOption(qp.Sobol(52), # weekly monitoring
... start_price = 100,
... strike_price = 120,
... volatility = 0.5,
... interest_rate = 0,
... t_final = 1,
... call_put = "call")
>>> qmc_stop_crit = qp.CubQMCSobolG(payoff,abs_tol=0.001)
>>> price, data = qmc_stop_crit.integrate()
>>> print(f"Option price = ${price:.3f} using {data.time_integrate:.3f} seconds, {data.n_total:.2e} samples")
Option price = $5.194 using 1.309 seconds, 1.31e+05 samples
```

Because this Integrand object has the built-in Brownian motion TrueMeasure, one only need provide the LD sampler.

Out of the money option price calculations can be sped up by adding an upward drift to the Brownian motion. The upward drift produces more in the money paths and also reduces the variation or variance of the final integrand, $f$. This is a form of importance sampling. Using a Brownian motion without drift we get

```python
>>> payoff = qp.AsianOption(qp.Sobol(52),
... start_price = 100, strike_price = 200)
>>> qmc_stopper = qp.CubQMCsobolG(payoff,abs_tol=0.001)
>>> price, data = qmc_stopper.integrate()
>>> print(f"Option price = ${price:.4f} using {data.time_integrate:.3f} seconds, n = {data.n_total:.2e} samples")
Option price = $0.1755 using 1.375 seconds, 1.31e+05 samples
```

Adding the upward drift gives us the answer faster:
The choice of a good drift is an art. The improvement in time is less than improvement in \( n \) because the integrand is more expensive to compute when the drift is employed. Referring to (11), in the case of no drift, the \( \lambda \) corresponds to the density for the discrete Brownian motion, and the variable transformation \( \Psi \) is chosen so that \( f(x) = g(\Psi(x)) \). However, in the case of a drift, the integrand becomes \( f_{\text{IS}}(x) = g(\Psi_{\text{IS}}(x))\lambda(\Psi_{\text{IS}}(x))/\lambda_{\text{IS}}(\Psi_{\text{IS}}(x)) \), which requires more computation time per integrand value.

## 8 Under the Hood

In this section, we look at the inner workings of QMCPY and point out features we hope will benefit the community of QMC researchers and practitioners. We also highlight important idiosyncrasies of QMC methods and how QMCPY addresses these challenges. For details, readers should refer to the QMCPy documentation [4].

### 8.1 LD Sequences

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

Two popular LD sequences are integration lattices and digital nets which we previously outlined in Table 1. These LD generators are comprised of two parts: the static generating vectors \( Z_1, Z_2, Z_3, \ldots \in [0,1)^d \) and the callable generator function. By default, QMCPy provides a number of high quality generating vectors for users to choose from. For instance, the default ordinary lattice vector was constructed by Cools, Kuo, and Nuyens [5] to support 3600 dimensions and \( 2^{20} \) samples. However, users who require more samples but fewer dimensions may switch to a generating vector constructed using...
LatNet Builder [26, 27] to support 750 dimensions and $2^{24}$ samples. Moreover, the `qp.Lattice` and `qp.DigitalNet` objects allow users to input their own generating vectors to produce highly customized sequences. To find such vectors, we recommend using LatNet Builder’s construction routines as the results can be easily parsed into a QMCPy-compatible format.

Along with the selection of a generating vector, QMCPy’s low discrepancy sequence routines expose a number of other customization parameters. For instance, the lattice generator extends the Magic Point Shop [31] to support either linear or natural ordering. Digital sequences permit either standard or Graycode ordering and maybe randomized via a digital shift optionally combined with a linear scrambling [29]. Halton sequences may be randomized via the routines of either Owen [34] or Hofert and Lemieux [18].

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, digital nets and 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 $j^{th}$ 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, unrasteredized 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 of the 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 [33].
8.3 Transformations

The transformation \( \Psi \) connects a DiscreteDistribution and TrueMeasure. So far, we have assumed the DiscreteDistribution mimics a \( U[0,1]^d \) distribution with PDF \( \varrho(x) = 1 \). However, it may be advantageous to utilize a DiscreteDistribution that mimics a different distribution. For instance, QMCPy includes a DiscreteDistribution corresponding to an IID standard Gaussian sequence. It may be transformed to mimic a general Gaussian distribution TrueMeasure while avoiding the inverse CDF transform required for nodes mimicking \( U[0,1]^d \).

Suppose we have a DiscreteDistribution mimicking density \( \varrho \) supported on \( \mathcal{X} \). Then using the variable transformation \( \Psi \),

\[
\mu = \int_T g(t) \lambda(t) \, dt = \int_{\mathcal{X}} f(x) \varrho(x) \, dx \quad \text{for} \quad f(x) = g(\Psi(x)) \frac{\lambda(\Psi(x))}{\varrho(x)} |\Psi'(x)|,
\]

which generalizes (9). QMCPy also includes support for composed variable transformations. Suppose that the variable transformation is a composition of several transformations: \( \Psi = \Psi_1 \circ \Psi_{L-1} \circ \cdots \circ \Psi_1 \) as in (11). Here, \( \Psi_l : \mathcal{X}_{l-1} \to \mathcal{X}_l \), \( \mathcal{X}_0 = \mathcal{X} \), and \( \mathcal{X}_L = \mathcal{T} \) so that the transformations are compatible with the DiscreteDistribution and TrueMeasure. Let \( \Psi_l = \Psi_l \circ \Psi_{l-1} \circ \cdots \circ \Psi_1 \) denote the composition of the first \( l \) transforms and assume that \( \Psi_0(x) = x \), the identity transform. Then we may write \( \mu = \int_{\mathcal{X}} f(x) \varrho(x) \, dx \) for

\[
f(x) = g(\Psi_L(x)) \frac{\lambda(\Psi_L(x))}{\varrho(x)} \prod_{l=1}^L |\Psi'_l(\Psi_{l-1}(x))|.
\]

It is often the case that \( \Psi_l \) is chosen such that \( \Psi_l(X) \) is stochastically equivalent to a random variable with density \( \lambda_l \) on sample space \( \mathcal{X}_l \). This implies

\[
\varrho_l(x) = \lambda_l(\Psi_l(x)) |\Psi'_l(x)|
\]

so that

\[
f(x) = g(\Psi_L(x)) \frac{\lambda(\Psi_L(x))}{\varrho(x)} \prod_{l=1}^L \frac{\varrho_l(\Psi_{l-1}(x))}{\lambda_l(\Psi_l(x))}.
\]

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

| Integrand   | $L$ | $\lambda_1$ | $\Psi_1$ | $\lambda_2$ | $\Psi_2$ | $f$ |
|-------------|-----|-------------|----------|-------------|----------|-----|
| $K$         | 1   | $\mathcal{N}(0,1/2)$ | (7b) | $g(\Psi_1(\cdot))$ | | | |
| K_gauss     | 1   | $\mathcal{N}(0,3/4)$ | (7b) | $g(\Psi_1(\cdot))$ | $\frac{\lambda(\Psi_1(\cdot))}{\lambda_1(\Psi_1(\cdot))}$ | | |
| K_gauss_kuma| $\text{Kum}$ | $\mathcal{F}_{\text{Kum}}^{-1}$ | (7b) | $g(\Psi_1(\cdot))$ | $\frac{\lambda(\Psi_2(\Psi_1(\cdot)))\lambda(\Psi_2(\Psi_1(\cdot)))}{\lambda_1(\Psi_1(\cdot))\lambda_2(\Psi_2(\Psi_1(\cdot)))}$ | | |

Here Kum denotes the multivariate Kumaraswamy distribution with independent marginals, and $\mathcal{F}_{\text{Kum}}^{-1}$ denotes the element-wise inverse cumulative distribution function. The following code evaluates the Keister integral (10) for $d = 1$ and error tolerance $\varepsilon = 5E-8$. The timings for each of these different integrands are displayed.

```python
>>> sobol = qp.Sobol(1)
>>> K = qp.Keister(sobol)
>>> K_gauss = qp.Keister(
...    qp.Gaussian(sobol,covariance=.75))
>>> K_gauss_kuma = qp.Keister(
...    qp.Gaussian(
...        qp.Kumaraswamy(sobol,a=.8,b=.8)))
>>> for this_K in [K, K_gauss, K_gauss_kuma]:
...    stopper = qp.CubQMCSobolG(this_K, abs_tol=5e-8)
...    sol, data = stopper.integrate()
...    print("Time: % .3f" % data.time_integrate)
Time: 2.023
Time: 0.350
Time: 0.071
```

Successful importance sampling places more points where the original integrand, $g$, varies more. Equivalently, successful importance sampling makes the transformed integrand, $f$, flatter. The shorter execution times correspond to flatter integrands, as illustrated in Figure 7. The above example uses $d = 1$ to facilitate the plot in Figure 7, however, the same example works for arbitrary dimensions.

Automatic Bayesian stopping criteria choose $n$ needed to satisfy the error tolerance via a Bayesian credible interval for $\mu$ [19, 20]. These are also options in QMCPy.
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.

9 Further Work

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

Multi-level (quasi-)Monte Carlo (ML(Q)MC) methods make possible the computation of expectations of functionals of stochastic differential equations and partial differential equations with random coefficients. Such use cases appear in quantitative finance and in geophysical applications. QMCPy’s ML(Q)MC’s capability is rudimentary, but under active development.

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

Recently, some QMC experts have focused on developing LD generators for Python. Well established packages such as SciPy [37] and PyTorch [35] are actively working to develop QMC modules that support numerous LD sequences and related functionalities. When these modules are released to the public, we plan to integrate the routines as optional backends for QMCPy’s LD generator objects. 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 DiscreteDistribution 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.

Control variates can be useful for QMC, as well as for IID MC [17]. These should be incorporated into QMCPy in a seamless way.

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

Acknowledgements The authors would like to thank the organizers for a wonderful MCQMC 2020. We also thank the referees for their many helpful suggestions. This work is supported in part by SigOpt and National Science Foundation grant DMS-1522687.

References

1. Burkhardt, J.: Various software (2020). URL http://people.sc.fsu.edu/~jburkardt/
2. Choi, S.C.T., Ding, Y., Hickernell, F.J., Jiang, L., Jiménez Rugama, L.A., Li, D., Jagadeeswaran, R., Tong, X., Zhang, K., Zhang, Y., Zhou, X.: GAIL: Guaranteed Automatic Integration Library (versions 1.0–2.3.1). MATLAB software, http://gailgithub.github.io/GAIL_Dev/ (2020). DOI 10.5281/zenodo.4018190
3. Choi, S.C.T., Hickernell, F.J., Jagadeeswaran, R., McCourt, M., Sorokin, A.: QMCPy: A quasi-Monte Carlo Python library. https://qmcsoftware.github.io/QMCSoftware/ (2020). DOI 10.5281/zenodo.3964489
4. Choi, S.C.T., Hickernell, F.J., Jagadeeswaran, R., McCourt, M.J., Sorokin, A.: QMCPy documentation (2020). URL https://qmcpy.readthedocs.io/en/latest/
5. Cools, R., Kuo, F.Y., Nuyens, D.: Constructing embedded lattice rules for multivariate integration. SIAM Journal on Scientific Computing 28(6), 2162–2188 (2006). DOI 10.1137/06065074X
6. 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. Springer-Verlag, Berlin (2016)
7. Dick, J., Kuo, F., Sloan, I.H.: High dimensional integration — the Quasi-Monte Carlo way. Acta Numer. 22, 133–288 (2013). DOI 10.1017/S0962492913000044
8. Giles, M.: Multi-level (quasi-)Monte Carlo software (2020). URL https://people.maths.ox.ac.uk/gilesm/mlmc/
9. 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 R’io, J.F., Wiebe, M., Peterson, P., G’erard-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). DOI 10.1038/s41586-020-2649-2
10. Hickernell, F.J., Sorokin, A.: Quasi-Monte Carlo (QMC) software in QMCPy Google Colaboratory notebook (2020). URL http://tinyurl.com/QMCPyTutorial
11. Hickernell, F.J., Sorokin, A.: Quasi-Monte Carlo (QMC) software in QMCPy Google Colaboratory notebook for MCQMC2020 article (2020). URL https://tinyurl.com/QMCPyArticle2021
12. Hickernell, F.J.: A generalized discrepancy and quadrature error bound. Math. Comp. 67, 299–322 (1998). DOI 10.1090/S0025-5718-98-00894-1
13. Hickernell, F.J.: Goodness-of-fit statistics, discrepancies and robust designs. Statist. Probab. Lett. 44, 73–78 (1999)
14. Hickernell, F.J., Jiang, L., Liu, Y., Owen, A.B.: Guaranteed conservative fixed width confidence intervals via Monte Carlo sampling. In: J. Dick, F.Y. Kuo, G.W. Peters, I.H. Sloan (eds.) Monte Carlo and Quasi-Monte Carlo Methods 2012, Springer Proceedings in Mathematics and Statistics, vol. 65, pp. 105–128. Springer-Verlag, Berlin (2013). DOI 10.1007/978-3-642-41095-6
15. Hickernell, F.J., Jiménez Rugama, Ll.A.: Reliable adaptive cubature using digital sequences. In: Cools and Nuyens [6], pp. 367–383. ArXiv:1410.8615 [math.NA]
16. Hickernell, F.J., Jiménez Rugama, Ll.A., Li, D.: Adaptive quasi-Monte Carlo methods for cubature. In: J. Dick, F.Y. Kuo, H. Woźniakowski (eds.) Contemporary Computational Mathematics — a celebration of the 80th birthday of Ian Sloan, pp. 597–619. Springer-Verlag (2018). DOI 10.1007/978-3-319-72456-0
17. Hickernell, F.J., Lemieux, C., Owen, A.B.: Control variates for quasi-Monte Carlo. Statist. Sci. 20, 1–31 (2005). DOI 10.1214/088342304000000468
18. Hofert, M., Lemieux, C.: qrng R package (2017). URL https://cran.r-project.org/web/packages/qrng/qrng.pdf
19. Jagadeeswaran, R., Hickernell, F.J.: Fast automatic Bayesian cubature using lattice sampling. Stat. Comput. 29, 1215–1229 (2019). DOI 10.1007/s11222-019-09895-9
20. Jagadeeswaran, R., Hickernell, F.J.: Fast automatic Bayesian cubature using Sobol’ sampling (2021+)
21. Jiménez Rugama, Ll.A., Hickernell, F.J.: Adaptive multidimensional integration based on rank-1 lattices. In: Cools and Nuyens [6], pp. 407–422. ArXiv:1411.1966
22. Keister, B.D.: Multidimensional quadrature algorithms. Computers in Physics 10, 119–122 (1996). DOI 10.1063/1.168565
23. Kucherenko, S.: BRODA (2020). URL https://www.broda.co.uk/index.html
24. Kuo, F.Y., Nuyens, D.: Application of quasi-Monte Carlo methods to elliptic PDEs with random diffusion coefficients — a survey of analysis and implementation. Found. Comput. Math. 16, 1631–1696 (2016). URL https://people.cs.kuleuven.be/~dirk.nuyens/qmc4pde/
25. L’Ecuyer, P.: SSJ: Stochastic Simulation in Java (2020). URL https://github.com/umontreal-simul/ssj
26. L’Ecuyer, P., Marion, P., Godin, M., Puchhammer, F.: A tool for custom construction of QMC and RQMC point sets. In: E. Arnaud, M. Giles, A. Keller (eds.) Monte Carlo and Quasi-Monte Carlo Methods: MCQMC, Oxford, 2020 (2020+)
27. L’Ecuyer, P., Munger, D.: LatNet builder (2018). URL https://umontreal-simul.github.io/latnetbuilder/
28. Marelli, S., Sudret, B.: UQLab: A framework for uncertainty quantification in MATLAB. In: The 2nd International Conference on Vulnerability and Risk Analysis and Management (ICVRAM 2014), pp. 2554–2563. ASCE Library (2014). URL https://www.uqlab.com
29. Matoušek, J.: On the $L_2$-discrepancy for anchored boxes. J. Complexity 14, 527–556 (1998)
30. Niederreiter, H.: Random Number Generation and Quasi-Monte Carlo Methods. CBMS-NSF Regional Conference Series in Applied Mathematics. SIAM, Philadelphia (1992)
31. Nuyens, D.: Magic point shop (2017). URL https://people.cs.kuleuven.be/~dirk.nuyens/qmc-generators/
32. OpenTURNS Developers: An open source initiative for the Treatment of Uncertainties, Risks ‘N Statistics (2020). URL http://www.openturns.org
33. Owen, A.B.: On dropping the first Sobol’ point (2020). ArXiv:2008.08051 [math.NA]
34. Owen, A.B.: Randomized Halton sequences in R (2020). URL http://statweb.stanford.edu/~owen/code/
35. PyTorch Developers: PyTorch (2020). URL https://pytorch.org
36. Robbe, P.: Low discrepancy sequences in Julia (2020). URL https://github.com/PieterjanRobbe/QMC.jl
37. SciPy Developers: SciPy Ecosystem (2018). URL www.scipy.org
38. The MathWorks, Inc.: MATLAB R2020b. Natick, MA (2020)