qmcpy

Release 0.1

Fred J. Hickernell, Aleksei Sorokin, Sou-Cheng T. Choi

Dec 24, 2019

CONTENTS

CHAPTER

ONE

ABOUT OUR QMC SOFTWARE COMMUNITY

Contents

build passing

1.1 Quasi-Monte Carlo Community Software

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

1.1.1 Citation

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

Fred J. Hickernell, Sou-Cheng T. Choi, and Aleksei Sorokin, “QMC Community Software.” Python software, 2019. Work in progress. Available from https://github.com/QMCSoftware/QMCSoftware

1.1.2 Developers

- Sou-Cheng T. Choi
- Fred J. Hickernell
- Aleksei Sorokin

1.1.3 Contributors

- Michael McCourt

1.1.4 Acknowledgment

We thank Dirk Nuyens for fruitful discussions related to Magic Point Shop.

1.1.5 References

[1] F.Y. Kuo & D. Nuyens. "Application of quasi-Monte Carlo methods to elliptic PDEs with random diffusion coefficients - a survey of analysis and implementation",Foundations of Computational Mathematics, 16(6):1631-1696, 2016. (springer link, arxiv link)

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

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

[4] Sou-Cheng T. Choi, "MINRES-QLP Pack and Reliable Reproducible Research via Supportable Scientific Software," Journal of Open Research Software, Volume 2, Number 1, e22, pp. 1-7, 2014.

[5] Sou-Cheng T. Choi and Fred J. Hickernell, "IIT MATH-573 Reliable Mathematical Software" [Course Slides], Illinois Institute of Technology, Chicago, IL, 2013. Available from http://gailgithub.github.io/GAIL_Dev/

[6] Daniel S. Katz, Sou-Cheng T. Choi, Hilmar Lapp, Ketan Maheshwari, Frank Loffler, Matthew Turk, Marcus D. Hanwell, Nancy Wilkins-Diehr, James Hetherington, James Howison, Shel Swenson, Gabrielle D. Allen, Anne C. Elster, Bruce Berriman, Colin Venters, "Summary of the First Workshop On Sustainable Software for Science: Practice and Experiences (WSSSPE1)," Journal of Open Research Software, Volume 2, Number 1, e6, pp. 1-21, 2014.

[7] Fang, K.-T., & Wang, Y. (1994). Number-theoretic Methods in Statistics. London, UK: CHAPMAN & HALL

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

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

[10] Kai-Tai Fang and Yuan Wang, Number-theoretic Methods in Statistics, Chapman & Hall, London, 1994.

[11] Fred J. Hickernell and Lluis Antoni Jimenez Rugama, "Reliable adaptive cubature using digital sequences", Monte Carlo and Quasi-Monte Carlo Methods: MCQMC, Leuven, Belgium, April 2014 (R. Cools and D. Nuyens, eds.), Springer Proceedings in Mathematics and Statistics, vol. 163, Springer-Verlag, Berlin, 2016, arXiv:1410.8615 [math.NA], pp. 367-383.

Contents

1.2 Python 3 Library of QMC Software

1.2.1 QMCPy

Package of main components

- Integrand classes
- True Measure classes
- Discrete Distribution classes
- Stopping Criterion classes
- Accumulate Data classes
- Third Party contributed classes
- integrate function

1.2.2 workouts

Example uses of QMCPy package

1.2.3 test

Sets of long and short unittests

1.2.4 outputs

Logs and figures generated by workouts

1.2.5 demos

Example use of QMCPy as an independent package

1.2.6 sphinx

Automated project documentation is compiled with Sphinx and is available at the following websites: * GitHub * Read the Docs

1.2.7 Installation

```
pip install qmcpy
```

A virtual environment is recommended for developers/contributors Ensure *.../python_prototypes/* is in your path Install dependencies with

```
pip install requirements.txt
```

Contents

1.3 QMCPy

1.3.1 Integrand

The function to integrate *Abstract class with concrete implementations*

- Linear: $y_i = \sum_{j=0}^{d-1}(x_{ij})$
- Keister: $y_i = \pi^{d/2} * \cos(||x_i||_2)$
- Asian Call
  - $S_i(t_j) = S(0)e^{(r-\frac{\sigma^2}{2})t_j + \sigma\mathcal{B}(t_j)}$
  - discounted call payoff $= max(\frac{1}{d}\sum_{j=0}^{d-1} S(jT/d) - K)\ ,\ 0)$
  - discounted put payoff $= max(K - \frac{1}{d}\sum_{j=0}^{d-1} S(jT/d))\ ,\ 0)$

1.3.2 True Measure

General measure used to define the integrand *Abstract class with concrete implementations*

- Uniform: $\mathcal{U}(a,b)$
- Gaussian: $\mathcal{N}(\mu,\sigma^2)$
- Brownian Motion: $\mathcal{B}(t_j) = B(t_{j-1}) + Z_j\sqrt{t_j - t_{j-1}}$ for $Z_j \sim \mathcal{N}(0,1)$

1.3.3 Discrete Distribution

Sampling nodes iid or lds (low-discrepancy sequence) *Abstract class with concrete implementations*

- IID Standard Uniform: $x_j \overset{iid}{\sim} \mathcal{U}(0,1)$
- IID Standard Gaussian: $x_j \overset{iid}{\sim} \mathcal{N}(0,1)$
- Lattice (base 2): $x_j \overset{lds}{\sim} \mathcal{U}(0,1)$
- Sobol (base 2): $x_j \overset{lds}{\sim} \mathcal{U}(0,1)$

1.3.4 Stopping Criterion

The stopping criterion to determine sufficient approximation *Abstract class with concrete implementations* Central Limit Theorem (CLT) $\hat{\mu}_n = \overline{Y}_n \approx \mathcal{N}(\mu, \frac{\sigma^2}{n})$ $\mathbb{P}[\hat{\mu}_n - \frac{\mathcal{Z}_{\alpha/2}\hat{\sigma}_n}{\sqrt{n}} \leq \mu \leq \hat{\mu}_n + \frac{\mathcal{Z}_{\alpha/2}\hat{\sigma}_n}{\sqrt{n}}] \approx 1-\alpha$

- CLT for $x_i \sim$ iid
- CLT Repeated for $\{x_{r,i}\}_{r=1}^{R} \sim$ lds

1.3.5 Accumulate Data Class

Stores data values of corresponding stopping criterion procedure *Abstract class with concrete implementations*

- Mean Variance Data (Controlled by CLT)
- Mean Variance Repeated Data (Controlled by CLT Repeated)

1.3.6 Integrate Method

Repeatedly samples the integrand at nodes generated by the discrete distribution and transformed to mimic the integrand's true measure until the Stopping Criterion is met *Function with arguments:*

- Integrand object
- True Measure object
- Discrete Distribution object
- Stopping Criterion object

Contents

1.4 Tests

1.4.1 Fast unittests

Quickly check functionality Run all in < 1 second

```
python -m unittest discover -s test/fasttests
```

1.4.2 Long unittests

Call workout functions Runs all in < 10 seconds

```
python -m unittest discover -s test/longtests
```

CHAPTER TWO

LICENSE

CHAPTER

THREE

QMCPY DOCUMENTATION

3.1 Integration Method

Main driver function for QMCPy.

`qmcpy.integrate.integrate`(*integrand*, *true_measure*, *discrete_distrib=None*, *stopping_criterion=None*)

Specify and compute integral of $f(\boldsymbol{x})$ for $\boldsymbol{x} \in \mathcal{X}$.

Parameters

- **integrand** (*Integrand*) – an object from class Integrand. If None (default), sum of two variables defined on unit square is used.
- **true_measure** (*TrueMeasure*) – an object from class TrueMeasure. If None (default), standard uniform distribution is used.
- **discrete_distrib** (*DiscreteDistribution*) – an object from class DiscreteDistribution. If None (default), IID standard uniform distribution is used.
- **stopping_criterion** (*StoppingCriterion*) – an object from class StoppingCriterion. If None (default), criterion based on central limit theorem with absolute tolerance equal to 0.01 is used.

Returns

tuple containing:

solution (`float`): estimated value of the integral

data (`AccumData`): input data and information such as number of sampling points and run time used to obtain solution

Return type tuple

3.2 Integrand Class

3.2.1 Asian Call Option Payoff

Definition for class AsianCall, a concrete implementation of Integrand

class `qmcpy.integrand.asian_call.AsianCall`(*bm_measure*, *volatility=0.5*, *start_price=30*, *strike_price=25*, *interest_rate=0*, *mean_type='arithmetic'*)

Specify and generate payoff values of an Asian Call option

__init__(*bm_measure*, *volatility=0.5*, *start_price=30*, *strike_price=25*, *interest_rate=0*, *mean_type='arithmetic'*)

Initialize AsianCall Integrand's'

Parameters

- **bm_measure** (*TrueMeasure*) – A BrownianMotion Measure object
- **volatility** (*float*) – sigma, the volatility of the asset
- **start_price** (*float*) – S(0), the asset value at t=0
- **strike_price** (*float*) – strike_price, the call/put offer
- **interest_rate** (*float*) – r, the annual interest rate
- **mean_type** (*string*) – 'arithmetic' or 'geometric' mean

g(*x*)

Original integrand to be integrated

Parameters **x** – nodes, $\boldsymbol{x}_{\mathfrak{u},i} = i^{\mathrm{th}}$ row of an $n \cdot |\mathfrak{u}|$ matrix

Returns $n \cdot p$ matrix with values $f(\boldsymbol{x}_{\mathfrak{u},i}, \mathbf{c})$ where if $\boldsymbol{x}'_i = (x_{i,\mathfrak{u}}, \mathbf{c})_j$, then $x'_{ij} = x_{ij}$ for $j \in \mathfrak{u}$, and $x'_{ij} = c$ otherwise

get_discounted_payoffs(*stock_path*, *dimension*)

Calculate the discounted payoff from the stock path

stock_path (ndarray): option prices at monitoring times dimension (int): number of dimensions

3.2.2 Keister Function

Definition for class Keister, a concrete implementation of Integrand

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

Specify and generate values $f(\boldsymbol{x}) = \pi^{d/2} \cos(\|\boldsymbol{x}\|)$ for $\boldsymbol{x} \in \mathbb{R}^d$.

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

Reference:

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

__init__(*dimension*)

Parameters **dimension** (*ndarray*) – dimension(s) of the integrand(s)

g(*x*)

Original integrand to be integrated

Parameters **x** – nodes, $\boldsymbol{x}_{\mathfrak{u},i} = i^{\mathrm{th}}$ row of an $n \cdot |\mathfrak{u}|$ matrix

Returns $n \cdot p$ matrix with values $f(\boldsymbol{x}_{\mathfrak{u},i}, \mathbf{c})$ where if $\boldsymbol{x}'_i = (x_{i,\mathfrak{u}}, \mathbf{c})_j$, then $x'_{ij} = x_{ij}$ for $j \in \mathfrak{u}$, and $x'_{ij} = c$ otherwise

3.2.3 A Linear Function

Definition for class Linear, a concrete implementation of Integrand

class qmcpy.integrand.linear.**Linear**(*dimension*)

Specify and generate values $f(\boldsymbol{x}) = \sum_{i=1}^{d} x_i$ for $\boldsymbol{x} = (x_1, \ldots, x_d) \in \mathbb{R}^d$

`__init__`(*dimension*)

Parameters **dimension** (*ndarray*) – dimension(s) of the integrand(s)

`g`(x)

Original integrand to be integrated

Parameters **x** – nodes, $\boldsymbol{x}_{\mathfrak{u},i} = i^{\mathtt{th}}$ row of an $n \cdot |\mathfrak{u}|$ matrix

Returns $n \cdot p$ matrix with values $f(\boldsymbol{x}_{\mathfrak{u},i}, \mathbf{c})$ where if $\boldsymbol{x}'_i = (x_{i,\mathfrak{u}}, \mathbf{c})_j$, then $x'_{ij} = x_{ij}$ for $j \in \mathfrak{u}$, and $x'_{ij} = c$ otherwise

3.2.4 Quick Construct for Function

Definition for class QuickConstruct, a concrete implementation of Integrand

class `qmcpy.integrand.quick_construct.QuickConstruct`(*dimension*, *custom_fun*)

Specify and generate values of a user-defined function

`__init__`(*dimension*, *custom_fun*)

Initialize custom Integrand

Parameters

- **dimension** (*ndarray*) – dimension(s) of the integrand(s)
- **custom_fun** (*int*) – a callable univariable or multivariate Python function that returns a real number.

Note: Input of the function:

x: nodes, $\boldsymbol{x}_{\mathfrak{u},i} = i^{\mathtt{th}}$ row of an $n \cdot |\mathfrak{u}|$ matrix

---

`g`(x)

Original integrand to be integrated

Parameters **x** – nodes, $\boldsymbol{x}_{\mathfrak{u},i} = i^{\mathtt{th}}$ row of an $n \cdot |\mathfrak{u}|$ matrix

Returns $n \cdot p$ matrix with values $f(\boldsymbol{x}_{\mathfrak{u},i}, \mathbf{c})$ where if $\boldsymbol{x}'_i = (x_{i,\mathfrak{u}}, \mathbf{c})_j$, then $x'_{ij} = x_{ij}$ for $j \in \mathfrak{u}$, and $x'_{ij} = c$ otherwise

3.3 Measure Class

Definitions of TrueMeasure Concrete Classes

class `qmcpy.true_measure.measures.BrownianMotion`(*dimension*, *time_vector=[array([0.250, 0.500, 0.750, 1.000])]*)

Brownian Motion Measure

`__init__`(*dimension*, *time_vector=[array([0.250, 0.500, 0.750, 1.000])]*)

Parameters

- **dimension** (*ndarray*) – dimension's' of the integrand's'
- **time_vector** (*list of ndarrays*) – monitoring times for the Integrand's'

class `qmcpy.true_measure.measures.Gaussian`(*dimension*, *mean=0*, *variance=1*)

Gaussian (Normal) Measure

__init__(*dimension*, *mean=0*, *variance=1*)

Parameters

- **dimension** (*ndarray*) – dimension's' of the integrand's'
- **mean** (*float*) – mu for Normal(mu,sigma^2)
- **variance** (*float*) – sigma^2 for Normal(mu,sigma^2)

class qmcpy.true_measure.measures.**Lebesgue**(*dimension*, *lower_bound=0.0*, *upper_bound=1*)

Lebesgue Uniform Measure

__init__(*dimension*, *lower_bound=0.0*, *upper_bound=1*)

Parameters **dimension** (*ndarray*) – dimension's' of the integrand's'

class qmcpy.true_measure.measures.**Uniform**(*dimension*, *lower_bound=0.0*, *upper_bound=1.0*)

Uniform Measure

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

Parameters

- **dimension** (*ndarray*) – dimension's' of the integrand's'
- **lower_bound** (*float*) – a for Uniform(a,b)
- **upper_bound** (*float*) – b for Uniform(a,b)

3.4 Discrete Distribution Class

This module implements mutiple subclasses of DiscreteDistribution.

class qmcpy.discrete_distribution.iid_generators.**IIDStdGaussian**(*rng_seed=None*)

Standard Gaussian

__init__(*rng_seed=None*)

Parameters **rng_seed** (*int*) – seed the random number generator for reproducibility

gen_dd_samples(*replications*, *n_samples*, *dimensions*)

Generate r nxd IID Standard Gaussian samples

Parameters

- **replications** (*int*) – Number of nxd matrices to generate (sample.size()[0])
- **n_samples** (*int*) – Number of observations (sample.size()[1])
- **dimensions** (*int*) – Number of dimensions (sample.size()[2])

Returns replications x n_samples x dimensions (numpy array)

class qmcpy.discrete_distribution.iid_generators.**IIDStdUniform**(*rng_seed=None*)

IID Standard Uniform

__init__(*rng_seed=None*)

Parameters **rng_seed** (*int*) – seed the random number generator for reproducibility

gen_dd_samples(*replications*, *n_samples*, *dimensions*)

Generate r nxd IID Standard Uniform samples

Parameters

- **replications** (*int*) – Number of nxd matrices to generate (sample.size()[0])
- **n_samples** (*int*) – Number of observations (sample.size()[1])
- **dimensions** (*int*) – Number of dimensions (sample.size()[2])

Returns replications x n_samples x dimensions (numpy array)

This module implements mutiple subclasses of DiscreteDistribution.

class qmcpy.discrete_distribution.lds_generators.**Lattice**(*rng_seed=None*)

Quasi-Random Lattice low discrepancy sequence (Base 2)

__init__(*rng_seed=None*)

Parameters **rng_seed** (*int*) – seed the random number generator for reproducibility

gen_dd_samples(*replications*, *n_samples*, *dimensions*, *scramble=True*)

Generate r nxd Lattice samples

Parameters

- **replications** (*int*) – Number of nxd matrices to generate (sample.size()[0])
- **n_samples** (*int*) – Number of observations (sample.size()[1])
- **dimensions** (*int*) – Number of dimensions (sample.size()[2])
- **scramble** (*bool*) – If true, random numbers are in unit cube, otherwise they are non-negative integers

Returns replications x n_samples x dimensions (numpy array)

class qmcpy.discrete_distribution.lds_generators.**Sobol**(*rng_seed=None*, *backend='Pytorch'*)

Quasi-Random Sobol low discrepancy sequence (Base 2)

__init__(*rng_seed=None*, *backend='Pytorch'*)

Parameters **rng_seed** (*int*) – seed the random number generator for reproducibility

gen_dd_samples(*replications*, *n_samples*, *dimensions*, *scramble=True*)

Generate r nxd Sobol samples

Parameters

- **replications** (*int*) – Number of nxd matrices to generate (sample.size()[0])
- **n_samples** (*int*) – Number of observations (sample.size()[1])
- **dimensions** (*int*) – Number of dimensions (sample.size()[2])
- **scramble** (*bool*) – If true, random numbers are in unit cube, otherwise they are non-negative integers

Returns replications x n_samples x dimensions (numpy array)

qmcpy.discrete_distribution.lds_generators.**randint**(*low*, *high=None*, *size=None*, *dtype='l'*)

Return random integers from *low* (inclusive) to *high* (exclusive).

Return random integers from the "discrete uniform" distribution of the specified dtype in the "half-open" interval [*low*, *high*). If *high* is None (the default), then results are from [0, *low*).

low [int or array-like of ints] Lowest (signed) integers to be drawn from the distribution (unless `high=None`, in which case this parameter is one above the *highest* such integer).

high [int or array-like of ints, optional] If provided, one above the largest (signed) integer to be drawn from the distribution (see above for behavior if `high=None`). If array-like, must contain integer values

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

dtype [dtype, optional] Desired dtype of the result. All dtypes are determined by their name, i.e., 'int64', 'int', etc, so byteorder is not available and a specific precision may have different C types depending on the platform. The default value is 'np.int'.

New in version 1.11.0.

out [int or ndarray of ints] *size*-shaped array of random integers from the appropriate distribution, or a single such random int if *size* not provided.

random.random_integers [similar to *randint*, only for the closed] interval [*low*, *high*], and 1 is the lowest value if *high* is omitted.

```
>>> np.random.randint(2, size=10)
array([1, 0, 0, 0, 1, 1, 0, 0, 1, 0]) # random
>>> np.random.randint(1, size=10)
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

Generate a 2 x 4 array of ints between 0 and 4, inclusive:

```
>>> np.random.randint(5, size=(2, 4))
array([[4, 0, 2, 1], # random
       [3, 2, 2, 0]])
```

Generate a 1 x 3 array with 3 different upper bounds

```
>>> np.random.randint(1, [3, 5, 10])
array([2, 2, 9]) # random
```

Generate a 1 by 3 array with 3 different lower bounds

```
>>> np.random.randint([1, 5, 7], 10)
array([9, 8, 7]) # random
```

Generate a 2 by 4 array using broadcasting with dtype of uint8

```
>>> np.random.randint([1, 3, 5, 7], [[10], [20]], dtype=np.uint8)
array([[ 8,  6,  9,  7], # random
       [ 1, 16,  9, 12]], dtype=uint8)
```

3.5 Data Class

Definition of MeanVarData, a concrete implementation of AccumData

class `qmcpy.accum_data.mean_var_data.`**`MeanVarData`**(*levels*, *n_init*)
Accumulated data for IIDDistribution calculations, and store the sample mean and variance of integrand values

`__init__`(*levels*, *n_init*)
Initialize data instance

Parameters

- **levels** (*int*) – number of integrands
- **n_init** (*int*) – initial number of samples

update_data (*integrand*, *true_measure*)
Update data

Parameters

- **integrand** (*Integrand*) – an instance of Integrand
- **true_measure** (*TrueMeasure*) – an instance of TrueMeasure

Returns None

Definition for MeanVarDataRep, a concrete implementation of AccumData

class qmcpy.accum_data.mean_var_data_rep.**MeanVarDataRep** (*levels*, *n_init*, *replications*)
Accumulated data Repeated Central Limit Stopping Criterion (CLTRep) calculations.

__init__ (*levels*, *n_init*, *replications*)
Initialize data instance

Parameters

- **levels** (*int*) – number of integrands
- **n_init** (*int*) – initial number of samples
- **replications** (*int*) – number of random nxm matrices to generate

update_data (*integrand*, *true_measure*)
Update data

Parameters

- **integrand** (*Integrand*) – an instance of Integrand
- **true_measure** (*TrueMeasure*) – an instance of TrueMeasure

Returns None

3.6 Stopping Criterion Class

Definition for CLT, a concrete implementation of StoppingCriterion

class qmcpy.stopping_criterion.clt.**CLT** (*discrete_distrib*, *true_measure*, *inflate=1.2*, *alpha=0.01*, *abs_tol=0.01*, *rel_tol=0*, *n_init=1024*, *n_max=10000000000.0*)
Stopping criterion based on the Central Limit Theorem (CLT)

__init__ (*discrete_distrib*, *true_measure*, *inflate=1.2*, *alpha=0.01*, *abs_tol=0.01*, *rel_tol=0*, *n_init=1024*, *n_max=10000000000.0*)

Parameters

- **discrete_distrib** –
- **true_measure** – an instance of DiscreteDistribution
- **inflate** – inflation factor when estimating variance
- **alpha** – significance level for confidence interval
- **abs_tol** – absolute error tolerance

- **rel_tol** – relative error tolerance
- **n_max** – maximum number of samples

stop_yet()
Determine when to stop

Definition for CLTRep, a concrete implementation of StoppingCriterion

class qmcpy.stopping_criterion.clt_rep.**CLTRep**(*discrete_distrib*, *true_measure*, *replications=16*, *inflate=1.2*, *alpha=0.01*, *abs_tol=0.01*, *rel_tol=0*, *n_init=32*, *n_max=1073741824*)
Stopping criterion based on var(stream_1_estimate, . . . , stream_16_estimate) < errorTol

__init__(*discrete_distrib*, *true_measure*, *replications=16*, *inflate=1.2*, *alpha=0.01*, *abs_tol=0.01*, *rel_tol=0*, *n_init=32*, *n_max=1073741824*)

Parameters

- **discrete_distrib** –
- **true_measure** (*DiscreteDistribution*) – an instance of DiscreteDistribution
- **replications** (*int*) – number of random nxm matrices to generate
- **inflate** (*float*) – inflation factor when estimating variance
- **alpha** (*float*) – significance level for confidence interval
- **abs_tol** (*float*) – absolute error tolerance
- **rel_tol** (*float*) – relative error tolerance
- **n_max** (*int*) – maximum number of samples

stop_yet()
Determine when to stop

3.7 Utilities

Meta-data and public utilities for qmcpy

Exceptions and Warnings thrown by qmcpy

exception qmcpy._util._exceptions_warnings.**DimensionError**
Class for raising error about dimension

exception qmcpy._util._exceptions_warnings.**DistributionCompatibilityError**
Class for raising error about incompatible distribution

exception qmcpy._util._exceptions_warnings.**DistributionGenerationError**
Class for raising error about parameter inputs to gen_dd_samples (method of a DiscreteDistribution)

exception qmcpy._util._exceptions_warnings.**DistributionGenerationWarnings**
Class for issuing warningssabout parameter inputs to gen_dd_samples (method of a DiscreteDistribution)

exception qmcpy._util._exceptions_warnings.**MaxSamplesWarning**
Class for issuing warning about using maximum number of data samples

exception qmcpy._util._exceptions_warnings.**MeasureCompatibilityError**
Class for raising error of incompatible measures

exception qmcpy._util._exceptions_warnings.**NotYetImplemented**
Class for raising error when a component has been implemented yet

exception qmcpy._util._exceptions_warnings.**ParameterError**
Class for raising error about input parameters

exception qmcpy._util._exceptions_warnings.**ParameterWarning**
Class for issuing warnings about unacceptable parameters

exception qmcpy._util._exceptions_warnings.**TransformError**
Class for raising error about transforming function to accommodate distribution

CHAPTER

FOUR

DEMOS

4.1 Welcome to QMCPy

4.1.1 Import

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

```
qmcpy 0.1
```

```
# Individual Imports
from qmcpy import integrate
from qmcpy.integrand import *
from qmcpy.true_measure import *
from qmcpy.discrete_distribution import *
from qmcpy.stopping_criterion import *
```

```
# Complete Import
from qmcpy import *
```

4.2 Important Notes

4.2.1 IID vs LDS

Low discrepancy sequences (LDS) such as lattice and Sobol are not independent like IID points - The below code generates 1 replication (squeezed out) of 4 samples of 2 dimensions

```
discrete_distrib = Lattice(rng_seed = 7)
x = discrete_distrib.gen_dd_samples(replications=1, n_samples=4, dimensions=2).
↪squeeze()
x
```

```
array([[ 0.625,  0.897],
       [ 0.125,  0.397],
       [ 0.875,  0.647],
       [ 0.375,  0.147]])
```

4.2.2 Multi-Dimensional Inputs

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

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

$$where : math : `[0,1]^d`istheunithypercubein : math : `\mathbb{R}^d`.$$

The integrand is defined everywhere except at $x = 0$ and hence the definite integral is also defined.

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

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

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

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

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

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

```
0.6309573444801932
```

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

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

```
1.001650000560437
```

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

```
norm(x)
```

```
1.0016486409914407
```

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

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

```
array([ 1.000,  0.010,  0.057])
```

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

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

```
array([ 1.000,  0.100,  0.238])
```

It is. Putting everything together, we have:

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

```
array([ 1.000,  0.631,  0.505])
```

We have got our proper function definition now.

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

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

```
dim = 1
integrand = QuickConstruct(dim, custom_fun=f)
sol, data = integrate(integrand, Uniform(dim))
print(data)
```

```
Solution: 0.6616
QuickConstruct (Integrand Object)
IIDStdUniform (Discrete Distribution Object)
    mimics          StdUniform
Uniform (True Measure Object)
    dimension       1
    a               0
    b               1
CLT (Stopping Criterion Object)
    abs_tol         0.010
    rel_tol         0
    n_max           10000000000
    inflate         1.200
    alpha           0.010
MeanVarData (AccumData Object)
    n               3166
    n_total         4190
    confid_int      [ 0.651  0.672]
    time_total      0.002
```

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

```
true_sol = 0.658582  # In WolframAlpha: Integral[x**Sqrt[x], {x,0,1}]
abs_tol = data.stopping_criterion.abs_tol
qmcpy_error = abs(true_sol - sol)
print(qmcpy_error < abs_tol)
```

```
True
```

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

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

```
dim = 2
integrand2 = QuickConstruct(dim, f)
sol2, data2 = integrate(integrand2, Uniform(dim))
print(data2)
```

```
Solution: 0.8244
QuickConstruct (Integrand Object)
IIDStdUniform (Discrete Distribution Object)
    mimics          StdUniform
Uniform (True Measure Object)
    dimension       2
    a               0
    b               1
CLT (Stopping Criterion Object)
    abs_tol         0.010
    rel_tol         0
    n_max           10000000000
    inflate         1.200
    alpha           0.010
MeanVarData (AccumData Object)
    n               5520
    n_total         6544
    confid_int      [ 0.814  0.834]
    time_total      0.002
```

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

```
true_sol2 = 0.827606  # In WolframAlpha:␣
↪Integral[Sqrt[x**2+y**2])**Sqrt[Sqrt[x**2+y**2]], {x,0,1}, {y,0,1}]
abs_tol2 = data2.stopping_criterion.abs_tol
qmcpy_error2 = abs(true_sol2 - sol2)
print(qmcpy_error2 < abs_tol2)
```

```
True
```

4.3 Integration Examples using QMCPy package

```
from qmcpy import *

from numpy import arange
```

4.3.1 Keister Example

Keister Integrand: - $y_i = \pi^{d/2} * \cos(||x_i||_2)$

Gaussian True Measure: - $\mathcal{N}(0, \frac{1}{2})$

Sobol Discrete Distribution: - $x_j \overset{lds}{\sim} \mathcal{U}(0,1)$

```
dim = 3
integrand = Keister(dim)
discrete_distrib = Sobol(rng_seed=7)
true_measure = Gaussian(dim, variance=1 / 2)
stopping_criterion = CLTRep(discrete_distrib, true_measure, abs_tol=.05)
_, data = integrate(integrand, true_measure, discrete_distrib, stopping_criterion)
print(data)
```

```
Solution: 2.1716
Keister (Integrand Object)
Sobol (Discrete Distribution Object)
    mimics          StdUniform
    rng_seed        7
    backend         pytorch
Gaussian (True Measure Object)
    dimension       3
    mu              0
    sigma           0.707
CLTRep (Stopping Criterion Object)
    abs_tol         0.050
    rel_tol         0
    n_max           1073741824
    inflate         1.200
    alpha           0.010
MeanVarDataRep (AccumData Object)
    n               128
    n_total         128
    confid_int      [ 2.164  2.179]
    time_total      0.007
    r               16
```

4.3.2 Asian Option Pricing Example

Single Level

Asian Call Option Integrand - $S_i(t_j) = S(0)e^{(r-\frac{\sigma^2}{2})t_j+\sigma\mathcal{B}(t_j)}$ - discounted put payoff $= max(K - \frac{1}{d}\sum_{j=0}^{d-1} S(jT/d))\ ,\ 0)$

Brownian Motion True Measure: - $\mathcal{B}(t_j) = B(t_{j-1}) + Z_j\sqrt{t_j - t_{j-1}}$ for $Z_j \sim \mathcal{N}(0,1)$

Lattice Discrete Distribution: - $x_j \overset{lds}{\sim} \mathcal{U}(0,1)$

```
time_vec = [arange(1 / 64, 65 / 64, 1 / 64)]
dim = [len(tv) for tv in time_vec]

discrete_distrib = Lattice(rng_seed=7)
true_measure = BrownianMotion(dim, time_vector=time_vec)
integrand = AsianCall(true_measure,
                volatility = .5,
                start_price = 30,
                strike_price = 25,
                interest_rate = .01,
                mean_type = 'geometric')
stopping_criterion = CLTRep(discrete_distrib, true_measure, abs_tol=.05)
```

(continues on next page)

(continued from previous page)

```
_, data = integrate(integrand, true_measure, discrete_distrib, stopping_criterion)
print(data)
```

```
Solution: 5.8356
AsianCall (Integrand Object)
    volatility      0.500
    start_price     30
    strike_price    25
    interest_rate   0.010
    mean_type       geometric
    exercise_time   1
Lattice (Discrete Distribution Object)
    mimics          StdUniform
    rng_seed        7
BrownianMotion (True Measure Object)
    dimension       64
    time_vector     [ 0.016  0.031  0.047 ...  0.969  0.984  1.000]
CLTRep (Stopping Criterion Object)
    abs_tol         0.050
    rel_tol         0
    n_max           1073741824
    inflate         1.200
    alpha           0.010
MeanVarDataRep (AccumData Object)
    n               2048
    n_total         2048
    confid_int      [ 5.833  5.838]
    time_total      0.276
    r               16
```

4.3.3 Asian Option Pricing Example

Multi-Level

$Y_0 = 0$

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

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

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

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

```
time_vec = [arange(1 / 4, 5 / 4, 1 / 4),
                 arange(1 / 16, 17 / 16, 1 / 16),
                 arange(1 / 64, 65 / 64, 1 / 64)]
dim = [len(tv) for tv in time_vec]

discrete_distrib = IIDStdGaussian(rng_seed=7)
true_measure = BrownianMotion(dim, time_vector=time_vec)
integrand = AsianCall(true_measure,
                volatility = .5,
                start_price = 30,
                strike_price = 25,
                interest_rate = .01,
```

(continues on next page)

(continued from previous page)

```
                mean_type = 'geometric')
stopping_criterion = CLT(discrete_distrib, true_measure, abs_tol=.05, n_max = 1e10)
_, data = integrate(integrand, true_measure, discrete_distrib, stopping_criterion)
print(data)
```

```
Solution: 5.8326
AsianCall (Integrand Object)
    volatility       [ 0.500  0.500  0.500]
    start_price      [30 30 30]
    strike_price     [25 25 25]
    interest_rate    [ 0.010  0.010  0.010]
    mean_type        ['geometric' 'geometric' 'geometric']
    exercise_time    [ 1.000  1.000  1.000]
IIDStdGaussian (Discrete Distribution Object)
    mimics           StdGaussian
BrownianMotion (True Measure Object)
    dimension        [ 4 16 64]
    time_vector      [array([ 0.250,  0.500,  0.750,  1.000])
                     array([ 0.062,  0.125,  0.188, ...,  0.875,  0.938,  1.000])
                     array([ 0.016,  0.031,  0.047, ...,  0.969,  0.984,  1.000])]
CLT (Stopping Criterion Object)
    abs_tol          0.050
    rel_tol          0
    n_max            10000000000
    inflate          1.200
    alpha            0.010
MeanVarData (AccumData Object)
    n                [ 239356.000  38466.000  6904.000]
    n_total          287798
    confid_int       [ 5.784  5.881]
    time_total       0.100
```

4.4 Scatter Plots of Samples

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

import matplotlib
%matplotlib inline
import matplotlib.pyplot as plt

SMALL_SIZE = 10
MEDIUM_SIZE = 12
BIGGER_SIZE = 14

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

(continues on next page)

(continued from previous page)

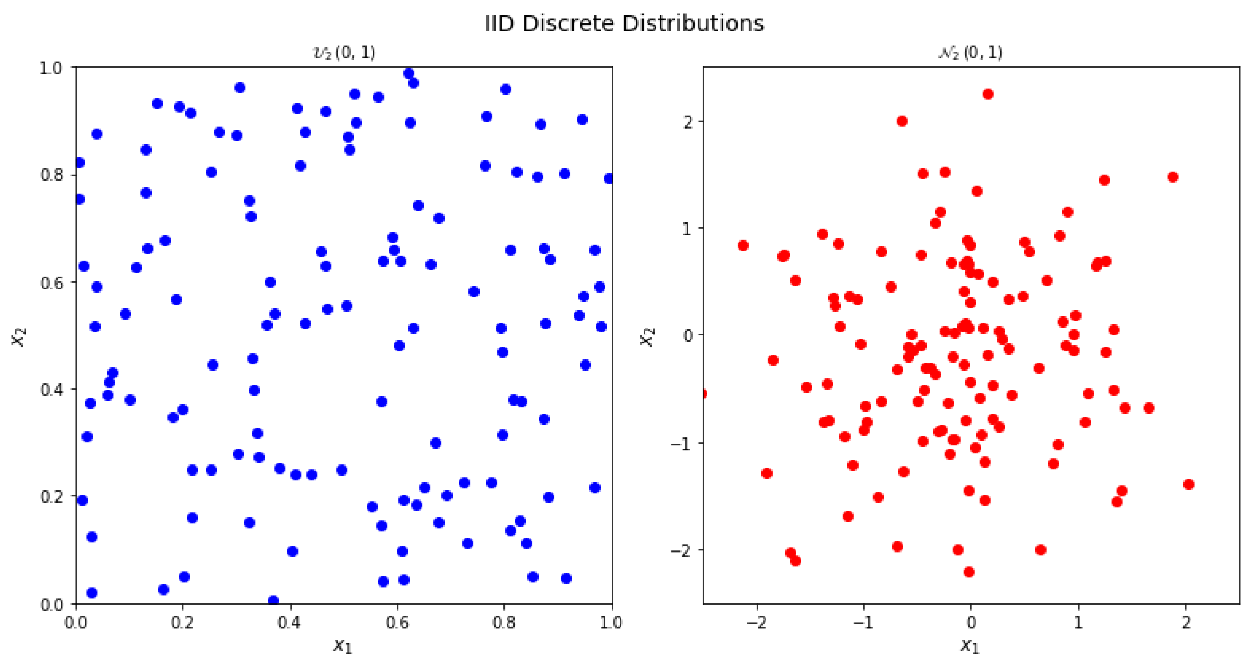
```
from qmcpy import *
```

```
n = 128
```

4.4.1 IID Samples

Visualize IID standard uniform and standard normal sampling points

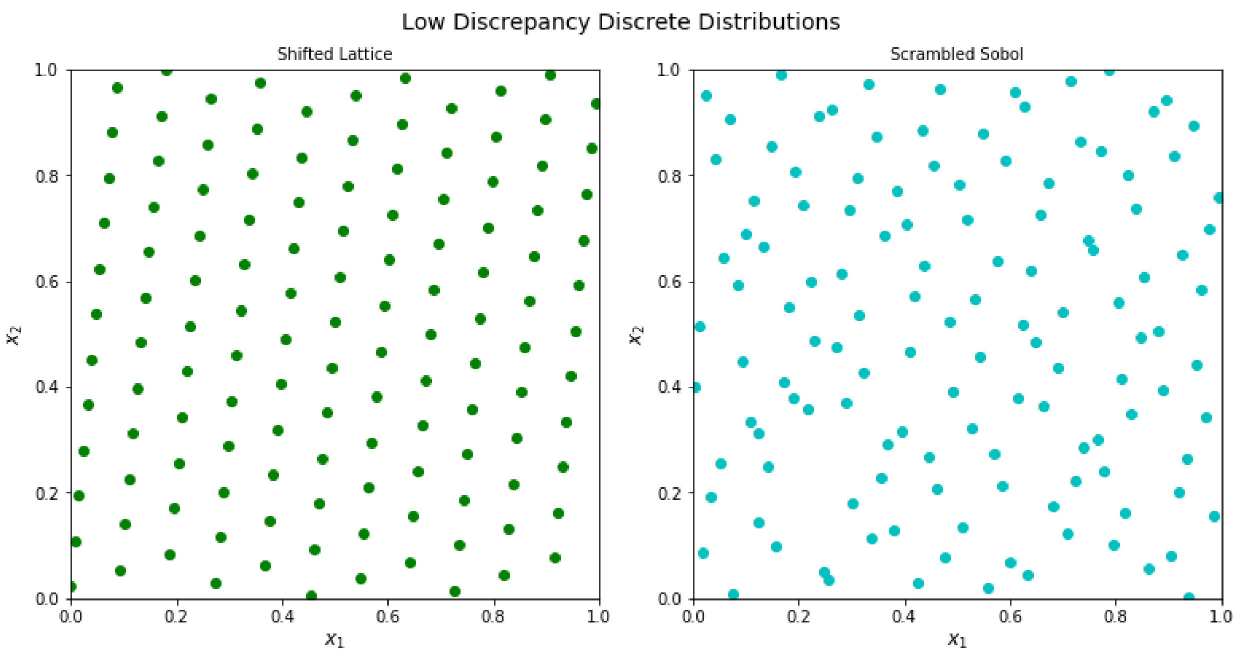
```
discrete_distribs = [IIDStdUniform(rng_seed=7), IIDStdGaussian(rng_seed=7)]
dd_names = ["$\\mathcal{U}_2\\,(0,1)$", "$\\mathcal{N}_2\\,(0,1)$"]
colors = ["b", "r"]
lims = [[0, 1], [-2.5, 2.5]]
fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(11, 6))
for i, (dd_obj, color, lim, dd_name) in enumerate(zip(discrete_distribs, colors, lims,
↪ dd_names)):
    samples = dd_obj.gen_dd_samples(1, n, 2).squeeze()
    ax[i].scatter(samples[:, 0], samples[:, 1], color=color)
    ax[i].set_xlabel("$x_1$")
    ax[i].set_ylabel("$x_2$")
    ax[i].set_xlim(lim)
    ax[i].set_ylim(lim)
    ax[i].set_aspect("equal")
    ax[i].set_title(dd_name)
fig.suptitle("IID Discrete Distributions")
plt.tight_layout()
fig.savefig("../outputs/sample_scatters/iid_dd.png", dpi=200)
```



4.4.2 LDS Samples

Visualize shifted lattice and scrambled Sobol sampling points

```
discrete_distribs = [Lattice(rng_seed=7), Sobol(rng_seed=7)]
dd_names = ["Shifted Lattice", "Scrambled Sobol"]
colors = ["g", "c"]
fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(11, 6))
for i, (dd_obj, color, dd_name) in \
        enumerate(zip(discrete_distribs, colors, dd_names)):
    samples = dd_obj.gen_dd_samples(1, n, 2).squeeze()
    ax[i].scatter(samples[:, 0], samples[:, 1], color=color)
    ax[i].set_xlabel("$x_1$")
    ax[i].set_ylabel("$x_2$")
    ax[i].set_xlim([0, 1])
    ax[i].set_ylim([0, 1])
    ax[i].set_aspect("equal")
    ax[i].set_title(dd_name)
fig.suptitle("Low Discrepancy Discrete Distributions")
plt.tight_layout()
fig.savefig("../outputs/sample_scatters/lds_dd.png", dpi=200)
```



4.4.3 Transform to the True Distribution

Transform our Discrete Distribution samples to mimic various True Distributions

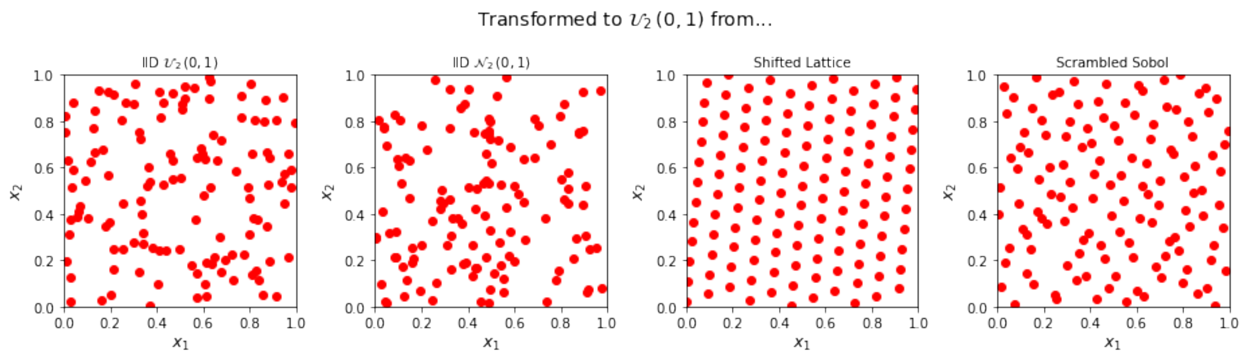
```
def plot_tm_tranformed(tm_name,true_measure,color,lim):
    discrete_distribs = [IIDStdUniform(rng_seed=7), IIDStdGaussian(rng_seed=7),
                         Lattice(rng_seed=7), Sobol(rng_seed=7)]
    dd_names = ["IID $\\mathcal{U}_2\\,(0,1)$", "IID $\\mathcal{N}_2\\,(0,1)$",
                "Shifted Lattice", "Scrambled Sobol"]
    fig, ax = plt.subplots(nrows=1, ncols=len(discrete_distribs), figsize=(13, 4))
    for k, (discrete_distrib, dd_name) in \
```

(continues on next page)

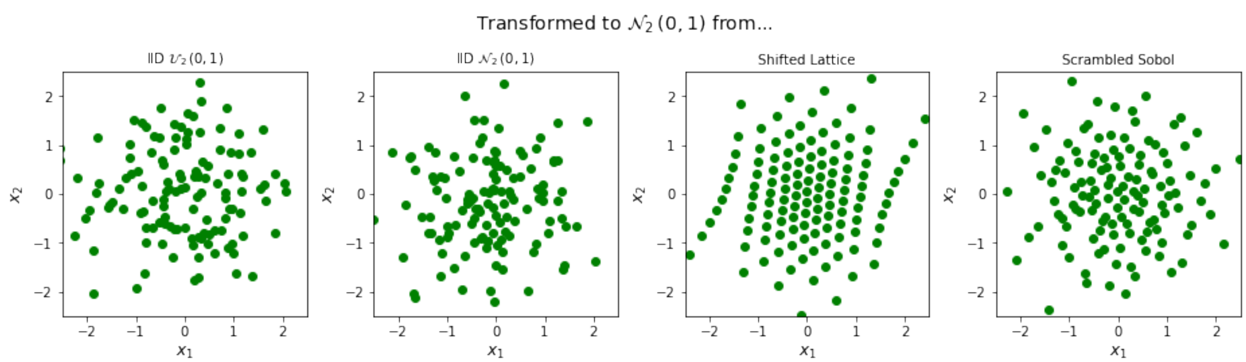
(continued from previous page)

```
            enumerate(zip(discrete_distribs, dd_names)):
        tm_obj = deepcopy(true_measure)
        dd_obj = deepcopy(discrete_distrib)
        tm_obj.set_tm_gen(dd_obj)
        tm_samples = tm_obj[0].gen_tm_samples(1, n).squeeze()
        ax[k].scatter(tm_samples[:, 0], tm_samples[:, 1], color=color)
        ax[k].set_xlabel("$x_1$")
        ax[k].set_ylabel("$x_2$")
        ax[k].set_xlim(lim)
        ax[k].set_ylim(lim)
        ax[k].set_aspect("equal")
        ax[k].set_title(dd_name)
    fig.suptitle("Transformed to %s from..." % tm_name)
    plt.tight_layout()
    prefix = type(true_measure).__name__
    fig.savefig("../outputs/sample_scatters/%s_tm_transform.png" % prefix, dpi=200)
```

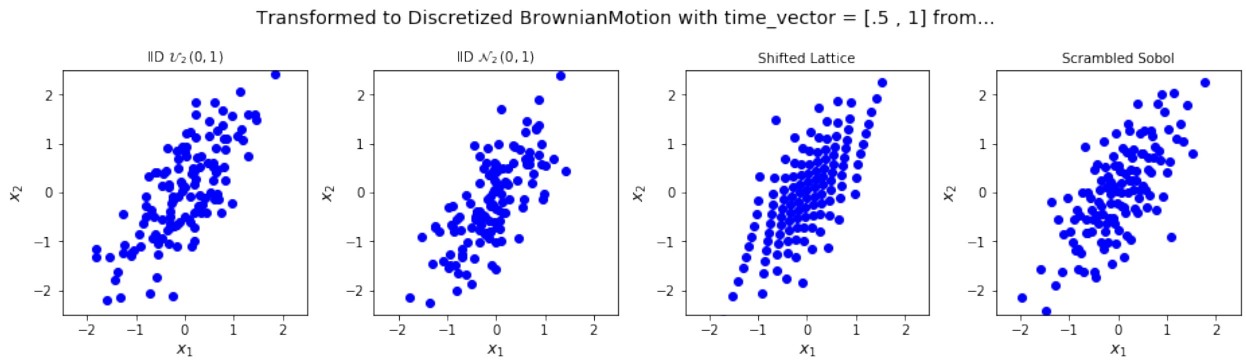
```
plot_tm_tranformed("$\\mathcal{U}_2\\,(0,1)$",Uniform(2),"r",[0, 1])
```




```
plot_tm_tranformed("$\\mathcal{N}_2\\,(0,1)$",Gaussian(2),"g",[-2.5, 2.5])
```



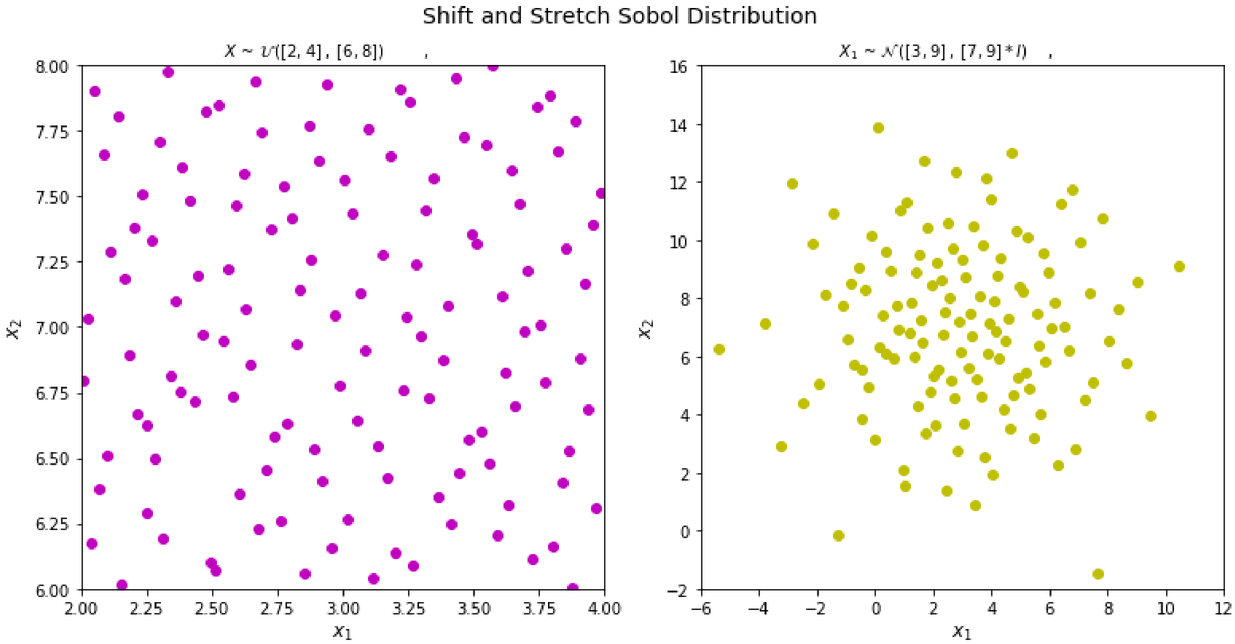

```
tm_obj = BrownianMotion(dimension=2, time_vector= [arange(1 / 2, 3 / 2, 1 / 2)])
plot_tm_tranformed("Discretized BrownianMotion with time_vector = [.5 , 1]",tm_obj,"b
↪",[-2.5, 2.5])
```



4.4.4 Shift and Stretch the True Distribution

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

```
u1_a, u1_b = 2, 4
u2_a, u2_b = 6, 8
g1_mu, g1_var = 3, 9
g2_mu, g2_var = 7, 9
discrete_distrib = Sobol(rng_seed=7)
u_obj = Uniform(dimension=array([2]),
                lower_bound=[array([u1_a, u2_a])],
                upper_bound=[array([u1_b, u2_b])])
n_obj = Gaussian(dimension=array([2]),
                 mean=[array([g1_mu, g2_mu])],
                 variance=[array([g1_var, g2_var])])
colors = ["m", "y"]
fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(11, 6))
for i, (true_measure, color) in enumerate(zip([u_obj, n_obj], colors)):
    tm_obj = deepcopy(true_measure)
    dd_obj = deepcopy(discrete_distrib)
    tm_obj.set_tm_gen(dd_obj)
    tm_samples = tm_obj[0].gen_tm_samples(1, n).squeeze()
    ax[i].scatter(tm_samples[:, 0], tm_samples[:, 1], color=color)
    ax[i].set_xlabel("$x_1$")
    ax[i].set_ylabel("$x_2$")
    ax[i].set_aspect("equal")
ax[0].set_title("$X$ ~ $\\mathcal{U}\\,([%d,%d] \:,\: [%d,%d])$\t,\t" % (u1_a, u1_b,
↪u2_a, u2_b))
ax[1].set_title("$X_1$ ~ $\\mathcal{N}\\,([%d,%d] \:,\: [%d,%d]*I)$\t,\t" % (g1_mu,␣
↪g1_var,g2_mu, g2_var))
ax[0].set_xlim([u1_a, u1_b])
ax[0].set_ylim([u2_a, u2_b])
spread_g1 = ceil(3 * g1_var**.5)
spread_g2 = ceil(3 * g2_var**.5)
ax[1].set_xlim([g1_mu - spread_g1, g1_mu + spread_g1])
ax[1].set_ylim([g2_mu - spread_g2, g2_mu + spread_g2])
fig.suptitle("Shift and Stretch Sobol Distribution")
plt.tight_layout()
fig.savefig("../outputs/sample_scatters/shift_stretch_tm.png", dpi=200)
```



4.4.5 Plots samples on a 2D Keister function

```
# Generate constants for 3d plot in following cell
abs_tol = .5
dim = 2
integrand = Keister(dim)
discrete_distrib = IIDStdGaussian(rng_seed=7)
true_measure = Gaussian(dimension=dim,variance=1/2)
stopping_criterion = CLT(discrete_distrib,true_measure, abs_tol=abs_tol, n_init=16, n_
↪max=1e10)
sol, data = integrate(integrand, true_measure, discrete_distrib, stopping_criterion)
print(data)
```

```
Solution: 2.0554
Keister (Integrand Object)
IIDStdGaussian (Discrete Distribution Object)
    mimics          StdGaussian
Gaussian (True Measure Object)
    dimension       2
    mu              0
    sigma           0.707
CLT (Stopping Criterion Object)
    abs_tol         0.500
    rel_tol         0
    n_max           10000000000
    inflate         1.200
    alpha           0.010
MeanVarData (AccumData Object)
    n               65
    n_total         81
    confid_int      [ 1.646  2.464]
    time_total      0.001
```

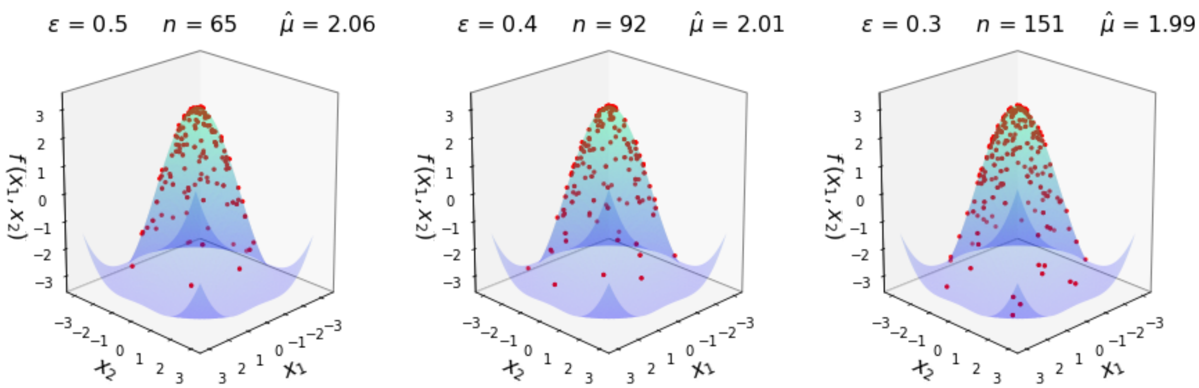
```
# Constants based on running the above CLT Example
eps_list = [.5, .4, .3]
n_list = [65, 92, 151]
mu_hat_list = [2.0554, 2.0143, 1.9926]

# qmcpy objects
dim = 2
integrand = Keister(dim)
true_measure = Gaussian(dim)
discrete_distrib = IIDStdGaussian(rng_seed=7)
true_measure.transform(integrand, discrete_distrib)

# Function Points
nx, ny = (100, 100)
points_fun = zeros((nx * ny, 3))
x = linspace(-3, 3, nx)
y = linspace(-3, 3, ny)
x_2d, y_2d = meshgrid(x, y)
points_fun[:, 0] = x_2d.flatten()
points_fun[:, 1] = y_2d.flatten()
points_fun[:, 2] = integrand[0].f(points_fun[:, :2])
x_surf = points_fun[:, 0].reshape((nx, ny))
y_surf = points_fun[:, 1].reshape((nx, ny))
z_surf = points_fun[:, 2].reshape((nx, ny))

# 3D Plot
fig = plt.figure(figsize=(15, 5))
ax1 = fig.add_subplot(131, projection="3d")
ax2 = fig.add_subplot(132, projection="3d")
ax3 = fig.add_subplot(133, projection="3d")

for idx, ax in enumerate([ax1, ax2, ax3]):
    # Surface
    ax.plot_surface(x_surf, y_surf, z_surf, cmap="winter", alpha=.2)
    # Scatters
    points = zeros((n, 3))
    points[:, :2] = true_measure[0].gen_tm_samples(1, n).squeeze()
    points[:, 2] = integrand[0].f(points[:, :2])
    ax.scatter(points[:, 0], points[:, 1], points[:, 2], color="r", s=5)
    n = n_list[idx]
    epsilon = eps_list[idx]
    mu = mu_hat_list[idx]
    ax.scatter(points[:, 0], points[:, 1], points[:, 2], color="r", s=5)
    ax.set_title("\t$\\epsilon$ = %-7.1f $n$ = %-7d $\\hat{\\mu}$ = %-7.2f "
                 % (epsilon, n, mu), fontdict={"fontsize": 16})
    # axis metas
    n *= 2
    ax.grid(False)
    ax.xaxis.pane.set_edgecolor("black")
    ax.yaxis.pane.set_edgecolor("black")
    ax.set_xlabel("$x_1$", fontdict={"fontsize": 16})
    ax.set_ylabel("$x_2$", fontdict={"fontsize": 16})
    ax.set_zlabel("$f\\:(x_1,x_2)$", fontdict={"fontsize": 16})
    ax.view_init(20, 45)
plt.savefig("../outputs/sample_scatters/Three_3d_SurfaceScatters.png", dpi=250, bbox_
↪inches="tight", pad_inches=.15)
```



4.5 A Monte Carlo vs Quasi-Monte Carlo Comparison

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

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

from matplotlib import pyplot as plt
import matplotlib
%matplotlib inline
```

```
distrib_names = ['IIDStdUniform','IIDStdGaussian','Lattice','Sobol']
```

4.6 Absolute Tolerance Plots

Testing Parameters - relative tolerance = 0 - lds initial sample size = 32 - iid initial sample size = 256 - Results averaged over 3 trials

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

Gaussian True Measure - $\mathcal{N}_3(0, \frac{1}{2})$

Data for the following plot can be generated by running ~~~ python workouts/wo_mc_vs_qmc/comp_abstols.py ~~~

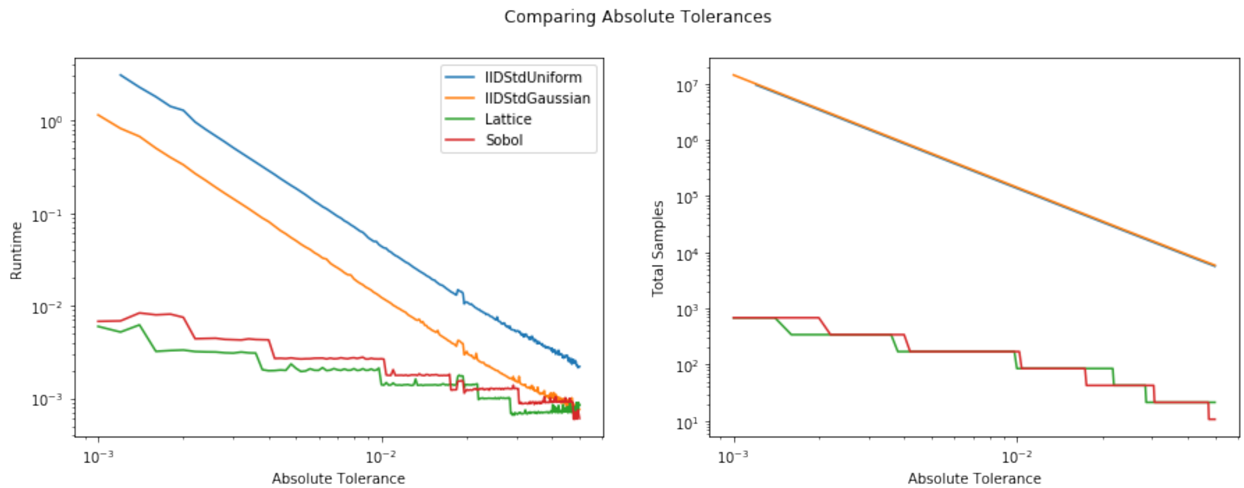
```
df_abstols = pd.read_csv('../outputs/mc_vs_qmc/abs_tol.csv')
df_abstols.loc[::25].style.hide_index()
```

```
fig,ax = plt.subplots(nrows=1, ncols=2, figsize=(15, 5))
abstols = df_abstols['abs_tol'].values
for distrib_name in distrib_names:
    times = df_abstols[distrib_name+'_time'].values
    n_total = df_abstols[distrib_name+'_n'].values
    ax[0].loglog(abstols, times, label=distrib_name)
    ax[1].loglog(abstols, n_total, label=distrib_name)
ax[0].legend(loc='upper right')
ax[0].set_xlabel('Absolute Tolerance')
ax[0].set_ylabel('Runtime')
```

(continues on next page)

(continued from previous page)

```
ax[1].set_xlabel('Absolute Tolerance')
ax[1].set_ylabel('Total Samples')
fig.suptitle('Comparing Absolute Tolerances')
plt.savefig('../outputs/mc_vs_qmc/abstols_plot.png',dpi=200)
```



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

4.7 Dimension Plots

Testing Parameters - absolute tolerance = 0 - relative tolerance = .01 - lds initial sample size = 32 - iid initial sample size = 256 - Results averaged over 3 trials

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

Gaussian True Measure - $\mathcal{N}_d(0, \frac{1}{2})$

Data for the following plot can be generated by running ~~~ python workouts/wo_mc_vs_qmc/comp_dimensions.py ~~~

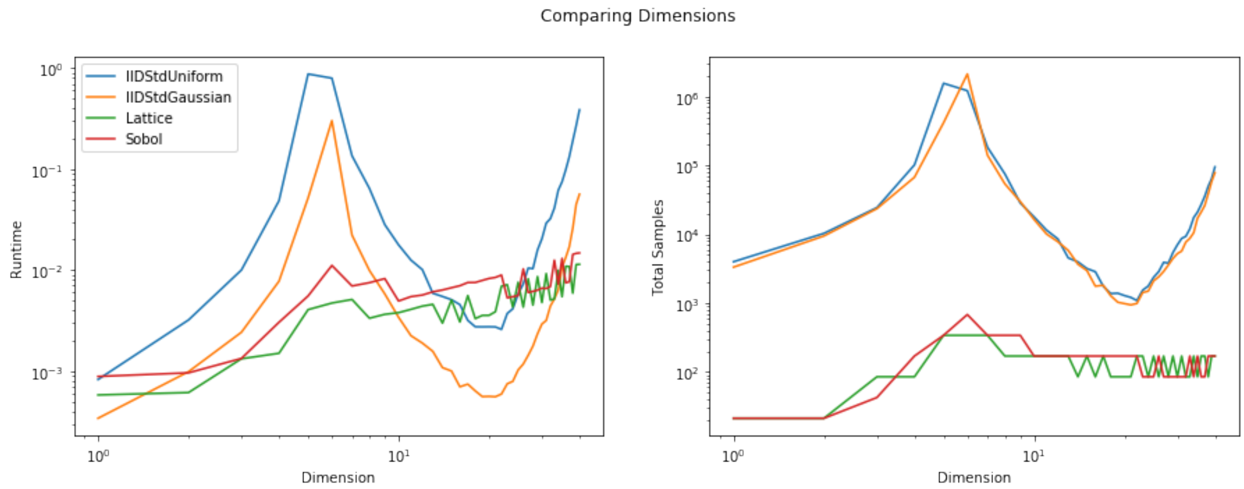
```
df_dimensions = pd.read_csv('../outputs/mc_vs_qmc/dimension.csv')
df_dimensions.dimension = df_dimensions.dimension.astype(int)
df_dimensions.loc[::4].style.hide_index()
```

```
fig,ax = plt.subplots(nrows=1, ncols=2, figsize=(15, 5))
dimensions = df_dimensions['dimension']
for distrib_name in distrib_names:
    times = df_dimensions[distrib_name+'_time'].values
    n_total = df_dimensions[distrib_name+'_n'].values
    ax[0].loglog(dimensions, times, label=distrib_name)
    ax[1].loglog(dimensions, n_total, label=distrib_name)
ax[0].legend(loc='upper left')
ax[0].set_xlabel('Dimension')
ax[0].set_ylabel('Runtime')
ax[1].set_xlabel('Dimension')
ax[1].set_ylabel('Total Samples')
```

(continues on next page)

(continued from previous page)

```
fig.suptitle('Comparing Dimensions')
plt.savefig('../outputs/mc_vs_qmc/dimension_plot.png',dpi=200)
```



4.8 Quasi-Random Sequence Generator Comparison

QMCPy's low-discrepancy-sequence generators are built upon generators developed by 1. D. Nuyens, *The Magic Point Shop of QMC point generators and generating vectors.* MATLAB and Python software, 2018. Available from https://people.cs.kuleuven.be/~dirk.nuyens/

```
from qmcpy import *

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

from numpy import *

from matplotlib import pyplot as plt
import matplotlib
%matplotlib inline
```

4.8.1 General Lattice & Sobol Generator Usage

The following example uses the `Lattice` object to generate samples. The same code works when replacing `Lattice` with `Sobol`

```
# Unshifted Samples
lattice_gen = Lattice(rng_seed=7)
unshifted_samples = lattice_gen.gen_dd_samples(replications=1, n_samples=4,␣
↪dimensions=2, scramble=False)
print('Shape:',unshifted_samples.shape)
print('Samples:\n'+str(unshifted_samples))
```

```
Shape: (1, 4, 2)
Samples:
```

(continues on next page)

(continued from previous page)

```
[[[ 0.000  0.000]
  [ 0.500  0.500]
  [ 0.250  0.750]
  [ 0.750  0.250]]]
```

```
# Shifted Samples
lattice_gen = Lattice(rng_seed=7)
shifted_samples = lattice_gen.gen_dd_samples(replications=2, n_samples=2,␣
↪dimensions=3) # defaults scramble=True
print('Shape:',shifted_samples.shape)
print('Samples:\n'+str(shifted_samples))
```

```
Shape: (2, 2, 3)
Samples:
[[[ 0.625  0.897  0.776]
  [ 0.125  0.397  0.276]]

 [[ 0.225  0.300  0.874]
  [ 0.725  0.800  0.374]]]
```

```
# Next Shifted Samples from same Lattice instance
next_shifted_samples = lattice_gen.gen_dd_samples(replications=2, n_samples=2,␣
↪dimensions = 3)
print('Shape:',next_shifted_samples.shape)
print('Samples:\n'+str(next_shifted_samples))
```

```
Shape: (2, 2, 3)
Samples:
[[[ 0.875  0.647  0.526]
  [ 0.375  0.147  0.026]]

 [[ 0.475  0.050  0.624]
  [ 0.975  0.550  0.124]]]
```

```
next_next_shifted_samples = lattice_gen.gen_dd_samples(replications=2, n_samples=4,␣
↪dimensions = 3)
print('Shape:',next_next_shifted_samples.shape)
print('Samples:\n'+str(next_next_shifted_samples))
```

```
Shape: (2, 4, 3)
Samples:
[[[ 0.750  0.272  0.151]
  [ 0.000  0.022  0.901]
  [ 0.250  0.772  0.651]
  [ 0.500  0.522  0.401]]

 [[ 0.350  0.675  0.249]
  [ 0.600  0.425  0.999]
  [ 0.850  0.175  0.749]
  [ 0.100  0.925  0.499]]]
```

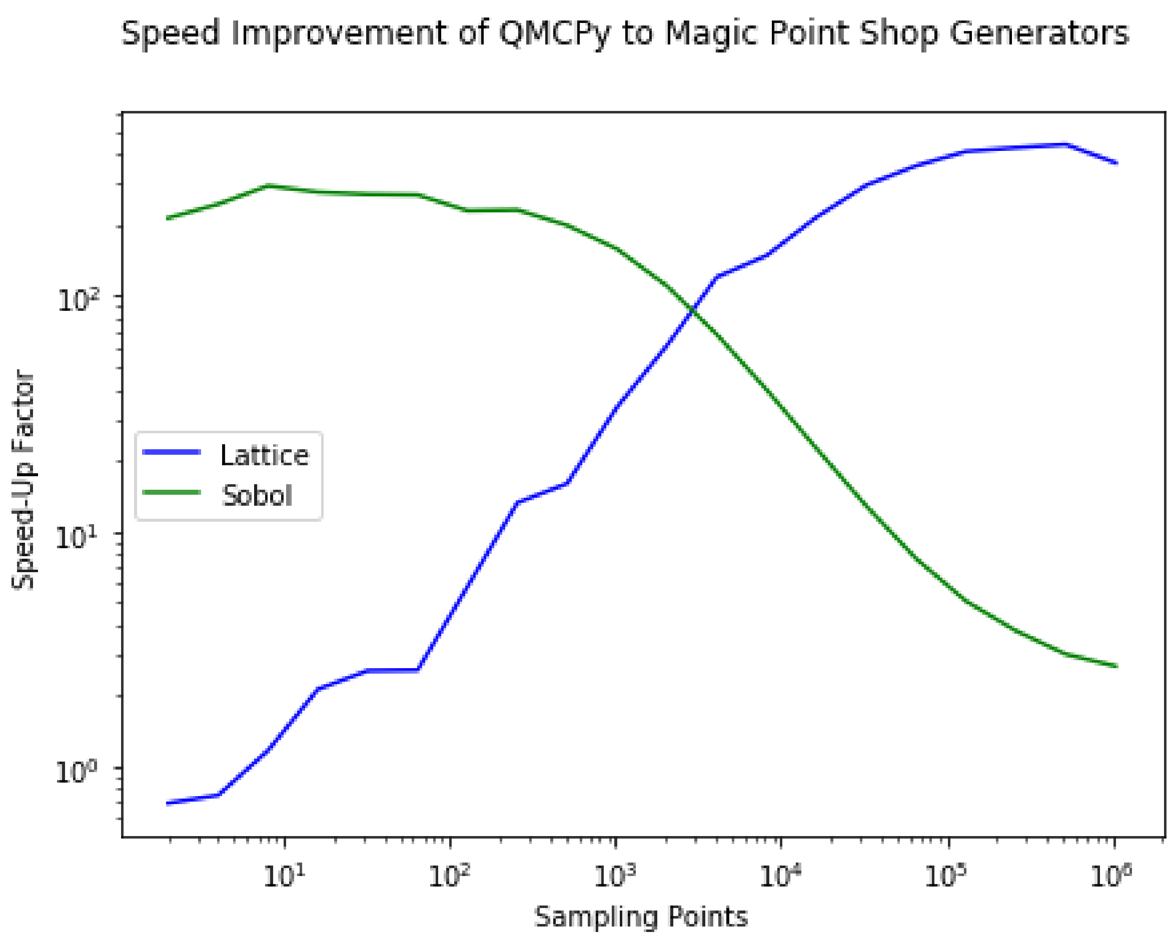
Once replications and dimensions are set in the first call to gen_dd_samples, they are enforced in following calls. The first call to gen_dd_samples can take any n_samples = 2^i. However, following calls require n_samples to be 2^i then 2^{i+1} then 2^{i+2} then … Rerunning the previous 3 blocks with different parameters may help clarify.

4.8.2 *Magic Point Shop* Generators vs QMCPy Generators

In an effort to improve the generators speed, QMCPy developers modified the algorithms developed in *The Magic Point Shop*. The following blocks visualize the speed improvement of QMCPy when generating 1 dimensional unshifted/unscrambled sequences. Data for the following plots can be generated by running ~~~ python workouts/wo_lds_sequences/mps_original_vs_qmcpy.py ~~~

```
df_mps = pd.read_csv('../outputs/lds_sequences/magic_point_shop_times.csv')
df_mps.style.hide_index()
```

```
fig,ax = plt.subplots(nrows=1, ncols=1, figsize=(7, 5))
n = df_mps.n
suf_lattice = df_mps.mps_lattice_time.values / df_mps.qmcpy_lattice_time.values
suf_Sobol = df_mps.mps_Sobol_time.values / df_mps.qmcpy_Sobol_time.values
ax.loglog(n, suf_lattice, label='Lattice', color='b')
ax.loglog(n, suf_Sobol, label='Sobol', color='g')
ax.legend(loc='center left')
ax.set_xlabel('Sampling Points')
ax.set_ylabel('Speed-Up Factor')
fig.suptitle('Speed Improvement of QMCPy to Magic Point Shop Generators')
plt.savefig('../outputs/lds_sequences/mps_vs_qmcpy_generators.png', dpi=200)
```



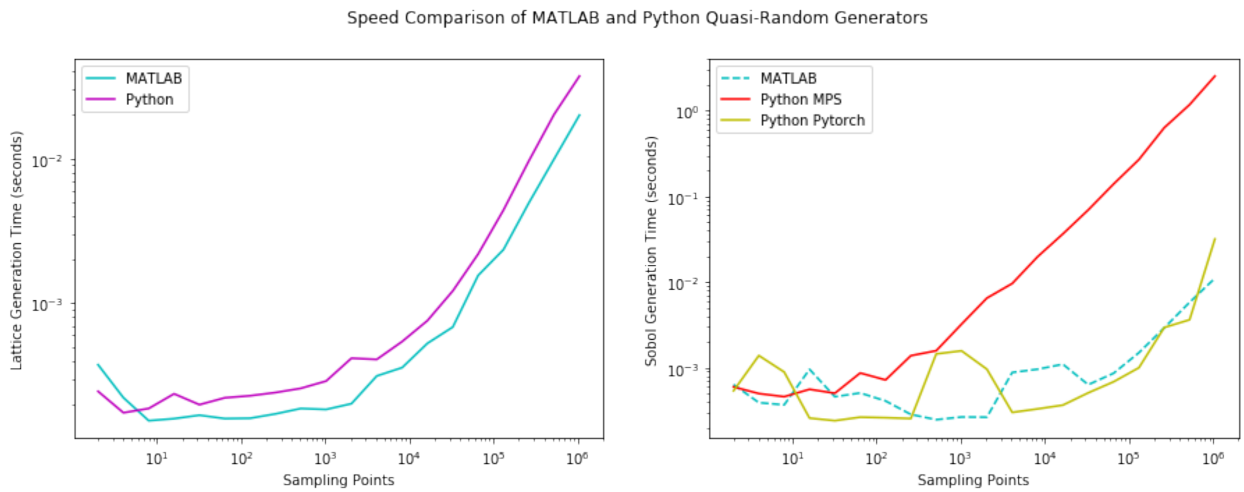
4.8.3 MATLAB vs Python Generator Speed

Compare the speed of low-discrepancy-sequence generators from MATLAB and Python. The following blocks visualize the speed improvement of MATLAB when generating 1 dimensional shifted/scrambled sequences. In the future, we hope to see similar generating times between the two languages. Python data for the following plots can be generated by running ~~~ python workouts/wo_lds_sequences/qmcpy_sequences.py ~~~ MATLAB data can be generated by running the file at `workouts/wo_lds_sequences/matlab_sequences.py`

Notes - For Python both generators are part of the QMCPy package, located at `qmcpy/discrete_distribution/lds_generators.py` - For MATLAB, the Sobol generator is built in, while the lattice generator is part of the GAIL package: - Sou-Cheng T. Choi, Yuhan Ding, Fred J. Hickernell, Lan Jiang, Lluis Antoni Jimenez Rugama, Da Li, Jagadeeswaran Rathinavel, Xin Tong, Kan Zhang, Yizhi Zhang, and Xuan Zhou, GAIL: Guaranteed Automatic Integration Library (Version 2.3) [MATLAB Software], 2019. Available from http://gailgithub.github.io/GAIL_Dev/ - lattice_gen from: https://github.com/GailGithub/GAIL_Dev/blob/master/Algorithms/%2Bgail/lattice_gen.m

```
df_matlab = pd.read_csv('../outputs/lds_sequences/matlab_sequence_times.csv',␣
↪header=None)
df_matlab.columns = ['n', 'matlab_Lattice_time', 'matlab_Sobol_time']
df_python = pd.read_csv('../outputs/lds_sequences/python_sequence_times.csv')
df_python.columns = ['n', 'python_Lattice_time', 'python_Sobol_MPS_time', 'python_
↪Sobol_Pytorch_time']
df_languages = pd.concat([df_matlab['n'],
    df_matlab['matlab_Lattice_time'], df_python['python_Lattice_time'],\
    df_matlab['matlab_Sobol_time'], df_python['python_Sobol_MPS_time'], df_python[
↪'python_Sobol_Pytorch_time']],
    axis = 1)
df_languages.style.hide_index()
```

```
fig,ax = plt.subplots(nrows=1, ncols=2, figsize=(15, 5))
n = df_languages.n
# Lattice Plot
ax[0].loglog(n, df_languages['matlab_Lattice_time'], label='MATLAB', color='c')
ax[0].loglog(n, df_languages['python_Lattice_time'], label='Python', color='m')
ax[0].legend(loc='upper left')
ax[0].set_xlabel('Sampling Points')
ax[0].set_ylabel('Lattice Generation Time (seconds)')
# Sobol Plot
ax[1].loglog(n, df_languages['matlab_Sobol_time'], '--',label='MATLAB', color='c')
ax[1].loglog(n, df_languages['python_Sobol_MPS_time'], label='Python MPS', color='r')
ax[1].loglog(n, df_languages['python_Sobol_Pytorch_time'], label='Python Pytorch',␣
↪color='y')
ax[1].legend(loc='upper left')
ax[1].set_xlabel('Sampling Points')
ax[1].set_ylabel('Sobol Generation Time (seconds)')
# Metas and Export
fig.suptitle('Speed Comparison of MATLAB and Python Quasi-Random Generators')
plt.savefig('../outputs/lds_sequences/matlab_vs_python_generators.png', dpi=200)
```


Speed Comparison of MATLAB and Python Quasi-Random Generators
MATLAB
Python
Lattice Generation Time (seconds)
Sampling Points
MATLAB
Python MPS
Python Pytorch
Sobol Generation Time (seconds)
Sampling Points

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

q

INDEX

K

L

M

N

P

Q

R

S

T

U