# ADAPTIVE QUASI-MONTE CARLO CUBATURE

BY

LLUÍS ANTONI JIMÉNEZ RUGAMA

Submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Applied Mathematics
in the Graduate College of the
Illinois Institute of Technology

Approved _____
Advisor

Chicago, Illinois
December 2016

## ACKNOWLEDGMENT

Obtaining a PhD degree marks a before and after. Little did I know about this journey when Professor Fred J. Hickernell invited me to Illinois Institute of Technology four years ago. A journey that he has made possible. I am thankful for his unparalleled commitment and dedication to my graduate training over these years. We enjoyed many discussions. I will always remember how our 60-minute meetings usually turned into 3-hour long conversations. In addition to the excellent academic training he has provided, I am even more grateful for everything else that he taught me: to be respectful, humble, and patient. With him I learned that sometimes it is more important to ask the right question rather than to know the right answer, which is key to see beyond our limits.

Professor Gregory Fasshauer has always been available to discuss my questions, and provided useful suggestions and comments that went beyond research. In addition, whenever I talked mathematics with him, I always felt my research problems became easier than I thought. For all these reasons, I would like to thank him.

I would like to also thank all the former and current GAIL members with whom I have collaborated to create this Matlab toolbox: Professor Sou-Cheng Choi, Yuhan Ding, Professor Fred J. Hickernell, Lan Jiang, Da Li, Xin Tong, Yizhi Zhang, and Xuan Zhou. In particular, Professor Sou-Cheng Choi's skills as a leader brought us together to work in the same direction with a common goal.

My gratitude also goes to my committee members, Professor Gady Agam, Professor Mustafa Bilgic, Professor Gregory Fasshauer, Professor Fred J. Hickernell, Professor Lulu Kang, and Professor Shuwang Li. Above all, I want to thank them because they were willing to dedicate time to attend my comprehensive and defense examinations, to review my thesis, and to provide helpful comments.

There is someone who discreetly passes unnoticed, but to whom I also owe a big thank you: Mrs. Gladys Collins who assisted me, helped me, and answered any question I had at anytime.

I am grateful to the following institutions and people who supported my PhD financially: "La Caixa" Foundation, the applied mathematics department at IIT, Professor Fred Hickernell through the NSF grants DMS-1115392 and DMS-1522687, Professor Art Owen through the NSF grant DMS-1357690, Dr. Adam Lyon and Fermilab, and SAMSI.

Although there are seven hours of time difference between Chicago and Barcelona, my family never became bored of sending me encouraging messages. Thanks to Gloria Rugama Sabe, Pedro Jiménez Pérez, Primitiva Jiménez Rugama, and Víctor Jiménez Rugama for their love. They made who I am and guided me until the very last day.

And last but not least, to my wife who gave me all this unconditional daily support to obtain my PhD, and with whom I have been working together to build our futures since the beginning.

I would like to dedicate a poem from Antonio Machado that naturally and beautifully explains how there is no path to follow, we build it ourselves. And I thank all the people above who helped me build my own.

## PROVERBIOS Y CANTARES - XXIX

| | |
|---|---|
| **Caminante no hay camino** | **Wayfarer, there is no path** |
| Caminante, son tus huellas | Wayfarer, the only way |
| el camino y nada más; | is your footprints and no other; |
| caminante, no hay camino, | wayfarer, there is no way, |

se hace camino al andar.
Al andar se hace el camino,
y al volver la vista atrás
se ve la senda que nunca
se ha de volver a pisar.
Caminante no hay camino,
sino estelas en la mar.

make your way by going farther.
By going farther, make your way,
till looking back at where you've wandered
you look back on that path you may
not set foot on from now onward.
Wayfarer, there is no way,
only trails of wake on water.

Antonio Machado Ruiz (1875–1939)

TABLE OF CONTENTS

LIST OF TABLES

# LIST OF FIGURES

# ABSTRACT

In some definite integral problems the analytical solution is either unknown or hard to compute. As an alternative, one can approximate the solution with numerical methods that estimate the value of the integral. However, for high dimensional integrals many techniques suffer from the *curse of dimensionality*. This can be solved if we use quasi-Monte Carlo methods which do not suffer from this phenomenon. Section 2.2 describes digital sequences and rank-1 lattice node sequences, two of the most common points used in quasi-Monte Carlo.

If one uses quasi-Monte Carlo, there is still another problem to address: how many points are needed to estimate the integral within a particular absolute error tolerance. In this dissertation, we propose two automatic cubatures based on digital sequences and rank-1 lattice node sequences that estimate high dimensional problems. These new algorithms are constructed in Chapter 3 and the user-specified absolute error tolerance is guaranteed to be satisfied for a specific set of integrands. In Chapter 4 we define a new estimator that satisfies a generalized tolerance function and includes a relative error tolerance option.

An important property of quasi-Monte Carlo methods is that they are effective when the function has low effective dimension. In [1], Sobol' defined the global sensitivity indices, which measure what part of the variance is explained by each dimension. We can use these indices to measure the effective dimensionality of a function. In Chapter 5 we extend our digital sequences cubature to estimate first order and total effect Sobol' indices.

# CHAPTER 1

# INTRODUCTION

## 1.1 Motivation

Multidimensional, or even infinite dimensional integration arises in many applications, including

- Finance: for pricing financial derivatives, computing default probabilities, etc. [2].
- Particle physics: for computing particle masses, [3, 4].
- Imaging: for image synthesis and rendering, [5, 6].
- Sensitivity analysis: for estimating Sobol' indices, a measure of the variance explained by a subset of the input variables, [1].
- Computational probability: to estimate multivariate probabilities, [7, 8].

From the probabilistic point of view, a multidimensional integral can represent the *expectation* of an operator (usually a functional) of a *stochastic process*. For continuous time stochastic processes, the infinite dimensional case is crucial.

Our goal is to develop algorithms that approximate multidimensional integrals with *guaranteed accuracy*. Moreover, the computational effort should be determined automatically by the algorithm, based on the error requirements and the difficulty of the problem.

## 1.2 Numerical Integration

Given an appropriate transformation of the integration domain, the problem

of interest can be written as

$$I(f) := \int_{[0,1]^d} f(\boldsymbol{x}) \, \mathrm{d}\boldsymbol{x}, \tag{1.1}$$

where $d \in \mathbb{N} \cup \{\infty\}$. Here, $d = \infty$ formally corresponds to the limit of a sequence of d-dimensional integrals as $d \to \infty$. Note that (1.1) can also be understood as $\mathbb{E}[f(\boldsymbol{X})]$, with $\boldsymbol{X} \sim U[0,1]^d$.

We are mainly interested in highly stratified Monte Carlo methods, also known as quasi-Monte Carlo (qMC) methods. Such methods offer substantial efficiency gains over IID Monte Carlo (IID MC) sampling, and do not suffer from the *curse of dimensionality* like tensor product cubature methods do. QMC methods are equally weighted sample averages of the integrand values,

$$I(f) \approx \frac{1}{n} \sum_{i=0}^{n-1} f(\boldsymbol{x}_i),$$

where $\{\boldsymbol{x}_i\}_{i=1}^{\infty}$ is a sequence that covers $[0,1]^d$ more evenly than IID uniform random numbers. Two important examples are digitally shifted digital sequences and shifted rank-1 lattice node sequences. The structure of these points requires $n = b^m$, with $m \in \mathbb{N}_0$, and $b$ a prime used to construct the quasi-Monte Carlo sequences.

While IID MC has a root mean square error of $\mathcal{O}\left(b^{(-1/2)m}\right)$, qMC converges as $\mathcal{O}\left(b^{(-1+\delta)m}\right)$ or even as $\mathcal{O}\left(b^{(-2+\delta)m}\right)$ under some smoothness conditions. The difficulty is determining how large $m$ should be to make the error smaller than $\varepsilon_a$. This is the reason why we need the qMC error bound introduced in Chapter 3.

### 1.3 Automatic Cubatures

From our results in Chapter 3, [9] and [10], for all $f$ in a particular cone of

functions $\mathcal{C}$ and whose Fourier coefficients decay rather smoothly with increasing $\kappa$,

$$\left| I(f) - \frac{1}{b^m} \sum_{i=0}^{b^m-1} f(\boldsymbol{x}_i) \right| \le \mathfrak{C}(r,m) \sum_{\kappa=\lfloor b^{m-r-1} \rfloor}^{b^{m-r}-1} \left| \tilde{f}_{m,\kappa} \right|, \quad m = r + \ell_*, r, \ell_* \in \mathbb{N}, \tag{1.2}$$

where $\tilde{f}_{m,\kappa}$ are the discrete Fourier coefficients of $f$ computed by a fast transform algorithm using the same $f(\boldsymbol{x}_i)$ as used for the cubature. Here, $\mathfrak{C}(r,m)$ is an inflation factor that depends on the definition of $\mathcal{C}$.

This inequality leads to the construction of *automatic* qMC algorithms, i.e. algorithms that find themselves the number of qMC integration points needed to meet a prescribed tolerance. For the estimation of (1.1), we designed cubatures based on Sobol' points and integration lattices with computational cost $\mathcal{O}(mb^m)$.

These algorithms have two inputs and are of the form $\widehat{I}(f, \varepsilon_a)$, where $f$ provides the value of the integrand at any point in the unit cube, and $\varepsilon_a$ is the user specified absolute error tolerance. Therefore, for $f \in \mathcal{C}$, the qMC cubature will *automatically* estimate the solution within the prescribed tolerance,

$$\left| I(f) - \widehat{I}(f, \varepsilon_a) \right| \le \varepsilon_a. \tag{1.3}$$

The implementation can be found in our GAIL MATLAB toolbox `http://gailgithub.github.io/GAIL_Dev/` with function names `cubSobol_g` and `cubLattice_g`, or as a C++ package in `http://mcuncert.hepforge.org/` (developed for the High Energy Physics community). MATLAB implementations also allow more general input tolerances taking into account the relative error tolerance $\varepsilon_r$, [11].

Based on these qMC cubatures, we designed new automatic algorithms that estimate first order and total effect Sobol' indices given a user specified tolerance, [12]. To achieve that purpose, we employed the integral definition of the indices and

defined a new estimator as a multidimensional function over integral values. Because these algorithms are not part of our GAIL toolbox, we include them in Appendices A and B.

# CHAPTER 2

# QUASI-MONTE CARLO METHODS

## 2.1 Background

The general form of our problem of interest is the estimation of the integral,

$$\int_{\mathbb{R}^d} g(\boldsymbol{y})\mu(\mathrm{d}\boldsymbol{y}), \quad \mu\left(\mathbb{R}^d\right) < \infty.$$

However, in order to use qMC methods we need to apply the appropriate transformation, and consider that our base problem becomes,

$$I(f) := \int_{[0,1]^d} f(\boldsymbol{x})\mathrm{d}\boldsymbol{x}. \tag{2.1}$$

For instance, if the inverse cumulative distribution function $F^{-1}(\boldsymbol{x})$ of the probability measure $\mu(\mathrm{d}\boldsymbol{y})/\mu\left(\mathbb{R}^d\right)$ exists, one transformation can be the substitution $\boldsymbol{y} = F^{-1}(\boldsymbol{x})$,

$$\int_{\mathbb{R}^d} g(\boldsymbol{y})\mu(\mathrm{d}\boldsymbol{y}) = \mu\left(\mathbb{R}^d\right) \int_{[0,1]^d} g(F^{-1}(\boldsymbol{x}))\,\mathrm{d}\boldsymbol{x}.$$

There are several different numerical methods to estimate the quantity in (2.1). For example, one can apply any tensor product rule or design a particular Gaussian quadrature rule. Nonetheless, many of these methods suffer from the *curse of dimensionality*, also defined as the exponential dependence of the convergence rate on the dimension of the problem $d$. In Table 2.1 we show four numerical integration methods with their corresponding convergence rates.

Table 2.1. Examples of convergence rates as a function of the number of evaluation points $n$ for integrals in dimension $d$.

| Method | Convergence rate |
|---|---|
| Trapezoidal rule | $\mathcal{O}(n^{-2/d})$ |
| Simpson's rule | $\mathcal{O}(n^{-4/d})$ |
| IID MC | $\mathcal{O}(n^{-1/2})$ |
| qMC | $\mathcal{O}(n^{-1+\delta})$ |

The last two methods, IID MC and qMC, are commonly used when solving problems in high dimensions because the convergence rates do not depend on the dimension. The theory behind these convergence rates come from the central limit theorem (MC), and the Koksma-Hlawka inequality (qMC).

QMC is an equally weighted cubature rule that can outperform IID MC. Furthermore, under some smoothness assumptions, convergence rates shown in table 2.1 can be improved using scrambling techniques [13], or special types of points [14].

## 2.2 Low-Discrepancy Sequences

In qMC, the integration over the unit cube (2.1) is approximated by averaging our integrand at a shifted set of points $\mathcal{P} \oplus \boldsymbol{\Delta} := \{\boldsymbol{z}_i \oplus \boldsymbol{\Delta}\}_{i=0}^{n-1}$ with good stratification properties. The stratification refers to how uniformly are the points distributed on each projection on all subset of dimensions. To measure how uniformly distributed are these points, we will to introduce the discrepancy of a point set.

**Definition 2.1.** *Let $\mathcal{M}$ be the set of all intervals of the form $\prod_{j=1}^{d}[a_j, b_j) = \{\boldsymbol{x} \in \mathbb{R}^d : a_j \leq x_j < b_j, 0 \leq a_j < b_j \leq 1\}$, $|\mathcal{P}|$ the cardinality of the set $\mathcal{P}$, and $\lambda$ the Lebesgue measure. Then, the discrepancy of a point set $\mathcal{P}$ is,*

$$D(\mathcal{P}) := \sup_{M \in \mathcal{M}} \left| \frac{|M \cap \mathcal{P}|}{|\mathcal{P}|} - \lambda(M) \right|,$$

Analogously, the star-discrepancy $D^*(\mathcal{P})$ is defined as $D(\mathcal{P})$ with the only difference that the supremum is taken over the sets of the form $\prod_{j=1}^{d}[0, b_j)$ instead of $\prod_{j=1}^{d}[a_j, b_j)$. Both $D(\mathcal{P})$ and $D^*(\mathcal{P})$ are equivalent under the well-known inequality,

$$D^*(\mathcal{P}) \leq D(\mathcal{P}) \leq 2^d D^*(\mathcal{P}).$$

The point sets used in qMC are also called *low-discrepancy* sequences since they satisfy $D(\mathcal{P}) = \mathcal{O}(\log(n)^d/n)$.

In this thesis, we will focus on two of the most common qMC points used in practice: digital nets and rank-1 lattice nodes. For a given prime base $b$, we will use the notation $\mathcal{P}_m := \{\boldsymbol{z}_i\}_{i=0}^{b^m-1}$ to indicate the size of our set $\mathcal{P}$, i.e. $b^m$. We will also consider adding a random shift $\boldsymbol{\Delta}$ to $\mathcal{P}$ since it may be convenient to avoid the origin $\mathbf{0}$ in our point set. This shift will preserve the discrepancy properties of $\mathcal{P}$.

**2.2.1 Digital Nets.** The first definition of digital net was introduced by Niederreiter [15] and also called a digital $(t, m, d)$-net,

**Definition 2.2.** *Let $\mathcal{A}$ be the set of all elementary intervals $A \subset [0,1)^d$ where $A = \prod_{j=1}^{s}[\alpha_j b^{-\gamma_j}, (\alpha_j + 1)b^{-\gamma_j})$, with integers $d \geq 1$, $b \geq 2$, $\gamma_j \geq 0$, and $b^{\gamma_j} > \alpha_j \geq 0$. For $m \geq t \geq 0$, the point set $\mathcal{P}_m \in [0,1)^d$ with $b^m$ points is a $(t, m, d) - net$ in base $b$ if every $A$ with volume $b^{t-m}$ contains $b^t$ points of $\mathcal{P}_m$.*

Thus, a $(t, m, d)$-net is defined such that all elementary intervals of volume $b^{t-m}$ will enclose the same proportion of points of $\mathcal{P}_m$, namely $b^{t-m}|\mathcal{P}_m|$ points. The most evenly spread nets are $(0, m, d)$-nets, since each elementary interval of the smallest volume possible, $b^{-m}$, contains exactly one point. The quality of any $(t, m, d)$-net is therefore measured by the parameter $t$, called $t$-value. Hence, the smaller the value of $t$, the better. In Figure 2.1, one may see a $(0, 5, 2)$-net in base 2.

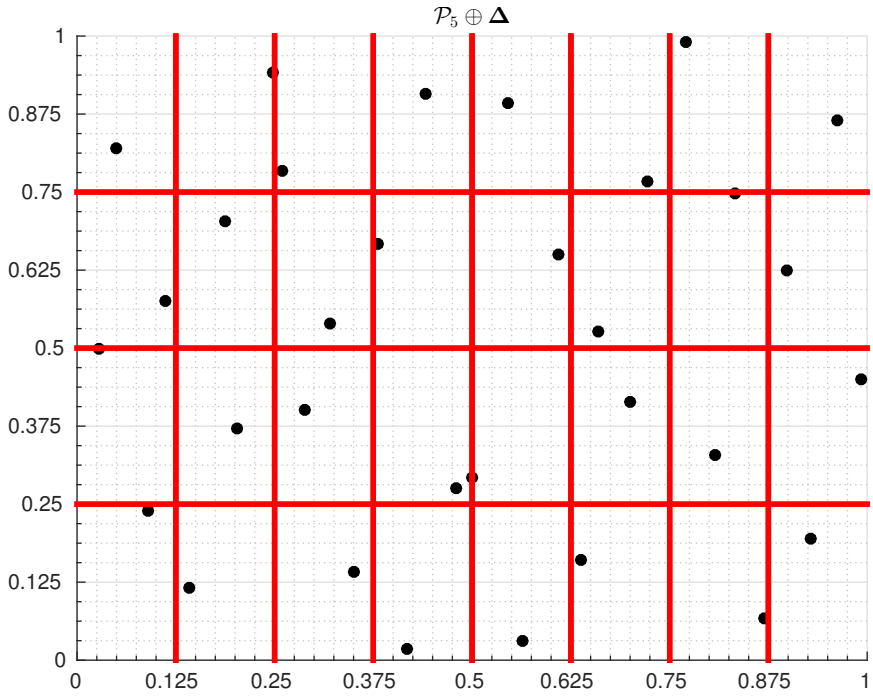


Figure 2.1. First 32 points of a scrambled Sobol' sequence in dimension 2 with digital shift. As an example, red lines indicate that each region of volume 1/32 contains 1 point.

By increasing $m$, we increase the number of points of the $(t, m, d)$-net. When $m \to \infty$, the digital net becomes a sequence,

**Definition 2.3.** *For integers $d \geq 1$, $b \geq 2$, and $t \geq 0$, the sequence $\{\boldsymbol{z}_i\}_{i\in\mathbb{N}_0}$ is a $(t, d)$-sequence in base $b$, if for every set $\mathcal{P}_{\ell,m} := \{\boldsymbol{z}_i\}_{i=\ell b^m}^{(\ell+1)b^m-1}$ with $\ell \geq 0$ and $m \geq t$, $\mathcal{P}_{\ell,m}$ is a $(t, m, d)$-net in base $b$.*

Under the right operations, digital nets have useful properties for the qMC error analysis. Indeed, each digital net $\mathcal{P}_m$ has a group structure under the digit-wise addition:

$$\boldsymbol{x} \oplus \boldsymbol{t} = \left(\sum_{\ell=1}^{\infty}[(x_{j\ell} + t_{j\ell}) \bmod b]b^{-\ell} \bmod 1\right)_{j=1}^{d}.$$

where $x_{j\ell}$ and $t_{j\ell}$ are the $b$-adic decompositions of the $j^{\text{th}}$ component. In addition, digital nets are also related to the orthonormal Fourier Walsh basis functions $\varphi_{\boldsymbol{k}}(\boldsymbol{x}) =$

$\mathrm{e}^{2\pi\sqrt{-1}\langle \boldsymbol{k},\boldsymbol{x}\rangle}$ through the following bilinear operation,

$$\langle \boldsymbol{k}, \boldsymbol{x}\rangle = \left(\sum_{j=1}^{d}\sum_{\ell=0}^{\infty} k_{j\ell}x_{j,\ell+1} \bmod b\right)/b, \quad \boldsymbol{k} \in \mathbb{N}_0^d.$$

**2.2.2 Rank-1 Lattice nodes.** Embedded rank-1 lattice nodes are *low-discrepancy* sequences noted for their simple construction. They have been deeply studied in [16].

**Definition 2.4.** *For a prime base $b$, and a generating vector $\boldsymbol{z}_{b^m} \in [0,1)^d$ with $b^m\boldsymbol{z}_{b^m} \bmod 1 = \mathbf{0}$, the rank-1 lattice node in base $b$ is the set of points,*

$$\mathcal{P}_m := \left\{j\boldsymbol{z}_{b^m} \bmod 1, \quad j = 0, \ldots, b^m - 1\right\}.$$

The choice of generating vector determines whether the points are well uniformly distributed or equivalently, if they have lower discrepancy. In Figure 2.2, we show 2 rank-1 lattices created with different generating vectors. The one on the left is a bad one. The optimal generating vectors are found minimizing the worst-case or average-case errors as summarized in [17] but there is also software to compute it, such as `latbuilder` [18].

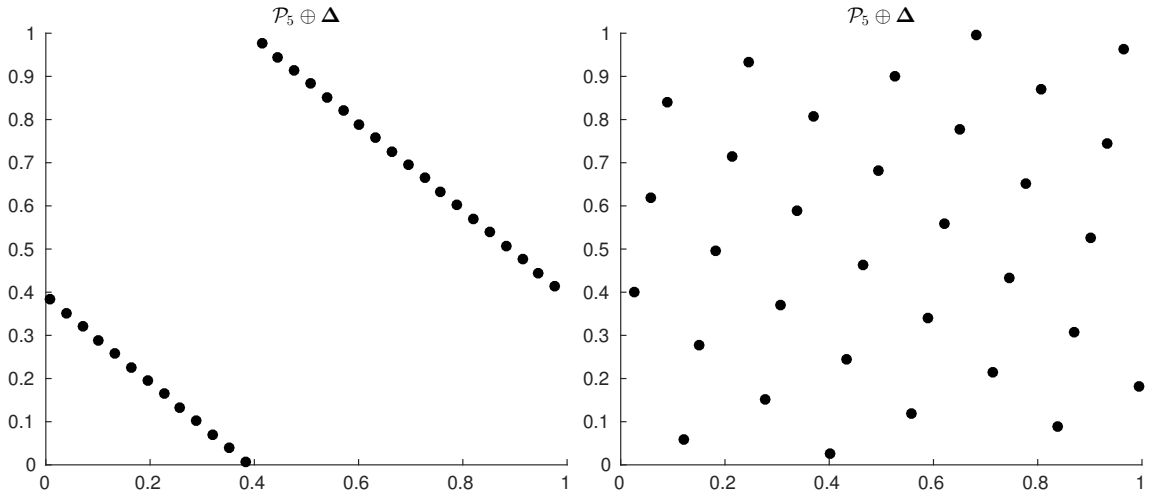


Figure 2.2. First 32 points of two rank-1 lattices in base 2 with random shift. On the left, the generating vector is $\boldsymbol{z} = (1, 31)/32$, and on the right, $\boldsymbol{z} = (1, 7)/32$.

**Definition 2.5.** *Given a sequence of generating vectors* $\boldsymbol{z}_1, \boldsymbol{z}_b, \boldsymbol{z}_{b^2}, \cdots$ *satisfying* $\boldsymbol{z}_{b^{m-1}} = b\boldsymbol{z}_{b^m} \bmod 1$*, the rank-1 lattice node sequence is*

$$\{\boldsymbol{z}_i\}_{i\in\mathbb{N}_0} := \left\{ \quad \sum_{\ell=0}^{\infty} i_\ell \boldsymbol{z}_{b^\ell} \bmod 1, \quad i = i_0 + i_1 b + i_2 b^2 + \cdots + i_{\lfloor \log_b(i) \rfloor} b^{\lfloor \log_b(i) \rfloor} \right\}.$$

As for digital nets, there exists an operation that provides structure to rank-1 lattices. More precisely, each rank-1 lattice node $\mathcal{P}_m$ has a group structure under the modulo one addition:

$$\boldsymbol{x} \oplus \boldsymbol{t} = \boldsymbol{x} + \boldsymbol{t} \bmod 1.$$

Furthermore, rank-1 lattices are related to the orthonormal Fourier basis functions through the following bilinear operation,

$$\langle \boldsymbol{k}, \boldsymbol{x} \rangle = \sum_{j=1}^{d} k_j x_j \bmod 1, \qquad \boldsymbol{k} \in \mathbb{Z}^d.$$

## 2.3 Error Bound

The Koksma-Hlawka inequality [19] is an important result in quasi-Monte

Carlo that bounds the error based on the discrepancy of the point set used. For any function $f$ with bounded variation $V(f)$ in the sense of Hardy and Krause, and any set of points $\mathcal{P} \in [0,1)^d$,

$$\left| I(f) - \frac{1}{|\mathcal{P}|} \sum_{\boldsymbol{z} \in \mathcal{P}} f(\boldsymbol{z} \oplus \boldsymbol{\Delta}) \right| \leq V(f)\, D^*(\mathcal{P} \oplus \boldsymbol{\Delta}).$$

Given that $|\mathcal{P}| = n$, this result shows that for any *low-discrepancy* sequence, the convergence rate is of order $\mathcal{O}(n^{-1+\delta})$, as shown in Table 2.1, since $D^*(\mathcal{P} \oplus \boldsymbol{\Delta})$ decays as $\mathcal{O}(n^{-1+\delta})$.

Nonetheless, the value of $V(f)$ becomes computationally impractical to bound. Thus, in this thesis we propose an alternative error estimation that leads to an adaptive automatic algorithm.

**2.3.1 Fourier Series of the Integrand.** The error of integration using digital nets or rank-1 lattices can be expressed in terms of the Fourier coefficients of the integrand. Let $\mathbb{K}$ be the space of wavenumbers, namely $\mathbb{N}_0^d$ or $\mathbb{Z}^d$ depending on if we refer to digital nets or rank-1 lattices. Thus, for $f \in L^2([0,1]^d)$,

$$f(\boldsymbol{x}) = \sum_{\boldsymbol{k} \in \mathbb{K}} \hat{f}_{\boldsymbol{k}} \varphi_{\boldsymbol{k}}(\boldsymbol{x}), \qquad \varphi_{\boldsymbol{k}}(\boldsymbol{x}) := \mathrm{e}^{2\pi\sqrt{-1}\langle \boldsymbol{k}, \boldsymbol{x} \rangle}. \tag{2.2}$$

By definition, $\langle \boldsymbol{k}, \boldsymbol{x} \oplus \boldsymbol{t} \rangle = \langle \boldsymbol{k}, \boldsymbol{x} \rangle + \langle \boldsymbol{k}, \boldsymbol{t} \rangle \bmod 1$. Hence, one has that $\varphi_{\boldsymbol{k}}(\boldsymbol{x} \oplus \boldsymbol{t}) = \varphi_{\boldsymbol{k}}(\boldsymbol{x}) \varphi_{\boldsymbol{k}}(\boldsymbol{t})$.

The case of the Fourier complex exponential series expansion requires $f$ to be periodic. If our integrand is not periodic, one can use any transformation $\phi$ from [20, Sec. 2.12] such that $f(\phi(\boldsymbol{x}))$ is periodic and,

$$\int_{[0,1]^d} f(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} = \int_{[0,1]^d} f(\phi(\boldsymbol{x}))\, \mathrm{d}\boldsymbol{x}.$$

In [21], Hickernell suggests the use of the Baker's transform which consists on applying $\phi(x) = 1 - |2x - 1|$ to each dimension. To provide a visual understanding of what the transform does, in Figure 2.3 we show how the function $f(x_1, x_2) = \sin\left((100(x_1 - 0.1)^2 + 25(x_2 - 0.1)^2)^{0.6}\right) / \left(100(x_1 - 0.1)^2 + 25(x_2 - 0.1)^2\right)^{0.3}$ is affected after applying the transform.

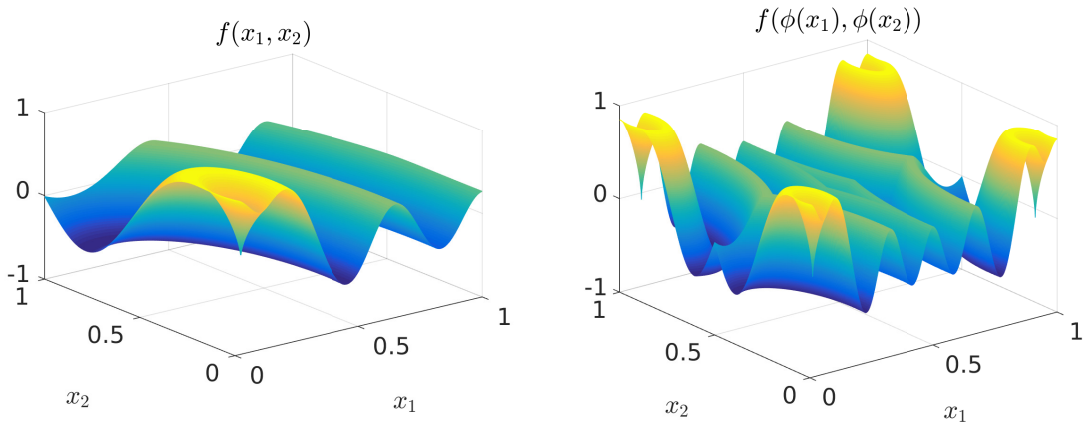


Figure 2.3. On the left, we show $f(x_1, x_2)$. On the right, the function after applying the Baker's transform, $f(\phi(x_1), \phi(x_2))$.

Before introducing the error based on the Fourier expansion (2.2) of the integrand, we need to define the dual set of $\mathcal{P}$, also named the dual-net if $\mathcal{P}$ is a digital net or the dual-lattice if $\mathcal{P}$ is a rank-1 lattice.

**Definition 2.6.** *The dual set of* $\mathcal{P}$, $|\mathcal{P}| < \infty$, *is*

$$\mathcal{P}^{\perp} := \{\boldsymbol{k} \in \mathbb{K} : \langle \boldsymbol{k}, \boldsymbol{z} \rangle = 0, \ \textit{for all } \boldsymbol{z} \in \mathcal{P}\}.$$

Since the finite set $\mathcal{P}$ is a group under $\oplus$, one may derive a useful formula for the average of a Fourier basis function. For all wavenumbers $\boldsymbol{k} \in \mathbb{K}$ and all $\boldsymbol{t} \in \mathcal{P}$ one has,

$$0 = \frac{1}{|\mathcal{P}|} \sum_{\boldsymbol{z} \in \mathcal{P}} [\varphi_{\boldsymbol{k}}(\boldsymbol{z}) - \varphi_{\boldsymbol{k}}(\boldsymbol{z} \oplus \boldsymbol{t})]$$

$$= \frac{1}{|\mathcal{P}|} \sum_{\boldsymbol{z} \in \mathcal{P}} [\varphi_{\boldsymbol{k}}(\boldsymbol{z}) - \varphi_{\boldsymbol{k}}(\boldsymbol{z})\varphi_{\boldsymbol{k}}(\boldsymbol{t})]$$
$$= [1 - \varphi_{\boldsymbol{k}}(\boldsymbol{t})] \frac{1}{|\mathcal{P}|} \sum_{\boldsymbol{z} \in \mathcal{P}} \varphi_{\boldsymbol{k}}(\boldsymbol{z}).$$

We know that if $\boldsymbol{k} \in \mathcal{P}^{\perp}$, one has $\varphi_{\boldsymbol{k}}(\boldsymbol{t}) = 1$ for all $\boldsymbol{t} \in \mathcal{P}$. By this equality, if $\varphi_{\boldsymbol{k}}(\boldsymbol{t}) \neq 1$ for some $\boldsymbol{t}$, then $\sum_{\boldsymbol{z} \in \mathcal{P}} \varphi_{\boldsymbol{k}}(\boldsymbol{z}) = 0$. It follows that the average of the sampled Fourier basis function values is either one or zero, depending on whether the wavenumber is in the dual set or not:

$$\frac{1}{|\mathcal{P}|} \sum_{\boldsymbol{z} \in \mathcal{P}} \varphi_{\boldsymbol{k}}(\boldsymbol{z}) = \mathbb{1}_{\mathcal{P}^{\perp}}(\boldsymbol{k}) = \begin{cases} 1, & \boldsymbol{k} \in \mathcal{P}^{\perp}, \\ 0, & \boldsymbol{k} \notin \mathcal{P}^{\perp}. \end{cases}$$

Hence, the integration error using a shifted point set $\mathcal{P} \oplus \boldsymbol{\Delta}$ can be expressed in terms of the Fourier coefficients,

$$\begin{aligned} \left| I(f) - \frac{1}{|\mathcal{P}|} \sum_{\boldsymbol{z} \in \mathcal{P}} f(\boldsymbol{z} \oplus \boldsymbol{\Delta}) \right| &= \left| \hat{f}_{\boldsymbol{0}} - \frac{1}{|\mathcal{P}|} \sum_{\boldsymbol{z} \in \mathcal{P}} \sum_{\boldsymbol{k} \in \mathbb{K}} \hat{f}_{\boldsymbol{k}} \varphi_{\boldsymbol{k}}(\boldsymbol{z} \oplus \boldsymbol{\Delta}) \right| \\ &= \left| \hat{f}_{\boldsymbol{0}} - \frac{1}{|\mathcal{P}|} \sum_{\boldsymbol{z} \in \mathcal{P}} \sum_{\boldsymbol{k} \in \mathbb{K}} \hat{f}_{\boldsymbol{k}} \varphi_{\boldsymbol{k}}(\boldsymbol{z}) \varphi_{\boldsymbol{k}}(\boldsymbol{\Delta}) \right| \\ &= \left| \hat{f}_{\boldsymbol{0}} - \sum_{\boldsymbol{k} \in \mathbb{K}} \hat{f}_{\boldsymbol{k}} \varphi_{\boldsymbol{k}}(\boldsymbol{\Delta}) \frac{1}{|\mathcal{P}|} \sum_{\boldsymbol{z} \in \mathcal{P}} \varphi_{\boldsymbol{k}}(\boldsymbol{z}) \right| \\ &= \left| \sum_{\boldsymbol{k} \in \mathcal{P}^{\perp} \setminus \{\boldsymbol{0}\}} \hat{f}_{\boldsymbol{k}} \varphi_{\boldsymbol{k}}(\boldsymbol{\Delta}) \right| \\ &\leq \sum_{\boldsymbol{k} \in \mathcal{P}^{\perp} \setminus \{\boldsymbol{0}\}} \left| \hat{f}_{\boldsymbol{k}} \right|. \end{aligned} \tag{2.3}$$

Equation (2.3) is a well-known result. However, we do not assume the knowledge of the Fourier coefficients. Our plan, detailed in Chapter 3, is to use the discrete Fourier coefficients instead. In order to achieve that goal, first we need to assume that the decay rate of higher frequency coefficients can be inferred from the decay rate of lower

frequency coefficients. Finally, we estimate these lower frequency coefficients using the Fast Fourier Transform.

# CHAPTER 3

# ADAPTIVE QUASI-MONTE CARLO CUBATURE

## 3.1 Wavenumber Space Mapping

Although inequality (2.3) is the basis of our algorithm, it remains the question about how to order the summation over the set $\mathcal{P}^{\perp} \setminus \{\mathbf{0}\}$ in high dimensions. One common choice is the hyper-cross norm, [16, Chap. 4]. Nonetheless, our iterative nature of the new automatic cubatures needs an alternative mapping of the wavenumber space $\mathbb{K}$. For the following, we will name our integral estimate

$$\hat{I}_m(f) := \frac{1}{|\mathcal{P}_m|} \sum_{\boldsymbol{z} \in \mathcal{P}_m} f(\boldsymbol{z} \oplus \boldsymbol{\Delta}). \tag{3.1}$$

For any digital or embedded rank-1 lattice sequence constructed as described in Chapter 2 and [9, 10], the tower property

$$\{\mathbf{0}\} = \mathcal{P}_0 \subset \cdots \subset \mathcal{P}_m \subset \cdots \subset \mathcal{P}_\infty = \{\boldsymbol{z}_i\}_{i=0}^{\infty},$$

implies that,

$$\mathbb{K} = \mathcal{P}_0^{\perp} \supset \cdots \supset \mathcal{P}_m^{\perp} \supset \cdots \supset \mathcal{P}_\infty^{\perp} = \{\mathbf{0}\}. \tag{3.2}$$

In addition, if we note $\mathcal{P}_m^{\perp} := \mathcal{P}_m^{\perp,0}$, there exist $b^m - 1$ other cosets such that $\mathbb{K} = \cup_{j=0}^{b^m-1} \mathcal{P}_m^{\perp,j}$ for all $m \in \mathbb{N}_0$, and property (3.2) can be generalized. Fix $j \in \mathbb{N}_0$ and let $j_m = j \bmod b^m$. Then,

$$\mathbb{K} = \mathcal{P}_0^{\perp,j_0} \supset \cdots \supset \mathcal{P}_{m-1}^{\perp,j_{m-1}} \supset \mathcal{P}_m^{\perp,j_m} \supset \cdots \supset \mathcal{P}_\infty^{\perp} = \{\mathbf{0}\}. \tag{3.3}$$

We will define an ordering of the wavenumber space $\tilde{\boldsymbol{k}} : \mathbb{N}_0 \to \mathbb{K}$ that satisfies $\{\tilde{\boldsymbol{k}}(\lambda b^m)\}_{\lambda=0}^{\infty} = \mathcal{P}_m^{\perp}$. This mapping requires to establish some conditions such that

equation (3.3) is not violated. Because digital nets and rank-1 lattices are associated to different wavenumber spaces, and operations $\langle\cdot,\cdot\rangle$ and $\oplus$, we present two different algorithms below.

**Algorithm 1** (Rank-1 lattice sequence mapping)**.** Let $\mathcal{P}_\infty = \{\boldsymbol{z}_i\}_{i=0}^{\infty}$ be a rank-1 lattice sequence, and define $\widetilde{\nu}_m(\boldsymbol{k}) := b^m \langle \boldsymbol{k}, \boldsymbol{z}_{b^{m-1}}\rangle$. We define $\tilde{\boldsymbol{k}} : \mathbb{N}_0 \to \mathbb{Z}^d$ one-to-one and onto recursively as follows:

**Step 1.** Set $\tilde{\boldsymbol{k}}(0) = \boldsymbol{0}$.

**Step 2.** For $m \in \mathbb{N}_0$

For $\kappa \in \{0, \dots, b^m - 1\}$

Let $a \in \{0, \dots, b-1\}$ be such that $\widetilde{\nu}_{m+1}(\tilde{\boldsymbol{k}}(\kappa)) = \widetilde{\nu}_m(\tilde{\boldsymbol{k}}(\kappa)) + ab^m$.

i) If $a \neq 0$, choose $\tilde{\boldsymbol{k}}(\kappa + ab^m) \in \{\boldsymbol{k} \in \mathbb{Z}^d : \widetilde{\nu}_{m+1}(\boldsymbol{k}) = \widetilde{\nu}_m(\tilde{\boldsymbol{k}}(\kappa))\}$.

ii) Choose $\tilde{\boldsymbol{k}}(\kappa + a'b^m) \in \{\boldsymbol{k} \in \mathbb{Z}^d : \widetilde{\nu}_{m+1}(\boldsymbol{k}) = \widetilde{\nu}_m(\tilde{\boldsymbol{k}}(\kappa)) + a'b^m\}$,

for $a' \in \{1, \dots, b-1\} \setminus \{a\}$.

**Algorithm 2** (Digital sequence mapping)**.** Given a digital sequence $\mathcal{P}_\infty = \{\boldsymbol{z}_i\}_{i=0}^{\infty}$, define $\tilde{\boldsymbol{k}} : \mathbb{N}_0 \to \mathbb{N}_0^d$ one-to-one and onto recursively as follows:

**Step 1.** Set $\tilde{\boldsymbol{k}}(0) = \boldsymbol{0}$.

**Step 2.** For $m \in \mathbb{N}_0$

For $\kappa \in \{0, \dots, b^m - 1\}$

Choose the values of $\tilde{\boldsymbol{k}}(\kappa + b^m), \dots, \tilde{\boldsymbol{k}}(\kappa + (b-1)b^m)$ from the sets

$$\left\{\boldsymbol{k} \in \mathbb{N}_0^d : \boldsymbol{k} \ominus \boldsymbol{k}(\kappa) \in \mathcal{P}_m^{\perp}, \left\langle \boldsymbol{k} \ominus \tilde{\boldsymbol{k}}(\kappa), \boldsymbol{z}_{b^m} \right\rangle = a/b\right\}, \quad a = 1, \dots, b-1,$$

but not necessarily in that order.

Algorithms 1 and 2 do not uniquely define the ordering and one has the flexibility to choose part of it. In the end, we want to fix $\tilde{\boldsymbol{k}}(\kappa)$ such that $\hat{f}(\tilde{\boldsymbol{k}}(\kappa)) \to 0$ as

$\kappa \to \infty$. For practical purposes, our automatic cubatures choose the ordering heuristically where the biggest approximated values of $\left|\hat{f}(\tilde{\boldsymbol{k}}(\kappa))\right|$ are mapped to the lowest values of $\kappa$. An important remark is that using Algorithms 1 and 2, $\tilde{\boldsymbol{k}}$ is one-to-one and onto, as proven in [9, 10].

### 3.2 Cone of Functions

We assume that the mapping $\tilde{\boldsymbol{k}}(\kappa)$ is already defined as explained in Section 3.1 and adopt the notation $\hat{f}_\kappa := \hat{f}(\tilde{\boldsymbol{k}}(\kappa))$. Since Algorithms 1 and 2 satisfy $\{\tilde{\boldsymbol{k}}(\lambda b^m)\}_{\lambda=0}^{\infty} = \mathcal{P}_m^{\perp}$, inequality (2.3) becomes,

$$\left| I(f) - \widehat{I}_m(f) \right| \leq \sum_{\lambda=1}^{\infty} \left| \hat{f}_{\lambda b^m} \right|, \tag{3.4}$$

Nevertheless, we do not assume the knowledge of the Fourier coefficients of our integrand $f$. Instead, we approximate them with $\tilde{f}_{m,\kappa}$,

$$\tilde{f}_{m,\kappa} := \widehat{I}_m(f\overline{\varphi_{\tilde{\boldsymbol{k}}(\kappa)}})$$

The values $\tilde{f}_{m,\kappa}$ correspond to the Fast Fourier Transform applied to the function values at points $\mathcal{P}_m \oplus \boldsymbol{\Delta}$, and according to [9, 10], the aliasing relationship between $\tilde{f}_{m,\kappa}$ and $\hat{f}_\kappa$ is,

$$\tilde{f}_{m,\kappa} = \hat{f}_\kappa + \sum_{\lambda=1}^{\infty} \hat{f}_{\kappa+\lambda b^m} \varphi_{\tilde{\boldsymbol{k}}(\kappa+\lambda b^m)-\tilde{\boldsymbol{k}}(\kappa)}(\boldsymbol{\Delta}). \tag{3.5}$$

For the automatic cubatures, the integrands need to have absolutely summable Fourier coefficients and $f(\boldsymbol{x})$ needs to exist for any $\boldsymbol{x} \in [0,1)^d$. Consider the following

sums defined for $\ell, m \in \mathbb{N}_0$, $\ell \leq m$:

$$S_m(f) = \sum_{\kappa=\lfloor b^{m-1} \rfloor}^{b^m-1} \left|\hat{f}_\kappa\right|, \qquad \widehat{S}_{\ell,m}(f) = \sum_{\kappa=\lfloor b^{\ell-1} \rfloor}^{b^\ell-1} \sum_{\lambda=1}^{\infty} \left|\hat{f}_{\kappa+\lambda b^m}\right|,$$

$$\breve{S}_m(f) = \widehat{S}_{0,m}(f) + \cdots + \widehat{S}_{m,m}(f) = \sum_{\kappa=b^m}^{\infty} \left|\hat{f}_\kappa\right|, \qquad \widetilde{S}_{\ell,m}(f) = \sum_{\kappa=\lfloor b^{\ell-1} \rfloor}^{b^\ell-1} \left|\tilde{f}_{m,\kappa}\right|.$$

Note that $\widetilde{S}_{\ell,m}(f)$ is the only one that can be observed from data because it involves the discrete transform coefficients. In fact, our automatic cubature will be based on this sum bounding the other three: $S_m(f), \widehat{S}_{\ell,m}(f)$, and $\breve{S}_m(f)$.

Let $\ell_* \in \mathbb{N}$ be some fixed integer and $\widehat{\omega}$ and $\mathring{\omega}$ be some bounded non-negative valued functions. We define a *cone*, $\mathcal{C}$, of functions with absolutely convergent Fourier coefficients that decay according to certain inequalities:

$$\mathcal{C} := \{f \in AC([0,1]^d) : \widehat{S}_{\ell,m}(f) \leq \widehat{\omega}(m-\ell)\breve{S}_m(f), \quad \ell \leq m, \\ \breve{S}_m(f) \leq \mathring{\omega}(m-\ell)S_\ell(f), \quad \ell_* \leq \ell \leq m\}. \quad (3.6)$$

We also require the existence of $r$ such that $\widehat{\omega}(r)\mathring{\omega}(r) < 1$ and that $\lim_{m\to\infty} \mathring{\omega}(m) = 0$. This set is a cone, i.e. $f \in \mathcal{C} \implies af \in \mathcal{C}\ \forall a \in \mathbb{R}$, but it is not convex. A wider discussion on the advantages and disadvantages of designing numerical algorithms for cones of functions can be found in [22].

Functions in $\mathcal{C}$ have Fourier coefficients that do not oscillate wildly. According to (3.4), the error of our integration is bounded by $\widehat{S}_{0,m}(f)$. Nevertheless, because we do not assume the knowledge of the true Fourier coefficients, we will use $S_\ell(f)$ as an indicator for the error. Intuitively, the cone conditions enforce these two sums to follow a similar trend. Thus, one can expect that small values of $S_\ell(f)$ imply small values of $\widehat{S}_{0,m}(f)$.

The first inequality controls how an infinite sum of some of the larger wavenumber coefficients are bounded above by a sum of all the surrounding coefficients. The second inequality controls how the sum of these surrounding coefficients is bounded above by a finite sum of some smaller wavenumber Fourier coefficients. In Figure 3.1 we can see how $S_8(f)$ can be used to bound $\breve{S}_{12}(f)$, and $\breve{S}_{12}(f)$ to bound $\hat{S}_{0,12}(f)$. Here, $\hat{S}_{0,12}(f)$ also corresponds to the error bound in (3.4).

For small $\ell$ the sum $S_\ell(f)$ includes only a few summands. Therefore, it could accidentally happen that $S_\ell(f)$ is too small compared to $\breve{S}_m(f)$. To avoid this possibility, the cone definition includes the constraint that $\ell$ is greater than some minimum $\ell_*$.

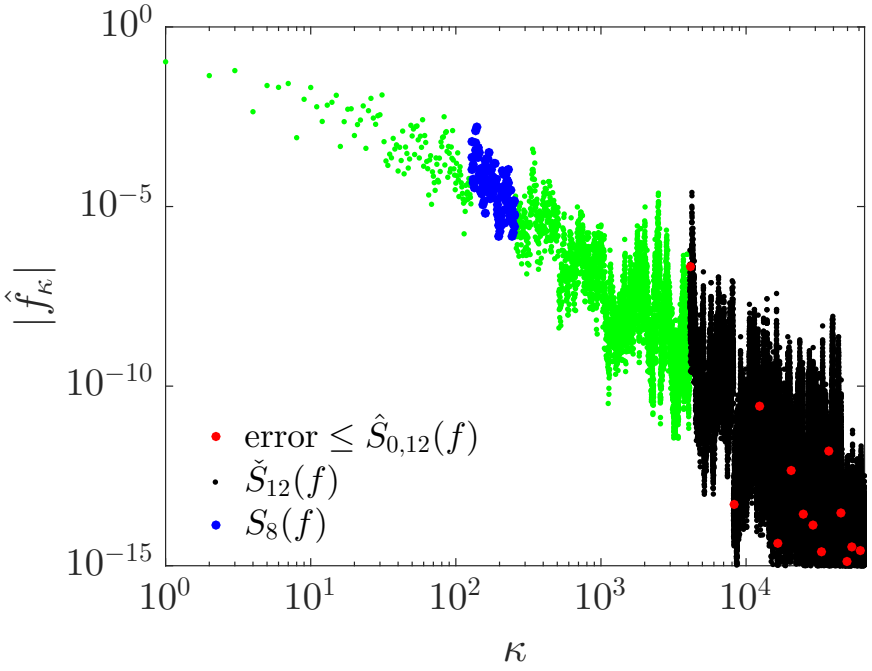


Figure 3.1. The magnitudes of the true Walsh Fourier coefficients for the function $f(x) = \mathrm{e}^{-3x}\sin(10x^2)$.

Because we do not assume the knowledge of the true Fourier coefficients, for functions in $\mathcal{C}$ we need bounds on $S_\ell(f)$ in terms of the sum of the discrete coefficients

$\widetilde{S}_{\ell,m}(f)$. Applying (3.5), and the definition of the cone in (3.6):

$$\begin{aligned} S_\ell(f) &= \sum_{\kappa=\lfloor b^{\ell-1}\rfloor}^{b^\ell-1} \left|\hat{f}_\kappa\right| = \sum_{\kappa=\lfloor b^{\ell-1}\rfloor}^{b^\ell-1} \left|\tilde{f}_{m,\kappa} - \sum_{\lambda=1}^{\infty} \hat{f}_{\kappa+\lambda b^m} \varphi_{\tilde{\boldsymbol{k}}(\kappa+\lambda b^m)-\tilde{\boldsymbol{k}}(\kappa)}(\boldsymbol{\Delta})\right| \\ &\le \sum_{\kappa=\lfloor b^{\ell-1}\rfloor}^{b^\ell-1} \left|\tilde{f}_{m,\kappa}\right| + \sum_{\kappa=\lfloor b^{\ell-1}\rfloor}^{b^\ell-1} \sum_{\lambda=1}^{\infty} \left|\hat{f}_{\kappa+\lambda b^m}\right| = \widetilde{S}_{\ell,m}(f) + \widehat{S}_{\ell,m}(f) \\ &\le \widetilde{S}_{\ell,m}(f) + \widehat{\omega}(m-\ell)\mathring{\omega}(m-\ell) S_\ell(f) \end{aligned} \tag{3.7}$$

and provided that $\widehat{\omega}(m-\ell)\mathring{\omega}(m-\ell) < 1$,

$$S_\ell(f) \le \frac{\widetilde{S}_{\ell,m}(f)}{1 - \widehat{\omega}(m-\ell)\mathring{\omega}(m-\ell)}. \tag{3.8}$$

Finally, by (3.4) and the cone conditions, (3.8) implies a data-based error bound:

$$\begin{aligned} \left|I(f) - \hat{I}_m(f)\right| &\le \sum_{\lambda=1}^{\infty} \left|\hat{f}_{\lambda b^m}\right| = \widehat{S}_{0,m}(f) \le \widehat{\omega}(m) \breve{S}_m(f) \\ &\le \widehat{\omega}(m)\mathring{\omega}(m-\ell) S_\ell(f) \\ &\le \frac{\widehat{\omega}(m)\mathring{\omega}(m-\ell)}{1 - \widehat{\omega}(m-\ell)\mathring{\omega}(m-\ell)} \widetilde{S}_{\ell,m}(f). \end{aligned} \tag{3.9}$$

### 3.3 Automatic Cubature

Inequality (3.9) suggests the following algorithm. First, choose $\ell_*$ and fix $r := m - \ell \in \mathbb{N}$ such that $\widehat{\omega}(r)\mathring{\omega}(r) < 1$ for $\ell \ge \ell_*$. Then, define

$$\mathfrak{C}(m) := \frac{\widehat{\omega}(m)\mathring{\omega}(r)}{1 - \widehat{\omega}(r)\mathring{\omega}(r)}. \tag{3.10}$$

The choice of the parameter $r$ is important. Larger $r$ means a smaller $\mathfrak{C}(m)$, but it also makes the error bound more dependent on smaller indexed Fourier coefficients.

**Algorithm 3** (Adaptive Cubatures $\widehat{I}$: `cubLattice_g`, `cubSobol_g`)**.** Fix $r$ and $\ell_*$, $\widehat{\omega}$ and $\mathring{\omega}$ describing $\mathcal{C}$ in (3.6). Given an absolute error tolerance, $\varepsilon_a$, initialize $m = \ell_* + r$ and do:

**Step 1.** Compute $\widetilde{S}_{m-r,m}(f)$.

**Step 2.** Check whether $\mathfrak{C}(m)\widetilde{S}_{m-r,m}(f) \le \varepsilon_a$. If true, return $\widehat{I}_m(f)$ defined in (3.1). If not, increment $m$ by one, and go to Step 1.

In [10], we prove the following theorem about the convergence of the automatic cubature and its upper bound on the computational complexity:

**Theorem 3.1.** *For* $m = \min\{m' \ge \ell_* + r : \mathfrak{C}(m')\widetilde{S}_{m'-r,m'}(f) \le \varepsilon_a\}$*, Algorithm 3 is successful whenever* $f \in \mathcal{C}$*,*

$$\left|I(f) - \widehat{I}_m(f)\right| \le \varepsilon_a. \tag{3.11}$$

*Thus, the number of function data needed is* $b^m$*. Defining* $m^* = \min\{m' \ge \ell_* + r : \mathfrak{C}(m')[1 + \widehat{\omega}(r)\mathring{\omega}(r)]S_{m'-r}(f) \le \varepsilon_a\}$*, we also have* $b^m \le b^{m^*}$*. This means that the computational cost can be bounded,*

$$\operatorname{cost}\left(\widehat{I}, f, \varepsilon_a\right) \le \$(f)b^{m^*} + cm^*b^{m^*}$$

*where* $\$(f)$ *is the cost of evaluating* $f$ *at one data point.*

*Proof.* By construction, the algorithm must be successful. Recall that the inequality used for building the algorithm is (3.9).

To find the upper bound on the computational cost, a similar result to (3.7) provides

$$\widetilde{S}_{\ell,m}(f) = \sum_{\kappa=b^{\ell-1}}^{b^\ell-1} \left|\tilde{f}_{m,\kappa}\right| = \sum_{\kappa=b^{\ell-1}}^{b^\ell-1} \left|\hat{f}_\kappa + \sum_{\lambda=1}^{\infty} \hat{f}_{\kappa+\lambda b^m} \varphi_{\tilde{\boldsymbol{k}}(\kappa+\lambda b^m)-\tilde{\boldsymbol{k}}(\kappa)}(\boldsymbol{\Delta})\right|$$

$$\leq \sum_{\kappa=b^{\ell-1}}^{b^\ell-1} \left|\hat{f}_\kappa\right| + \sum_{\kappa=b^{\ell-1}}^{b^\ell-1} \sum_{\lambda=1}^{\infty} \left|\hat{f}_{\kappa+\lambda b^m}\right| = S_\ell(f) + \widehat{S}_{\ell,m}(f)$$
$$\leq [1 + \hat{\omega}(m-\ell)\mathring{\omega}(m-\ell)] S_\ell(f).$$

Replacing $\widetilde{S}_{\ell,m}(f)$ in the error bound in (3.9) proves that the choice of $m$ needed to satisfy the tolerance is no greater than $m^*$ defined above.

Since we use the Fast Fourier Transform, the computation cost of $\widetilde{S}_{m-r,m}(f)$ is described in terms of $\mathcal{O}(mb^m)$ operations. Thus, the total cost of Algorithm 3 is,

$$\text{cost}\left(\widehat{I}_m, f, \varepsilon_a\right) \leq \$(f) b^{m^*} + c m^* b^{m^*}$$

□

We implemented Algorithm 3 in Matlab for digital sequences as `cubSobol_g`, and for rank-1 lattice sequences as `cubLattice_g`. The first uses Matlab's Owen scrambled Sobol' sequences [23], and the second one Dirk Nuyens' generator from `https://people.cs.kuleuven.be/~dirk.nuyens/qmc-generators/genvecs/exod2_base2_m20.txt`. Both functions are part of the Guaranteed Automatic Integration Library (GAIL) [24]. They estimate the integral given a user-input absolute error tolerance $\varepsilon_a$. However, one may need to satisfy other tolerance conditions such as the relative error tolerance $\varepsilon_r$. Chapter 4 generalizes the tolerance as a function of $\varepsilon_a$ and $\varepsilon_r |I(f)|$.

## 3.4 Numerical Examples

**3.4.1 Keister's Example.** The default algorithm parameters are

$$\ell_* = 6, \qquad r = 4, \qquad \mathfrak{C}(m) = 5 \times 2^{-m}, \tag{3.12}$$

and mapping $\tilde{\boldsymbol{k}}$ is fixed heuristically according to Algorithm 1 or 2. Fixing $\mathfrak{C}$ partially determines $\widehat{\omega}$ and $\mathring{\omega}$ since $\widehat{\omega}(m) = \mathfrak{C}(m)/\widehat{\omega}(r)$ and $\widehat{\omega}(r)\mathring{\omega}(r) = \mathfrak{C}(r)/[1 + \mathfrak{C}(r)]$.

To test the implementations from [24], we apply both cubatures `cubSobol_g` and `cubLattice_g` to the integrand presented in [25],

$$\int_{\mathbb{R}^d} \mathrm{e}^{-\|\boldsymbol{t}\|^2} \cos(\|\boldsymbol{t}\|)\, \mathrm{d}\boldsymbol{t} = \pi^{d/2} \int_{[0,1]^d} \cos\left(\sqrt{\frac{1}{2}\sum_{j=1}^{d} [\Phi^{-1}(x_j)]^2}\right) \mathrm{d}\boldsymbol{x}. \tag{3.13}$$

In the above integral, $\Phi$ is the standard Gaussian distribution function.

We set the absolute error tolerance to $\varepsilon_a = 0.002$ and generate 500 IID random values of the dimension $d = \lfloor \mathrm{e}^D \rfloor$, with $D$ uniformly distributed between 0 and $\log(20)$. Each time, a new scrambled and shifted Sobol' sequence, and a shifted rank-1 lattice was used.

Figure 3.2 shows the results for `cubSobol_g`. In 96.4% of the integrations, the absolute error tolerance was met. For the cases where the tolerance was not met, the function did not lie inside the cone $\mathcal{C}$. Because the mapping $\tilde{\boldsymbol{k}}(\kappa)$ is chosen heuristically, it depends on the choice of sequence used and hence, the cone $\mathcal{C}$ also depends on the sequence.

The red dots are each of the 500 integrations performed, and the vertical dashed line shows the tolerance limit at $\varepsilon_a = 0.002$ (on the right side, it means the cubature failed to meet the tolerance). In addition, the green line is the empirical distribution of the integration errors and the magenta one, the empirical distribution of the computing times.

Figure 3.3 presents the results for the algorithm `cubLattice_g`. Again, the algorithm succeeded 99.2% of the time.

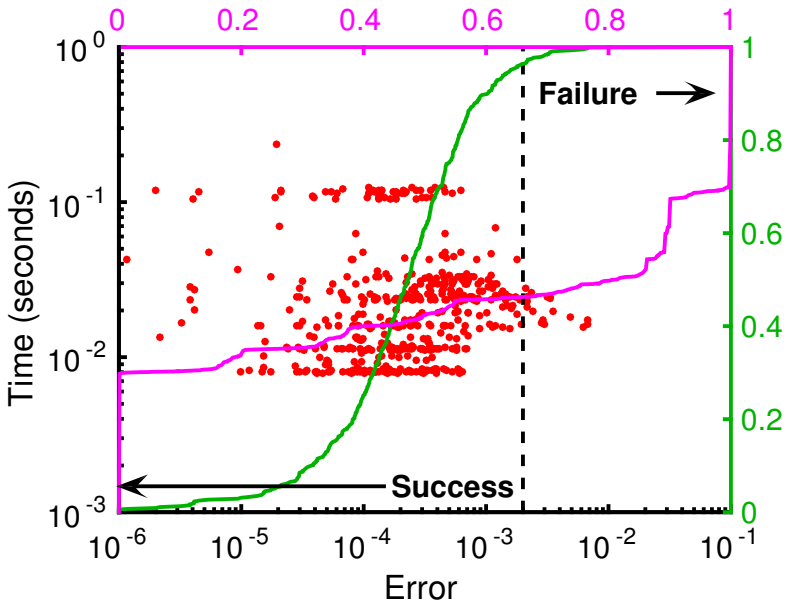


Figure 3.2. Results of integrating (3.13) using `cubSobol_g` with a 96.4% success rate. The dashed line is set to the error tolerance $\varepsilon_a = 0.002$. Green and magenta lines show the empirical distribution functions of errors and times respectively.

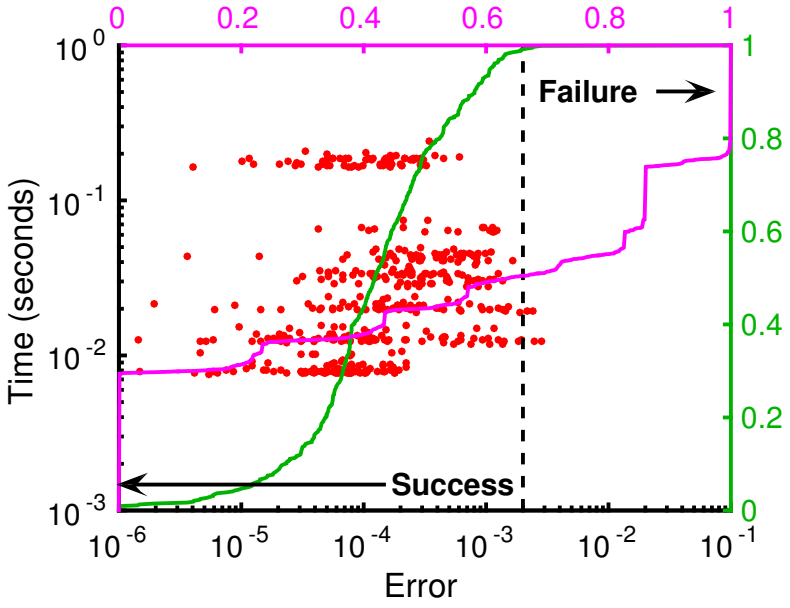


Figure 3.3. Results of integrating (3.13) using `cubLattice_g` with a 99.2% success rate. The dashed line is set to the error tolerance $\varepsilon_a = 0.002$. Green and magenta lines show the empirical distribution functions of errors and times respectively.

We also compare the results with another technique called randomized quasi-Monte Carlo. In this case for each $m$, instead of using the discrete Fourier coefficients, we estimate the error by applying the central limit theorem. Given a fixed number of points $b^m$, we estimate our integral using $N$ IID sequences, either scrambled and digitally shifted Sobol' sequences, or shifted rank-1 lattice sequences. If we denote by $\hat{\mu}_j$ each estimate,

$$\lim_{N\to\infty} \frac{\frac{1}{N}\sum_{j=1}^{N}\hat{\mu}_j - I(f)}{\sqrt{\mathrm{Var}(\hat{\mu}_1)/N}} \sim N(0,1).$$

Thus, for a confidence of $1-\alpha$ and $N$ large enough,

$$\mathbb{P}\left(\left|\frac{1}{N}\sum_{j=1}^{N}\hat{\mu}_j - I(f)\right| \le \Phi^{-1}(1-\alpha/2)\sqrt{\mathrm{Var}(\hat{\mu}_1)/N}\right) \approx 1-\alpha,$$

with $\Phi(x)$ the standard Gaussian distribution function. Because $\mathrm{Var}(\hat{\mu}_1)$ is not known, we use the unbiased sample variance estimator, $\widehat{\mathrm{Var}}(\hat{\mu}_1)$. In addition, we consider an inflation factor $\mathfrak{Q}$ that adjusts the error bound as $\mathfrak{Q}\Phi^1(1-\alpha/2)\sqrt{\widehat{\mathrm{Var}}(\hat{\mu}_1)/N}$. Starting from $m = 10$, if $\mathfrak{Q}\Phi^1(1-\alpha/2)\sqrt{\widehat{\mathrm{Var}}(\hat{\mu}_1)/N} \le \varepsilon_a$ the algorithm returns $\frac{1}{N}\sum_{j=1}^{N}\hat{\mu}_j$. Otherwise, we increase $m$ by one and repeat the process. This technique does not guarantee the absolute error tolerance condition (3.11). First, since we do not know $\mathrm{Var}(\hat{\mu}_1)$, we need to estimate it. Furthermore, the central limit theorem is an asymptotic result that does not tell us how big $N$ should be. In addition, the number of function evaluations is $N$ times higher for the randomized algorithms, which is not convenient for the applications where the integrand is costly to evaluate. One example is the top quark mass measurement at Fermilab where millions of integrands need to be estimated and the average evaluation time is 0.01 seconds. To estimate one integral with $2^{10}$ points, our adaptive cubatures would take around 100 seconds, but the randomized algorithms $100N$ seconds ($N$ should be at least 10).

We apply this technique to (3.13). The confidence is fixed to 95% ($\alpha = 5\%$)

since it is approximately the success rate of `cubSobol_g` and `cubLattice_g`. The additional parameters are $N = 30$, and $\mathfrak{Q} = 1.2$. Figure 3.4 shows the results for the scrambled Sobol' sequences while Figure 3.5 uses shifted rank-1 lattices. The first succeeded to meet the tolerance 99.8% of the time while the second 100%. The fact that both values are greater than the expected 95% means that quasi-Monte Carlo stratification is helpful. Even though both randomized algorithms show exceptional precision, computation times are slower. In average, the randomized Sobol' algorithm is 7 times slower than `cubSobol_g`, and the randomized lattices algorithm is 1.14 times slower than `cubLattice_g`. This big difference in computation time for the Sobol' case is due to the cost of generating the Sobol' points.

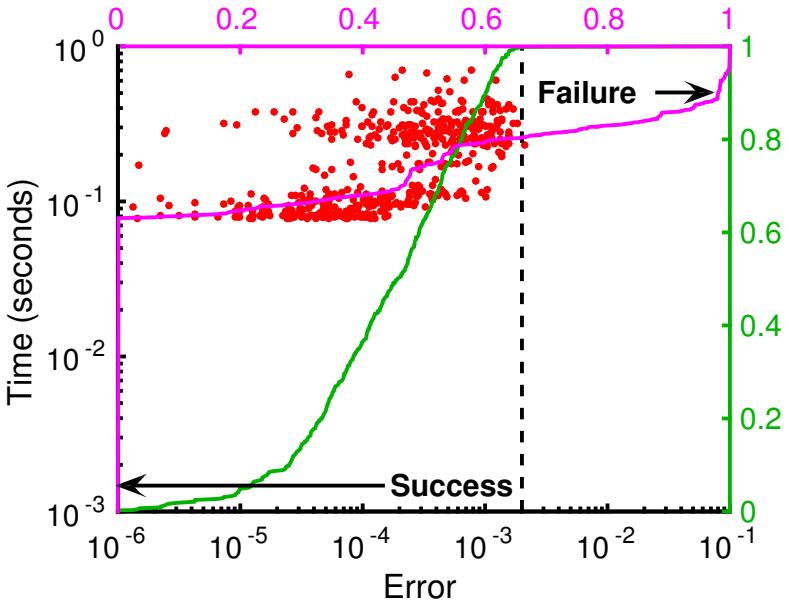


Figure 3.4. Results of integrating (3.13) using the randomized quasi-Monte Carlo with scrambled and digitally shifted Sobol' sequences. The success rate is 99.8% and the dashed line is set to the error tolerance $\varepsilon_a = 0.002$. Green and magenta lines show the empirical distribution functions of errors and times respectively.

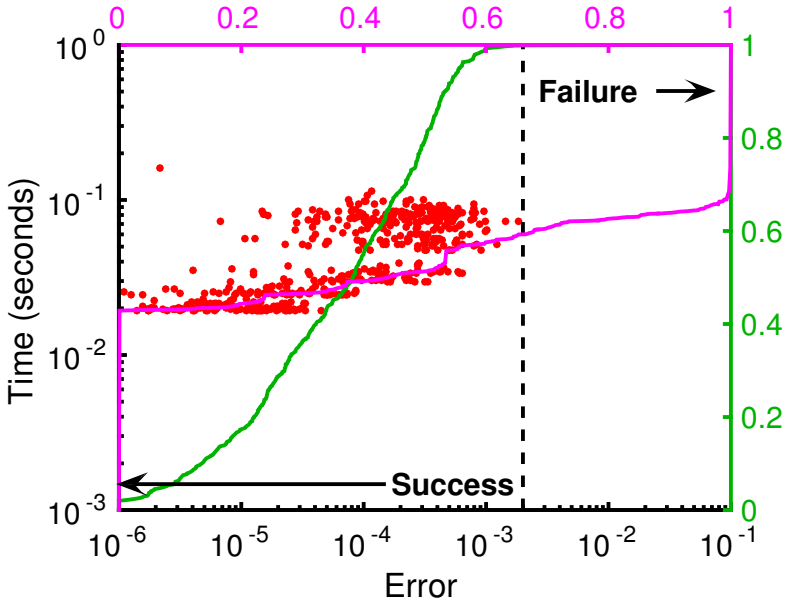


Figure 3.5. Results of integrating (3.13) using the randomized quasi-Monte Carlo with shifted rank-1 lattice sequences. The success rate is 100% and the dashed line is set to the error tolerance $\varepsilon_a = 0.002$. Green and magenta lines show the empirical distribution functions of errors and times respectively.

**3.4.2 Geometric Mean Asian Call Option.** In this Section, we price a geometric mean Asian call option with the same algorithm parameters as for the Keister example, (3.12). The integrand for this option is,

$$f(\boldsymbol{x}) = \mathrm{e}^{-rT} \max\left(\left(\prod_{i=1}^{d} S_{t_i}(\boldsymbol{x})\right)^{1/d} - K, 0\right), \tag{3.14}$$
$$S_{t_i}(\boldsymbol{x}) = S_0 \mathrm{e}^{(r-\sigma^2/2)t_i + \sigma \sum_{j=1}^{i} \Phi^{-1}(x_j)\sqrt{t_j - t_{j-1}}},$$

where $\Phi(x)$ refers to the standard Gaussian distribution function. The definition of $S_{t_i}(\boldsymbol{x})$ is not unique and corresponds to the Cholesky decomposition explained in Section 5.4.2.

The option parameters are set to $t_i = T \times i/d$ for $i = 0, \ldots, d$, $S_0 = K = 100$, $T = 1$, and $r = 3\%$. The test is performed on 500 samples whose dimensions are

chosen IID uniformly among $1, 2, 4, 8, 16, 32$, and $64$, and the volatility $\sigma$ is also chosen IID uniformly from 10% to 70%. We set the absolute error tolerance to one cent, $\varepsilon_a = \$0.01$. Because the integrand (3.14) is not periodic, when we use rank-1 lattices we apply the Baker's transform as detailed in Section 2.3.1.

For this example, `cubSobol_g` had a success of 100%, and `cubLattice_g` of 99.8%. Results are presented in Figures 3.6 and 3.7 respectively. With the same parametrization as before, the randomized quasi-Monte Carlo algorithms succeeded 99.2% and 100% of the time. Again, although both randomized algorithms seem to have good precision, the computation time is higher for the randomized case. In average, the randomized Sobol' algorithm is 4.26 times slower than `cubSobol_g` and the randomized lattices algorithm is 1.35 times slower than `cubLattice_g`. The randomized quasi-Monte Carlo results are shown in Figures 3.8 and 3.9.

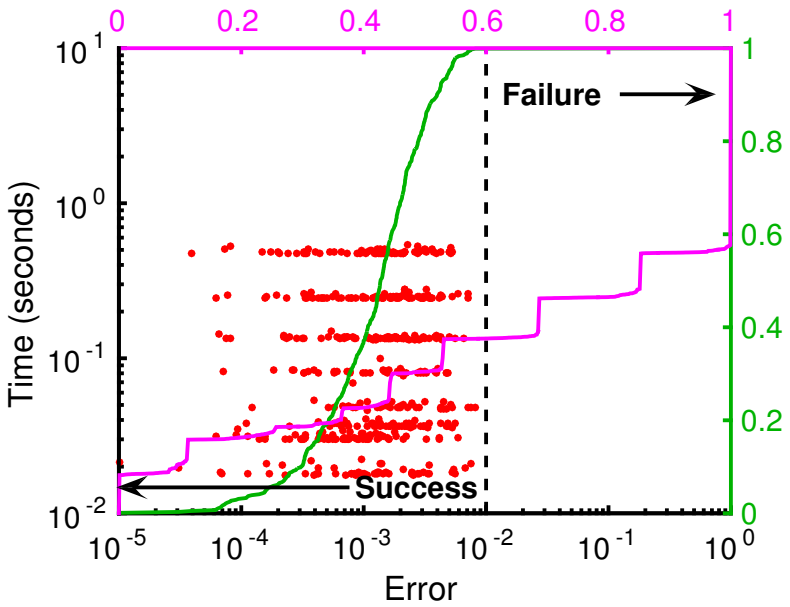


Figure 3.6. Results of integrating (3.14) using `cubSobol_g` with a 100% success rate. The dashed line is set to the error tolerance $\varepsilon_a = 0.01$. Green and magenta lines show the empirical distribution functions of errors and times respectively.

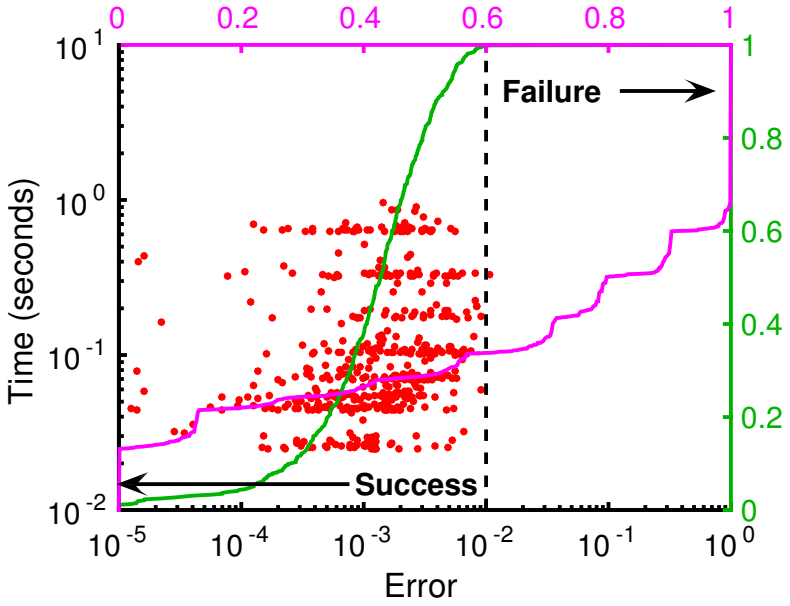


Figure 3.7. Results of integrating (3.14) using `cubLattice_g` with a 99.8% success rate. The dashed line is set to the error tolerance $\varepsilon_a = 0.01$. Green and magenta lines show the empirical distribution functions of errors and times respectively.

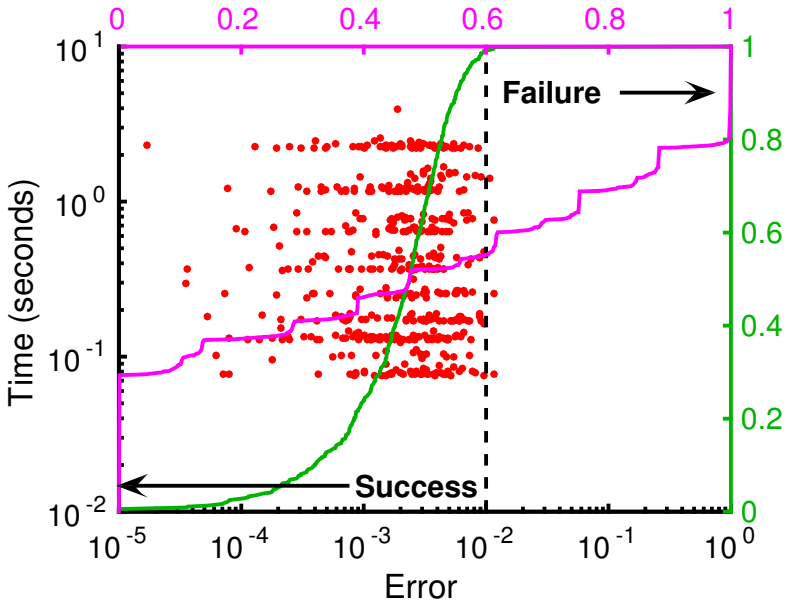


Figure 3.8. Results of integrating (3.14) using randomized quasi-Monte Carlo with scrambled and digitally shifted Sobol' sequences. The success rate is 99.2% and the dashed line is set to the error tolerance $\varepsilon_a = 0.01$. Green and magenta lines show the empirical distribution functions of errors and times respectively.

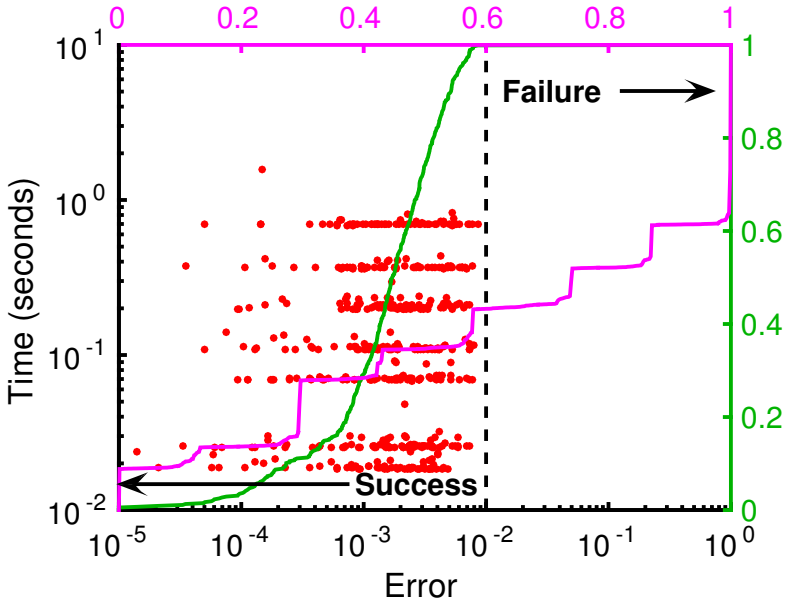


Figure 3.9. Results of integrating (3.14) using the randomized quasi-Monte Carlo with shifted rank-1 lattice sequences. The success rate is 100% and the dashed line is set to the error tolerance $\varepsilon_a = 0.01$. Green and magenta lines show the empirical distribution functions of errors and times respectively.

**3.4.3 Multivariate Gaussian Probabilities.** For this example, we estimate the probability of a multivariate Gaussian random variable to fall inside a given hyper-rectangle. In this case, for a mean $\boldsymbol{\mu} = \mathbf{0}$ and a covariance matrix $\Sigma \in \mathbb{R}^{d\times d}$, we are interested in approximating the value of

$$\mathbb{P}\left(\boldsymbol{a} \leq \boldsymbol{X} \leq \boldsymbol{b}\right) = \int_{\boldsymbol{a}}^{\boldsymbol{b}} \frac{\mathrm{e}^{\boldsymbol{x}^t \Sigma^{-1} \boldsymbol{x}}}{(2\pi)^{d/2} \left|\Sigma\right|^{1/2}} \,\mathrm{d}\boldsymbol{x}. \tag{3.15}$$

The algorithm parameters for this example are the same as before, (3.12). As opposed to the previous examples, Keister and geometric mean Asian call option, we do not know how to explicitly find the solution of (3.15). Although we cannot check if the absolute error tolerance is met, we see that applying Genz's transform [7] for $d = 1000$, $\boldsymbol{a}$ and $\boldsymbol{b}$ chosen randomly, $\Sigma = 0.4\,\mathsf{I} + 0.6\,\mathbf{1}\mathbf{1}^T$, and $\varepsilon_a = 10^{-3}$, `cubSobol_g` takes around 2.75 seconds. In comparison, estimating the same probability using the

IID MC algorithm `meanMC_g` also in GAIL [24] takes 313 seconds which suggests that quasi-Monte Carlo can be more efficient than IID Monte Carlo.

# CHAPTER 4

# GENERALIZED TOLERANCE FUNCTION

## 4.1 How to Generalize the Tolerance Function

Algorithm 3 automatically stops when the absolute error tolerance is met. This means that,

$$\left| I(f) - \widehat{I}(f, \varepsilon_a) \right| \leq \varepsilon_a.$$

Note that to construct the automatic Algorithm 3, namely $\widehat{I}$, we built a theoretical error bound $\widehat{\varepsilon}_m$ such that $\left| I(f) - \widehat{I}_m(f) \right| \leq \widehat{\varepsilon}_m(f)$. Hence, Algorithm $\widehat{I}(f)$ finds the minimum $m$ such that $\widehat{\varepsilon}_m(f) \leq \varepsilon_a$ and returns $\widehat{I}_m(f)$.

Nonetheless, we may also want to satisfy other hybrid error conditions that can also consider relative error tolerances $0 \leq \varepsilon_r \leq 1$. Indeed, the tolerance function generalization constructed in this chapter will work for any approximation problem involving a functional $I$ and its approximation algorithm $\widehat{I}$. The only requirement is to obtain a parametrized error bound $\widehat{\varepsilon}_m$ for the estimator $\widehat{I}_m$, satisfying $\widehat{\varepsilon}_m \to 0$ as $m \to \infty$.

We start by defining a tolerance function whose inputs will be $\varepsilon_a$ and $\varepsilon_r \left| I(f) \right|$, namely $\text{tol}(\varepsilon_a, \varepsilon_r \left| I(f) \right|)$. Then, let $\text{tol}(t, s) : \mathbb{R}^+ \times \mathbb{R}^+ \to \mathbb{R}^+$ be Lipschitz-1 in $s$ and non-decreasing in both arguments, i.e. $\text{tol}(t, s) \leq \text{tol}(t', s')$ for $t \leq t'$ and $s \leq s'$. Two examples of functions are $\text{tol}(t, s) = \max(t, s)$, and $\text{tol}(t, s) = \theta t + (1 - \theta)s$, $0 \leq \theta \leq 1$. In addition, if we want our algorithm $\widehat{I}$ to better satisfy the generalized tolerance function, we have to bias our estimator $\widehat{I}_m$.

**Definition 4.1.** *The new biased estimator $\widetilde{I}_m$ is,*

$$\widetilde{I}_m(f) := \widehat{I}_m(f) + \Delta_{m,-}(f)\,,$$

$$\Delta_{m,\pm}(f) := \frac{1}{2}\left[ \text{tol}\left( \varepsilon_a, \varepsilon_r \left| \widehat{I}_m(f) - \widehat{\varepsilon}_m(f) \right| \right) \pm \text{tol}\left( \varepsilon_a, \varepsilon_r \left| \widehat{I}_m(f) + \widehat{\varepsilon}_m(f) \right| \right) \right].$$

For this new estimator $\widetilde{I}_m$, the stopping criteria is a direct consequence of Lemma 4.1.

**Lemma 4.1.** *If* $\widehat{\varepsilon}_m(f) \leq \Delta_{m,+}(f)$*, then*

$$\left|I(f) - \widetilde{I}_m(f)\right| \leq \text{tol}(\varepsilon_a, \varepsilon_r \left|I(f)\right|)$$

*Thus, for the new algorithm* $\widetilde{I}$ *based on* $\widetilde{I}_m$ *and* $\widehat{\varepsilon}_m$*, we increase* $m$ *until* $\widehat{\varepsilon}_m(f) \leq \Delta_{m,+}(f)$*.*

*Proof.* For simplicity, we will drop the dependence on $f$. Firstly,

$$\begin{aligned}
0 = \pm\left(\widetilde{I}_m - \widehat{I}_m - \Delta_{m,-}\right) &\leq \Delta_{m,+} - \widehat{\varepsilon}_m \\
&\iff \widehat{I}_m + \Delta_{m,-} - \Delta_{m,+} + \widehat{\varepsilon}_m \leq \widetilde{I}_m \leq \widehat{I}_m + \Delta_{m,-} + \Delta_{m,+} - \widehat{\varepsilon}_m \\
&\iff \widehat{I}_m - \text{tol}\left(\varepsilon_a, \varepsilon_r \left|\widehat{I}_m + \widehat{\varepsilon}_m\right|\right) + \widehat{\varepsilon}_m \leq \widetilde{I}_m \leq \widehat{I}_m + \text{tol}\left(\varepsilon_a, \varepsilon_r \left|\widehat{I}_m - \widehat{\varepsilon}_m\right|\right) - \widehat{\varepsilon}_m.
\end{aligned}$$

Since tol is Lipschitz-1 in $s$ and $\varepsilon_r \leq 1$, then $c \to c \pm \text{tol}(\varepsilon_a, \varepsilon_r \left|c\right|)$ is non-decreasing. To prove it we need to show that if $c' \geq c$, then $c' \pm \text{tol}(\varepsilon_a, \varepsilon_r \left|c'\right|) \geq c \pm \text{tol}(\varepsilon_a, \varepsilon_r \left|c\right|)$. However,

$$\begin{aligned}
c' - c \pm \text{tol}(\varepsilon_a, \varepsilon_r \left|c'\right|) \mp \text{tol}(\varepsilon_a, \varepsilon_r \left|c\right|) & \\
&\geq c' - c - \left|\text{tol}(\varepsilon_a, \varepsilon_r \left|c'\right|) - \text{tol}(\varepsilon_a, \varepsilon_r \left|c\right|)\right| \\
&\overset{Lip-1}{\geq} c' - c - \varepsilon_r \left||c'| - |c|\right| \\
&\overset{\varepsilon_r \leq 1}{\geq} c' - c - |c' - c| = 0.
\end{aligned}$$

Hence, $c \to c \pm \text{tol}(\varepsilon_a, \varepsilon_r \left|c\right|)$ is non-decreasing. If we consider $c = I$ and $c' = \widehat{I}_m + \widehat{\varepsilon}_m$ on the left, and $c = \widehat{I}_m - \widehat{\varepsilon}_m$ and $c' = I$ on the right,

$$I - \text{tol}\left(\varepsilon_a, \varepsilon_r \left|I\right|\right) \leq \widehat{I}_m + \widehat{\varepsilon}_m - \text{tol}\left(\varepsilon_a, \varepsilon_r \left|\widehat{I}_m + \widehat{\varepsilon}_m\right|\right) \tag{4.1}$$

$$\widehat{I}_m - \widehat{\varepsilon}_m + \text{tol}\left(\varepsilon_a, \varepsilon_r \left|\widehat{I}_m - \widehat{\varepsilon}_m\right|\right) \leq I + \text{tol}\left(\varepsilon_a, \varepsilon_r \left|I\right|\right) \tag{4.2}$$

Because we assume that $\widehat{\varepsilon}_m \leq \Delta_{m,+}$, then $\widehat{\varepsilon}_m - \text{tol}\left(\varepsilon_a, \varepsilon_r \left|\widehat{I}_m + \widehat{\varepsilon}_m\right|\right) \leq -\widehat{\varepsilon}_m + \text{tol}\left(\varepsilon_a, \varepsilon_r \left|\widehat{I}_m - \widehat{\varepsilon}_m\right|\right)$. Therefore, averaging (4.1) and (4.2),

$$I - \text{tol}\left(\varepsilon_a, \varepsilon_r \left|I\right|\right) \leq \widetilde{I}_m \leq I + \text{tol}\left(\varepsilon_a, \varepsilon_r \left|I\right|\right) \Longleftrightarrow \left|I - \widetilde{I}_m\right| \leq \text{tol}\left(\varepsilon_a, \varepsilon_r \left|I\right|\right)$$

□

An interesting remark is that $\Delta_{m,-}$ is always shrinking $\widehat{I}_m$ into $\widetilde{I}_m$. In other words, $\widehat{I}_m$ and $\Delta_{m,-}$ have always opposite signs. This means that $\widetilde{I}_m$ is $\widehat{I}_m$ biased towards 0. The purpose is to facilitate the generalized tolerance to be met. For instance, if we consider the pure relative error case, the bias is helpful because

$$\sup_{I \in [\widehat{I}_m - \varepsilon, \widehat{I}_m + \varepsilon]} \left|\frac{I - \widetilde{I}_m}{I}\right| \leq \sup_{I \in [\widehat{I}_m - \varepsilon, \widehat{I}_m + \varepsilon]} \left|\frac{I - \widehat{I}_m}{I}\right|.$$

As in Theorem 3.1, we also provide an upper bound on the computational cost.

**Theorem 4.2.** *Let* $m^* = \min\{m \geq 0 : \widehat{\varepsilon}_m(f) \leq \text{tol}(\varepsilon_a, \varepsilon_r \left|I(f)\right|)/(1 + \varepsilon_r)\}$. *Then,* $\widehat{\varepsilon}_{m^*}(f) \leq \Delta_{m^*,+}(f)$ *and*

$$\text{cost}\left(\widetilde{I}, f, \varepsilon_a, \varepsilon_r\right) \leq \$(f) b^{m^*} + c m^* b^{m^*}$$

*where* $\$(f)$ *is the cost of evaluating* $f$ *at one data point.*

*Proof.* Consider

$$\left.\begin{aligned} \widehat{I}_m > 0 \Rightarrow |I| \leq \left|\widehat{I}_m + \widehat{\varepsilon}_m\right| \\ \widehat{I}_m < 0 \Rightarrow |I| \leq \left|\widehat{I}_m - \widehat{\varepsilon}_m\right| \end{aligned}\right\} \Longrightarrow |I| \leq \left|\widehat{I}_m + \operatorname{sign}\left(\widehat{I}_m\right)\widehat{\varepsilon}_m\right|.$$

This means that,

$$\operatorname{tol}\left(\varepsilon_a, \varepsilon_r \left|\widehat{I}_m + \operatorname{sign}\left(\widehat{I}_m\right)\widehat{\varepsilon}_m\right|\right) \geq \operatorname{tol}\left(\varepsilon_a, \varepsilon_r |I|\right) \tag{4.3}$$

In addition, by the Lipschitz condition, $\operatorname{tol}(t, s') \geq \operatorname{tol}(t, s) - |s - s'|$. If we choose $s' = \varepsilon_r \left|\widehat{I}_m \pm \widehat{\varepsilon}_m\right|$ and $s = \varepsilon_r |I|$,

$$\begin{aligned} \operatorname{tol}\left(\varepsilon_a, \varepsilon_r \left|I - I + \widehat{I}_m \pm \widehat{\varepsilon}_m\right|\right) &\geq \operatorname{tol}\left(\varepsilon_a, \varepsilon_r |I|\right) - \varepsilon_r \left||I| - \left|\widehat{I}_m \pm \widehat{\varepsilon}_m\right|\right| \\ &\geq \operatorname{tol}\left(\varepsilon_a, \varepsilon_r |I|\right) - \varepsilon_r \left|-I + \widehat{I}_m \pm \widehat{\varepsilon}_m\right| \end{aligned}$$

Then,

$$\begin{aligned} \operatorname{tol}\left(\varepsilon_a, \varepsilon_r \left|\widehat{I}_m - \operatorname{sign}\left(\widehat{I}_m\right)\widehat{\varepsilon}_m\right|\right) &= \operatorname{tol}\left(\varepsilon_a, \varepsilon_r \left|I - I + \widehat{I}_m - \operatorname{sign}\left(\widehat{I}_m\right)\widehat{\varepsilon}_m\right|\right) \\ &\geq \operatorname{tol}\left(\varepsilon_a, \varepsilon_r |I|\right) - \varepsilon_r \left|-I + \widehat{I}_m - \operatorname{sign}\left(\widehat{I}_m\right)\widehat{\varepsilon}_m\right| \\ &\geq \operatorname{tol}\left(\varepsilon_a, \varepsilon_r |I|\right) - 2\varepsilon_r\widehat{\varepsilon}_m \end{aligned} \tag{4.4}$$

Noticing that

$$\Delta_{m,+} = \frac{1}{2}\left[\operatorname{tol}\left(\varepsilon_a, \varepsilon_r \left|\widehat{I}_m - \operatorname{sign}\left(\widehat{I}_m\right)\widehat{\varepsilon}_m\right|\right) + \operatorname{tol}\left(\varepsilon_a, \varepsilon_r \left|\widehat{I}_m + \operatorname{sign}\left(\widehat{I}_m\right)\widehat{\varepsilon}_m\right|\right)\right],$$

and taking the average of equations (4.3) and (4.4), we find that

$$\Delta_{m,+} \geq \operatorname{tol}\left(\varepsilon_a, \varepsilon_r |I|\right) - \varepsilon_r\widehat{\varepsilon}_m.$$

Finally, by the assumption,

$$\Delta_{m^*,+} - \widehat{\varepsilon}_{m^*} \geq \operatorname{tol}(\varepsilon_a, \varepsilon_r |I|) - (1+\varepsilon_r)\widehat{\varepsilon}_{m^*} \geq 0.$$

By Lemma 4.1, if $\Delta_{m^*,+}(f) \geq \widehat{\varepsilon}_{m^*}(f)$, then $\left|I(f) - \widetilde{I}_{m^*}(f)\right| \leq \operatorname{tol}(\varepsilon_a, \varepsilon_r |I(f)|)$ and hence, the computational cost. □

### 4.2 Numerical Examples

We adapted the algorithms `cubSobol_g` and `cubLattice_g` to accept the error tolerances of the form $\operatorname{tol}(\varepsilon_a, \varepsilon_r |I(f)|) = \max(\varepsilon_a, \varepsilon_r |I(f)|)$, and $\operatorname{tol}(\varepsilon_a, \varepsilon_r |I(f)|) = \theta\varepsilon_a + (1-\theta)\varepsilon_r |I(f)|$, for any $0 \leq \theta \leq 1$. Note that pure absolute or relative error tolerances correspond to the second function with $\theta = 1$ and $\theta = 0$ respectively.

The numerical examples of this section are the same as in Section 3.4. The only difference is that we test the algorithms for pure relative error tolerance with $\varepsilon_r = 0.002$.

In Figure 4.1, the results for `cubSobol_g` show a success of 95.2%. Because the actual value of the tolerance function changes as a function of $|I(f)|$, we plot the normalized errors $\left|I(f) - \widetilde{I}(f, \varepsilon_a, \varepsilon_r)\right| / \operatorname{tol}(\varepsilon_a, \varepsilon_r |I(f)|)$ instead. Thus, if these values are equal or smaller than 1, we succeed to meet the tolerance condition $\left|I(f) - \widetilde{I}(f, \varepsilon_a, \varepsilon_r)\right| \leq \operatorname{tol}(\varepsilon_a, \varepsilon_r |I(f)|)$.

For `cubLattice_g`, the success is of 98.2% as seen in Figure 3.3.

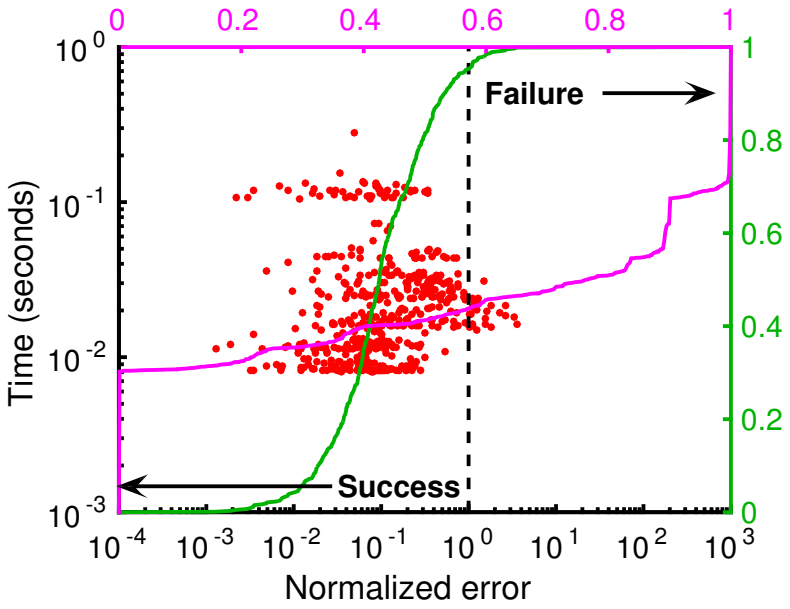


Figure 4.1. Results of integrating (3.13) using `cubSobol_g` with pure relative error tolerance $\varepsilon_r = 0.002$. The success rate is 95.2%. Green and magenta lines show the empirical distribution functions of normalized errors and times respectively.

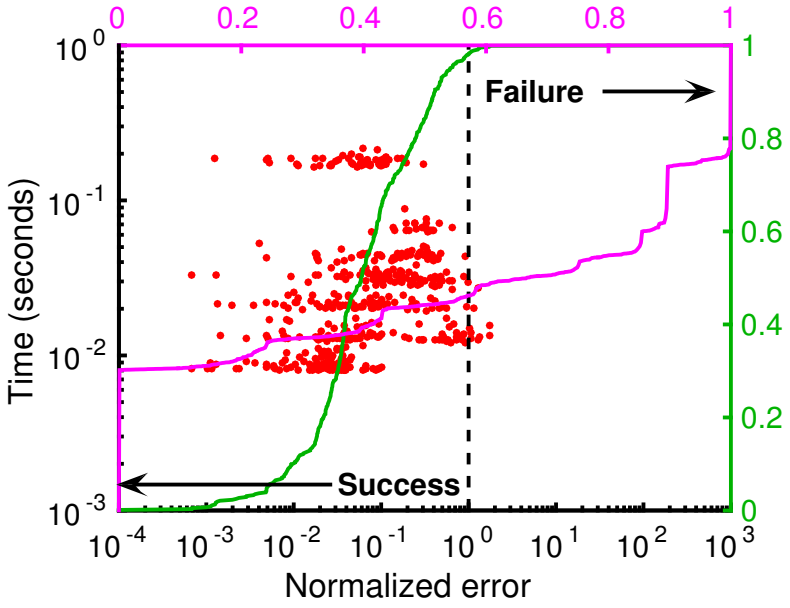


Figure 4.2. Results of integrating (3.13) using `cubLattice_g` with pure relative error tolerance $\varepsilon_r = 0.002$. The success rate is 98.2%. Green and magenta lines show the empirical distribution functions of normalized errors and times respectively.

# CHAPTER 5

# SOBOL' INDICES

The work discussed in this chapter was done in collaboration with Laurent Gilquin, from Inria Grenoble, and submitted for publication [12].

## 5.1 Sobol' Indices

In 1990, Sobol' introduced the sensitivity indices [26] for the first time. These indices quantify what part of the overall variance of a random model $f(\boldsymbol{X})$ is explained by each subset of inputs. They can be estimated using their integral formulation and since the models are assumed to have several dimensions, quasi-Monte Carlo methods are a good choice. Thus, in this chapter we define these quantities and extend `cubSobol_g` to estimate first-order and total effect indices.

Let $f \in L^2([0,1]^d)$ be a function where $f(\boldsymbol{x})$ is defined for all $\boldsymbol{x} \in [0,1]^d$, $\boldsymbol{x} = (x_1, \ldots, x_d)$. Denote by $\mathcal{D} = \{1, \ldots, d\}$ the set of coordinate indexes. As described in Sobol' [26], the random variable $f(\boldsymbol{X})$ is modeled by assuming $\boldsymbol{X}$ to be uniformly distributed on $[0,1]^d$. However, in this chapter we will simply consider $f$ to be a function defined over $[0,1]^d$.

If $u$ is a subset of $\mathcal{D}$, we note by $-u$ its complement and by $|u|$ its cardinality. Under this notation, $\boldsymbol{x}_u$ represents a point in $[0,1]^{|u|}$ with components $x_j$, $j \in u$. Given two points $\boldsymbol{x}$ and $\boldsymbol{x}'$, we also define the hybrid point $\boldsymbol{w} = (\boldsymbol{x}_u : \boldsymbol{x}'_{-u}) \in [0,1]^d$ as $w_j = x_j$ if $j \in u$ and $w_j = x'_j$ if $j \notin u$.

For $u \subset \mathcal{D}$, we define $f_u(\boldsymbol{x})$ recursively as follows,

$$f_\varnothing := \int_{[0,1]^d} f(\boldsymbol{x}) \mathrm{d}\boldsymbol{x} = I(f), \qquad f_u(\boldsymbol{x}) := \int_{[0,1]^{|-u|}} f(\boldsymbol{x}) \mathrm{d}\boldsymbol{x}_{-u} - \sum_{v \subset u} f_v(\boldsymbol{x}). \tag{5.1}$$

The ANOVA decomposition of $f$ [27, 1] is,

$$f(\boldsymbol{x}) = \sum_{u \subseteq \mathcal{D}} f_u(\boldsymbol{x}). \tag{5.2}$$

which can be inferred from taking $u = \mathcal{D}$ in (5.1).

By construction (5.1), $\int_{[0,1]} f_u(\boldsymbol{x}) \mathrm{d}x_j = 0$ for any $j \in u$, [1]. This implies that the ANOVA decomposition is orthogonal, $\int_{[0,1]^d} f_u(\boldsymbol{x}) f_v(\boldsymbol{x}) \mathrm{d}\boldsymbol{x} = 0$ for $u \neq v$. Let $\sigma^2_{\emptyset}(f) = 0$, $\sigma^2_u(f) = \int_{[0,1]^{|u|}} f_u(\boldsymbol{x})^2 \mathrm{d}\boldsymbol{x}_u$, and $\sigma^2(f) = \int_{[0,1]^d} (f(\boldsymbol{x}) - I(f))^2 \mathrm{d}\boldsymbol{x}$. The variance decomposition of $f$ is defined as,

$$\sigma^2(f) = \sum_{u \subseteq \mathcal{D}} \sigma^2_u(f).$$

Using the variance decomposition, Sobol' introduced the quantities:

$$\underline{\tau}^2_u(f) = \sum_{v \subseteq u} \sigma^2_v(f), \qquad \overline{\tau}^2_u(f) = \sum_{v \cap u \neq \varnothing} \sigma^2_v(f), \qquad u \subset \mathcal{D}.$$

The above quantities $\underline{\tau}^2_u(f)$ and $\overline{\tau}^2_u(f)$ measure the importance of $\boldsymbol{x}_u$. The first quantifies what part of $\sigma^2(f)$ is explained by $\boldsymbol{x}_u$ alone, while the second quantifies what part of $\sigma^2(f)$ is explained by $\boldsymbol{x}_u$, including any interaction with $\boldsymbol{x}_{-u}$. Indeed, $\underline{\tau}^2_u(f)$ and $\overline{\tau}^2_u(f)$ satisfy the following properties: $0 \leq \underline{\tau}^2_u(f) \leq \overline{\tau}^2_u(f)$ and $\underline{\tau}^2_u(f) = \sigma^2(f) - \overline{\tau}^2_{-u}(f)$. These two measures are commonly found in the literature in their normalized form, $\underline{S}_u(f) = \underline{\tau}^2_u(f)/\sigma^2(f)$ as the $|u|$-order Sobol' index, and $\overline{S}_u(f) = \overline{\tau}^2_u(f)/\sigma^2(f)$ as the total effect Sobol' index of order $|u|$. I apologize for $S$ meaning two different things in the thesis.

In this chapter, we will estimate the normalized indices of order 1, namely $|u| = 1$, i.e. $u = \{j\}$ for some $j \in \mathcal{D}$. For $u \in \mathcal{D}$, one can also derive the integral

formula that defines the normalized indices,

$$\underline{S}_u(f) = \frac{\int_{[0,1]^{2d}} \left(f(\boldsymbol{x}) - f(\boldsymbol{x}_u : \boldsymbol{x'}_{-u})\right) f(\boldsymbol{x'}) \mathrm{d}\boldsymbol{x} \mathrm{d}\boldsymbol{x'}}{\int_{[0,1]^d} f(\boldsymbol{x})^2 \mathrm{d}\boldsymbol{x} - I(f)^2}, \tag{5.3}$$

$$\overline{S}_u(f) = \frac{\frac{1}{2}\int_{[0,1]^{d+1}} (f(\boldsymbol{x'}) - f(\boldsymbol{x}_u : \boldsymbol{x'}_{-u}))^2 \mathrm{d}\boldsymbol{x}_u \mathrm{d}\boldsymbol{x'}}{\int_{[0,1]^d} f(\boldsymbol{x})^2 \mathrm{d}\boldsymbol{x} - I(f)^2}. \tag{5.4}$$

Our extended `cubSobol_g` cubature will be based on estimating integrals (5.3) and (5.4).

## 5.2 Estimation of Sobol' Indices

Consider two independent well uniformly distributed point sets $\mathcal{P} = \{\boldsymbol{z}_i\}_{i=0}^{n-1}$ and $\mathcal{P}' = \{\boldsymbol{z'}_i\}_{i=0}^{n-1}$ in dimension $d$. The usual way to estimate (5.3) and (5.4) is

$$\widehat{\underline{S}}_u(f) = \frac{\widehat{\underline{\tau}^2_u}(f)}{\widehat{\sigma}^2(f)} = \frac{\frac{1}{n}\sum_{i=0}^{n-1} \left(f(\boldsymbol{z}_i) - f(\boldsymbol{z}_{i,u} : \boldsymbol{z'}_{i,-u})\right) f(\boldsymbol{z'}_i)}{\frac{1}{n}\sum_{i=0}^{n-1} f(\boldsymbol{z}_i)^2 - \left(\frac{1}{n}\sum_{i=0}^{n-1} f(\boldsymbol{z}_i)\right)^2}, \qquad u \subset \mathcal{D}, \quad |u| = 1 \tag{5.5}$$

$$\widehat{\overline{S}}_u(f) = \frac{\widehat{\overline{\tau}^2_u}(f)}{\widehat{\sigma}^2(f)} = \frac{\frac{1}{2n}\sum_{i=0}^{n-1} (f(\boldsymbol{z'}_i) - f(\boldsymbol{z}_{i,u} : \boldsymbol{z'}_{i,-u}))^2}{\frac{1}{n}\sum_{i=0}^{n-1} f(\boldsymbol{z}_i)^2 - \left(\frac{1}{n}\sum_{i=0}^{n-1} f(\boldsymbol{z}_i)\right)^2}, \qquad u \subset \mathcal{D}, \quad |u| = 1. \tag{5.6}$$

Using this estimation methodology, the approximation of a single pair ($\widehat{\underline{S}}_u(f)$, $\widehat{\overline{S}}_u(f)$) requires $3n$ evaluations of our function $f$. Thus, for all first-order and total effect indices, one requires $3nd$ function evaluations. Saltelli [28], using a combinatorial formalism, suggested reducing the number of function evaluations to $n(d+2)$ for the above estimator. The main idea is that there is no need to reevaluate $f(\boldsymbol{z}_i)$ and $f(\boldsymbol{z'}_i)$ for each $u$. One can simply evaluate $f(\boldsymbol{z}_i)$ $n$ times, $f(\boldsymbol{z'}_i)$ $n$ times, and $f(\boldsymbol{z}_{i,u} : \boldsymbol{z'}_{i,-u})$ $nd$ times, which accounts for the $n(d+2)$ function evaluations. Although this result is interesting, the estimation still requires a number of function evaluations that grows linearly with respect to $d$.

Alternatively, [29] proposed a more efficient technique to evaluate all first-order indices, and only requires $2n$ function evaluations. This alternative relies on the construction of two replicated point sets. The notion of replicated point sets was first introduced by McKay through his replicated Latin Hypercubes in [30]. In order to apply this definition to other types of points, in [31] we generalized this notion to the following.

**Definition 5.1.** *Let $\mathcal{P} = \{\boldsymbol{z}_i\}_{i=0}^{n-1}$ and $\mathcal{P}' = \{\boldsymbol{z'}_i\}_{i=0}^{n-1}$ be any two point sets in $[0,1]^d$. For $u \subset \mathcal{D}$, let $\mathcal{P}^u = \{\boldsymbol{z}_{i,u}\}_{i=0}^{n-1}$ and $\mathcal{P}'^u = \{\boldsymbol{z'}_{i,u}\}_{i=0}^{n-1}$ denote the subset of dimensions of $\mathcal{P}$ and $\mathcal{P}'$ indexed by $u$. We say that $\mathcal{P}$ and $\mathcal{P}'$ are two replicated point sets of order $a \in \{1, \dots, d-1\}$ if for any $u \subset \mathcal{D}$ such that $|u| = a$, $\mathcal{P}^u$ and $\mathcal{P}'^u$ are the same point set in $[0,1]^a$. We denote by $\pi_u$ the permutation that rearranges the order of points in $\mathcal{P}'^u$ into $\mathcal{P}^u$.*

The method introduced in [29] allows to estimate all first-order Sobol' indices with only two replicated point sets of order 1. The key point of this method is to use the permutations $\pi_u$ to replace the hybrid points in equation (5.5).

More precisely, let $\mathcal{P} = \{\boldsymbol{z}_i\}_{i=0}^{n-1}$ and $\mathcal{P}' = \{\boldsymbol{z'}_i\}_{i=0}^{n-1}$ be two replicated point sets of order 1. Define $\{y_i\}_{i=0}^{n-1} := \{f(\boldsymbol{z}_i)\}_{i=0}^{n-1}$ and $\{y'_i\}_{i=0}^{n-1} := \{f(\boldsymbol{z'}_i)\}_{i=0}^{n-1}$ the two sets of function evaluations obtained with $\mathcal{P}$ and $\mathcal{P}'$. From Definition 5.1, we know that $\boldsymbol{z'}_{\pi_u(i),u} = \boldsymbol{z}_{i,u}$. Therefore,

$$\begin{aligned} y'_{\pi_u(i)} &= f(\boldsymbol{z'}_{\pi_u(i),u} : \boldsymbol{z'}_{\pi_u(i),-u}) \\ &= f(\boldsymbol{z}_{i,u} : \boldsymbol{z'}_{\pi_u(i),-u}). \end{aligned}$$

Thus, each $\widehat{\underline{S}}_u$ can be estimated with (5.5) replacing $f(\boldsymbol{z}_{i,u} : \boldsymbol{z'}_{i,-u})$ by $y'_{\pi_u(i)}$, without requiring further function evaluations for each $u$. This estimation method has been studied deeply and generalized in Tissot et al. [32] to the case of closed second-order

indices for orthogonal arrays. In the following we will refer to this method as the replication procedure.

### 5.3 New Estimator For Sobol' Indices

In this section we will construct two different algorithms, `cubSobol_SI_all_g` as Variant A, and `cubSobol_SI_fo_g` as Variant B, to estimate the Sobol' indices. They can be found in Appendix A and Appendix B respectively.

First we need to define a new estimator, different than (5.5) and (5.6), that allows us extend the definition of the error bound (3.9) to Sobol' indices. We consider the two integral formulas (5.3) and (5.4),

$$\begin{aligned}
\underline{S}_u(\boldsymbol{I}(f)) &= \frac{\int_{[0,1)^{2d}} \left(f(\boldsymbol{x}) - f(\boldsymbol{x}_u : \boldsymbol{x'}_{-u})\right) f(\boldsymbol{x'}) \mathrm{d}\boldsymbol{x} \mathrm{d}\boldsymbol{x'}}{\int_{[0,1)^d} f(\boldsymbol{x})^2 \mathrm{d}\boldsymbol{x} - \left(\int_{[0,1)^d} f(\boldsymbol{x}) \mathrm{d}\boldsymbol{x}\right)^2}, \\
\underline{S}_u(\boldsymbol{I}(f)) &= \frac{I_1(f)}{I_3(f) - (I_4(f))^2}, \\
\overline{S}_u(\boldsymbol{I}(f)) &= \frac{\frac{1}{2}\int_{[0,1)^{d+1}} (f(\boldsymbol{x'}) - f(\boldsymbol{x}_u : \boldsymbol{x'}_{-u}))^2 \mathrm{d}\boldsymbol{x}_u \mathrm{d}\boldsymbol{x'}}{\int_{[0,1)^d} f(\boldsymbol{x})^2 \mathrm{d}\boldsymbol{x} - \left(\int_{[0,1)^d} f(\boldsymbol{x}) \mathrm{d}\boldsymbol{x}\right)^2}, \\
\overline{S}_u(\boldsymbol{I}(f)) &= \frac{I_2(f)}{I_3(f) - (I_4(f))^2},
\end{aligned}$$

where $\boldsymbol{I}(f) = (I_1(f), I_2(f), I_3(f), I_4(f))$ is a vector of integral values. Indices $\underline{S}_u(\boldsymbol{I}(f))$ and $\overline{S}_u(\boldsymbol{I}(f))$ are defined as functions over the vector $\boldsymbol{I}(f)$. If we estimate $\boldsymbol{I}(f)$ by $\widehat{\boldsymbol{I}}(f)$ with vector of error bounds $\varepsilon_{\widehat{\boldsymbol{I}}}(f) = (\varepsilon_{\widehat{I}_1}(f), \varepsilon_{\widehat{I}_2}(f), \varepsilon_{\widehat{I}_3}(f), \varepsilon_{\widehat{I}_4}(f))$, according to (3.9) we know that $\boldsymbol{I}(f) \in B_{\varepsilon_{\widehat{\boldsymbol{I}}}(f)}(\widehat{\boldsymbol{I}}(f)) = [\widehat{\boldsymbol{I}}(f) - \varepsilon_{\widehat{\boldsymbol{I}}}(f), \widehat{\boldsymbol{I}}(f) + \varepsilon_{\widehat{\boldsymbol{I}}}(f)]$. For simplicity, we drop the dependence on $f$ for the remaining of the chapter.

Therefore, as an alternative to the common biased Sobol' indices estimators (5.5) and (5.4), we can define the following new estimators with their respective

absolute error bounds:

$$\begin{aligned}
\widehat{\underline{S}}_u &= \frac{1}{2}\left(\min\left(\max_{\boldsymbol{I}\in B_{\varepsilon_{\widehat{\boldsymbol{I}}}}(\widehat{\boldsymbol{I}})} \underline{S}_u(\boldsymbol{I}), 1\right) + \max\left(\min_{\boldsymbol{I}\in B_{\varepsilon_{\widehat{\boldsymbol{I}}}}(\widehat{\boldsymbol{I}})} \underline{S}_u(\boldsymbol{I}), 0\right)\right) \\
\varepsilon_{\widehat{\underline{S}}_u} &= \frac{1}{2}\left(\min\left(\max_{\boldsymbol{I}\in B_{\varepsilon_{\widehat{\boldsymbol{I}}}}(\widehat{\boldsymbol{I}})} \underline{S}_u(\boldsymbol{I}), 1\right) - \max\left(\min_{\boldsymbol{I}\in B_{\varepsilon_{\widehat{\boldsymbol{I}}}}(\widehat{\boldsymbol{I}})} \underline{S}_u(\boldsymbol{I}), 0\right)\right)
\end{aligned} \tag{5.7}$$

and,

$$\begin{aligned}
\widehat{\overline{S}}_u &= \frac{1}{2}\left(\min\left(\max_{\boldsymbol{I}\in B_{\varepsilon_{\widehat{\boldsymbol{I}}}}(\widehat{\boldsymbol{I}})} \overline{S}_u(\boldsymbol{I}), 1\right) + \max\left(\min_{\boldsymbol{I}\in B_{\varepsilon_{\widehat{\boldsymbol{I}}}}(\widehat{\boldsymbol{I}})} \overline{S}_u(\boldsymbol{I}), 0\right)\right) \\
\varepsilon_{\widehat{\overline{S}}_u} &= \frac{1}{2}\left(\min\left(\max_{\boldsymbol{I}\in B_{\varepsilon_{\widehat{\boldsymbol{I}}}}(\widehat{\boldsymbol{I}})} \overline{S}_u(\boldsymbol{I}), 1\right) - \max\left(\min_{\boldsymbol{I}\in B_{\varepsilon_{\widehat{\boldsymbol{I}}}}(\widehat{\boldsymbol{I}})} \overline{S}_u(\boldsymbol{I}), 0\right)\right)
\end{aligned} \tag{5.8}$$

Because numerator and denominator are both known to be positive, maximizing $\underline{S}_u(\boldsymbol{I})$ or $\overline{S}_u(\boldsymbol{I})$ can be done by maximizing the numerators $I_1$ and $I_2$, and minimizing the denominator $I_3 - I_4^2$. Analogously, to minimize $\underline{S}_u(\boldsymbol{I})$ or $\overline{S}_u(\boldsymbol{I})$, one minimizes the numerators $I_1$ and $I_2$, and maximizes the denominator $I_3 - I_4^2$. As an example, Figure 5.1 illustrates the region of possible values of $\underline{S}_2(\boldsymbol{I})$ given the true value of $I_1$ for the test function of Bratley *et al.* [33]. This function is further described in Section 5.4.

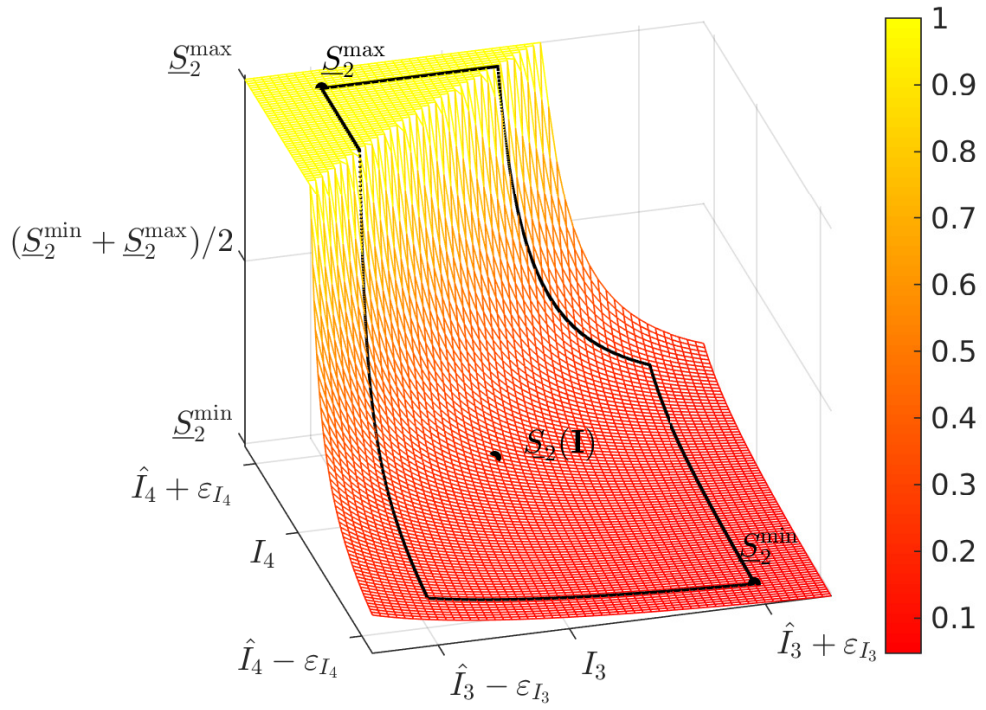


Figure 5.1. The delimited region on the figure represents the values $\underline{S}_2$ in $\boldsymbol{I} \in B_{(0,0,0.05,0.1)}(\widehat{\boldsymbol{I}})$ for the Bratley *et al.* function. There, $\underline{S}_2^{\min} = \max\left(\min_{\boldsymbol{I}\in B_{\varepsilon_{\boldsymbol{I}}}(\widehat{\boldsymbol{I}})} \underline{S}_2(\boldsymbol{I}), 0\right)$ and $\underline{S}_2^{\max} = \min\left(\max_{\boldsymbol{I}\in B_{\varepsilon_{\boldsymbol{I}}}(\widehat{\boldsymbol{I}})} \underline{S}_2(\boldsymbol{I}), 1\right)$, and $\widehat{\underline{S}}_2 = (\underline{S}_2^{\min} + \underline{S}_2^{\max})/2$.

Under the assumption that each integrand in $\underline{S}_u$ and $\overline{S}_u$ lies inside $\mathcal{C}$, these new estimators satisfy:

$$\underline{S}_u \in \left[\widehat{\underline{S}}_u - \varepsilon_{\widehat{\underline{S}}_u}, \widehat{\underline{S}}_u + \varepsilon_{\widehat{\underline{S}}_u}\right], \qquad \overline{S}_u \in \left[\widehat{\overline{S}}_u - \varepsilon_{\widehat{\overline{S}}_u}, \widehat{\overline{S}}_u + \varepsilon_{\widehat{\overline{S}}_u}\right].$$

Thus, like in `cubSobol_g`, `cubSobol_SI_all_g` and `cubSobol_SI_fo_g` increase the value of $m$ until $\varepsilon_{\widehat{\underline{S}}_u} \leq \varepsilon_a$ and $\varepsilon_{\widehat{\overline{S}}_u} \leq \varepsilon_a$ for all $u$. More specifically, at each $m$ the new extended cubatures will loop over the indices $u$ and verify if the error condition is satisfied. For those indices where the error bound is not small enough, it will increase $m$.

Algorithm `cubSobol_SI_all_g`, or Variant A, is designed according to Saltelli's strategy. In addition, we will discuss a possible improvement suggested by Owen [34] to estimate small first-order Sobol' indices. We will distinguish between Variant A.a not including the improvement, and Variant A.b including it. The other Algorithm `cubSobol_SI_fo_g`, or Variant B, uses the replication procedure. For the replication procedure, we apply the results from [31] to construct two replicated digital sequences. Furthermore, these replicated sequences can be scrambled as long as both sequences share the same scrambling dimensionwise. By construction of the replication procedure, Owen's estimator for small indices cannot be applied. Therefore, there is no Variant B.b.

**Algorithm 4.** . Variants A and B

Set: $m \leftarrow \ell_* + r$

$bool \leftarrow false$

**while** !$bool$ **do**

Construct or extend point sets $\mathcal{P}_m$ and $\mathcal{P}'_m$

**for** $u = 1, \ldots, d$ **do**

**if** Variant A **then**

**if** !$bool_u$ **then**

estimate $\underline{\widehat{S}}_u$ and $\overline{\widehat{S}}_u$ with (5.7) and (5.8), and Saltelli's strategy

$bool_u \leftarrow \varepsilon_{\underline{\widehat{S}}_u} \le \varepsilon_a$ & $\varepsilon_{\overline{\widehat{S}}_u} \le \varepsilon_a$

$m_u \leftarrow m$

**end if**

**end if**

**if** Variant B **then**

**if** !$bool_u$ **then**

estimate $\underline{\widehat{S}}_u$ with formula (5.7) and the replication procedure

$bool_u \leftarrow \varepsilon_{\underline{\widehat{S}}_u} \le \varepsilon_a$

$m_u \leftarrow m$

**end if**

**end if**

**end for**

$bool \leftarrow \forall u : \ bool_u$

$m \leftarrow m + 1$

**end while**

return the Sobol' estimates.

The cost of our algorithm in terms of function evaluations depends on whether Variant A or Variant B is selected. To discuss this cost we note by $m^\star$ the ending iteration. If Variant A is selected, the cost of our algorithm is:

$$\sum_{u \in \mathcal{D}} b^{m_u} + 2 \times b^{m^\star}, \qquad m^\star = \max_{u \in \mathcal{D}} \ m_u,$$

where:

- $b^{m_u}$ is the number of evaluations $f\left(\boldsymbol{z}_{i,u} : \boldsymbol{z}'_{i,-u}\right)$ used to estimate both the first-order index $\underline{S}_u$ and the total effect index $\overline{S}_u$,
- $2 \times b^{m^\star}$ is the number of evaluations $f\left(\boldsymbol{z}_i\right)$ and $f\left(\boldsymbol{z}'_i\right)$ used in the estimation of each first-order and total effect indices.

If all $m_u$ are equal, the cost of Variant A becomes $b^{m^\star}(d+2)$ and we recover the cost specified by Saltelli with $n = b^{m^\star}$.

If we chose Variant B, the cost of our algorithm equals $2 \times b^{m^\star}$. This cost corresponds to the one of the replication procedure where $2n = 2 \times b^{m^\star}$, independent of $d$.

**5.3.1 Improvement.** When the value of $\underline{S}_u$ is small, it usually becomes harder to estimate. For this reason, we consider the use of a new estimator for $\underline{\tau}_u^2(f)$ in (5.5) to evaluate small first-order Sobol' indices in Variant A. We will distinguish Variant A.a from Variant A.b, being the latter the one that considers the new estimator. This estimator called "Correlation 2" has been introduced by Owen in [34]. In his article, he discussed and highlighted the efficiency of "Correlation 2" when estimating small first-order indices. Our aim is to show how the use of "Correlation 2" may reduce the total number of function evaluations in Variant A.b compared to Variant A.a. This new estimator is,

$$\widehat{\underline{\tau}_u^2} = \frac{1}{n}\sum_{i=0}^{n-1}(f(\boldsymbol{z}_i) - f(\boldsymbol{t}_{i,u} : \boldsymbol{z}_{i,-u}))(f(\boldsymbol{z}_{i,u} : \boldsymbol{z'}_{i,-u}) - f(\boldsymbol{z'}_i)).$$

Indeed, the new estimator requires an additional set of $n$ function evaluations to estimate $\underline{\tau}_u^2$. However, we expect to increase the precision of our estimate sufficiently enough to see an improvement in our cost.

Below, we discuss the potential improvement brought by using "Correlation 2". For that, assume that the number of small first-order indices is known and equals to $\gamma$. We note by $u_1, \ldots, u_\gamma$ the indices of the corresponding inputs and $\Gamma = \{1, \ldots, \gamma\}$. The cost of Variant A.b is,

$$\sum_{j\in\Gamma} b^{m''_{u_j}} + \sum_{j\in\Gamma} b^{m'_{u_j}} + \sum_{j\in\mathcal{D}\setminus\Gamma} b^{m_{u_j}} + 2\times b^{m^\star}, \tag{5.9}$$

where,

- if $j \in \Gamma$, then $b^{m''_{u_j}}$ is the number of evaluations $f\left(\boldsymbol{t}_{i,u_j} : \boldsymbol{z}_{i,-u_j}\right)$ to estimate $\underline{S}_{u_j}$,
- if $j \in \Gamma$, then $b^{m'_{u_j}}$ is the number of evaluations $f\left(\boldsymbol{z}_{i,u_j} : \boldsymbol{z'}_{i,-u_j}\right)$ to estimate both $\underline{S}_{u_j}$ and $\overline{S}_{u_j}$,

- if $j \in \mathcal{D} \setminus \Gamma$, then $b^{m_{u_j}}$ is the number of evaluations $f\left(\boldsymbol{z}_{i,u_j} : \boldsymbol{z'}_{i,-u_j}\right)$ to estimate both $\underline{S}_{u_j}$ and $\overline{S}_{u_j}$,
- $2 \times b^{m^\star}$ is the number of evaluations $f\left(\boldsymbol{z}_i\right)$ and $f\left(\boldsymbol{z'}_i\right)$ used in the estimation of each first-order and total effect index.

We recall the cost of Variant A.a is,

$$\sum_{j \in \Gamma} b^{m_{u_j}} + \sum_{j \in \mathcal{D} \setminus \Gamma} b^{m_{u_j}} + 2 \times b^{m^\star}. \tag{5.10}$$

The difference (5.9) − (5.10) equals,

$$\sum_{j \in \Gamma} b^{m_{u_j}} \left( b^{m''_{u_j} - m_{u_j}} + b^{m'_{u_j} - m_{u_j}} - 1 \right) = \sum_{j \in \Gamma} c_j. \tag{5.11}$$

Hence, the sign of this difference indicates whether or not using "Correlation 2" brings an improvement. We distinguish two cases :

1) for $j \in \Gamma$, the first case is if the total effect index $\overline{S}_{u_j}$ requires as much or more evaluations than the first-order index $\underline{S}_{u_j}$. Since the total effect estimator requires the function values used for the small indices, we have $m'_{u_j} = m_{u_j}$ and $c_j > 0$.
2) for $j \in \Gamma$, the second case is if the total effect index $\overline{S}_{u_j}$ requires less evaluations than the first-order index $\underline{S}_{u_j}$. In this case, if both $m''_{u_j} < m_{u_j}$ and $m'_{u_j} < m_{u_j}$ then $c_j \leq 0$.

Overall we expect to observe case 2) more often than case 1). The main reason it that the numerator of $\underline{S}_u$ requires to estimate $2d$ dimensional integrals against only $d+1$ dimensional integrals for the numerator of $\overline{S}_u$. Furthermore, in case 2), we expect

the two conditions $m''_{u_j} < m_{u_j}$ and $m'_{u_j} < m_{u_j}$ to usually hold because "Correlation 2" is shown to perform better for small first-order indices, [34].

In practice, one does not know which are the small Sobol' indices $(u_1, \ldots, u_\gamma)$. To overcome this issue, Variant A.b proceeds as follows. If at the end of the first iteration $\widehat{\underline{S}}_u$ is smaller than a threshold (in our case 0.1, as suggested by Owen), then the estimator (5.5) is switched to "Correlation 2" for this particular $u$.

### 5.4 Applications

We test our `cubSobol_SI_all_g` and `cubSobol_SI_fo_g` algorithms with two classical test functions and one financial example. In each case, Sobol' indices are estimated with the following three variants:

- Variant A.a as Variant A without using "Correlation 2".
- Variant A.b as Variant A using "Correlation 2".
- Variant B.

The threshold to decide whether or not a first-order index is small is set to 0.1.

For the two test functions, results of the three variants are compared based on the true estimation errors. These errors correspond to the absolute difference between the true values and their estimates:

$$\delta_{\underline{S}_u} = \left|\underline{S}_u - \widehat{\underline{S}_u}\right|, \qquad \delta_{\overline{S}_u} = \left|\overline{S}_u - \widehat{\overline{S}_u}\right|.$$

We also compare the total number of evaluations of each variant. Results are averaged over 100 repetitions using Owen's scrambled Sobol' sequences [35, 23]. For each repetition, we fix the tolerance $\varepsilon = 5.10^{-3}$, set $\ell_* = 5$, and $r = 4$. The inflation factor (3.10) is set to $\mathfrak{C}(m) = 10 \times 2^{-m}$.

The failure rates are the proportion of the repetitions in which the true estimation errors failed to satisfy $\delta_{\underline{S}_u} \leq \varepsilon_a$ and $\delta_{\underline{S}_u} \leq \varepsilon_a$. When $\delta_{\underline{S}_u} > \varepsilon_a$ or $\delta_{\underline{S}_u} > \varepsilon_a$, we can infer that our algorithm parameters that define the cone are not conservative enough and some integrands fall outside the cone (3.6).

**5.4.1 Classical test functions.** The two classical test functions considered in this article are the g-function introduced by Sobol' [1], and the function introduced by Bratley *et al.* [33]. The idea is to test our method over two categories of functions, additive (Bratley *et al.* function) and multiplicative (g-function).

**Sobol' g-function**. The g-function is defined as follows:

$$f(\boldsymbol{x}) = \prod_{j=1}^{d} g_j(x_j), \qquad g_j(x_j) = \frac{|4x_j - 2| + a_j}{1 + a_j},\ a_j \geq 0.$$

Each value $a_j$ determines the relative importance of the $x_j$. When the value of $a_j$ gets closer to zero the variable $x_j$ becomes more influent. For this example, we chose $d = 6$ and $a_1 = 0, a_2 = 0.5, a_3 = 3, a_4 = 9, a_5 = 99$, and $a_6 = 99$.

Table 5.1 shows the averaged estimation errors obtained with Variant A.a as well as the averaged total number of evaluations performed. Table 5.2 shows the same results obtained with Variant A.b.

Table 5.1. Averaged estimation errors $\delta_{\underline{S}_u}$ , $\delta_{\overline{S}_u}$ and total number of evaluations for Variant A.a.

| input | $\underline{S}_u$ | $\delta_{\underline{S}_u}$ | Failure rates | $\overline{S}_u$ | $\delta_{\overline{S}_u}$ | Failure Rates |
|---|---|---|---|---|---|---|
| $x_1$ | 0.5868 | 0.0004 | 2% | 0.6901 | 0.0005 | 2% |
| $x_2$ | 0.2608 | 0.0006 | 2% | 0.3562 | 0.0004 | 0% |
| $x_3$ | 0.0367 | 0.0011 | 3% | 0.0563 | 0.0008 | 0% |
| $x_4$ | 0.0058 | 0.0012 | 2% | 0.0092 | 0.0003 | 0% |
| $x_5$ | $5.10^{-5}$ | 0.0003 | 0% | $9.10^{-5}$ | $< 10^{-4}$ | 0% |
| $x_6$ | $5.10^{-5}$ | 0.0003 | 0% | $9.10^{-5}$ | $< 10^{-4}$ | 0% |
| Total number of evaluations: 63 088 | | | | | | |

Table 5.2. Averaged estimation errors $\delta_{\underline{S}_u}$ , $\delta_{\overline{S}_u}$ and total number of evaluations for Variant A.b.

| input | $\underline{S}_u$ | $\delta_{\underline{S}_u}$ | Failure rates | $\overline{S}_u$ | $\delta_{\overline{S}_u}$ | Failure rates |
|---|---|---|---|---|---|---|
| $x_1$ | 0.5868 | 0.0005 | 3% | 0.6901 | 0.0006 | 2% |
| $x_2$ | 0.2608 | 0.0007 | 3% | 0.3562 | 0.0003 | 0% |
| $x_3$ | 0.0367 | 0.0015 | 5% | 0.0563 | 0.0009 | 0% |
| $x_4$ | 0.0058 | 0.0003 | 0% | 0.0092 | 0.0003 | 0% |
| $x_5$ | $5.10^{-5}$ | $< 10^{-4}$ | 0% | $9.10^{-5}$ | $< 10^{-4}$ | 0% |
| $x_6$ | $5.10^{-5}$ | $< 10^{-4}$ | 0% | $9.10^{-5}$ | $< 10^{-4}$ | 0% |
| Total number of evaluations: 62 703 | | | | | | |

The main observation is that both approaches give similar results in terms of estimation errors, failure rates, and total number of evaluations. The use of "Correlation 2" in Variant A.b to estimate the four small first-order Sobol' indices $\underline{S}_3, \underline{S}_4, \underline{S}_5, \underline{S}_6$ does not seem to bring much improvement.

Table 5.3 shows averaged estimation errors $\delta_{\underline{S}_u}$ and total number of evaluations for Variant B.

Table 5.3. Averaged estimation errors $\delta_{\underline{S}_u}$ and total number of evaluations for Variant B.

| input | $\underline{S}_u$ | $\delta_{\underline{S}_u}$ | Failure rates |
|---|---|---|---|
| $x_1$ | 0.5868 | 0.0011 | 7% |
| $x_2$ | 0.2608 | 0.0013 | 8% |
| $x_3$ | 0.0367 | 0.0018 | 9% |
| $x_4$ | 0.0058 | 0.0013 | 10% |
| $x_5$ | $5.10^{-5}$ | 0.0031 | 4% |
| $x_6$ | $5.10^{-5}$ | 0.0036 | 5% |
| Total number of evaluations: 32 768 | | | |

The results show that the use of Variant B leads to slightly higher estimation errors and higher failure rates than those obtained with Variant A.a or Variant A.b. However, the total number of evaluations is twice as small. As such, this approach remains interesting when one only wants to estimate first-order indices.

**Bratley et al. function**. In this second example, we consider the Bratley *et al.* function defined by,

$$f(x_1, \ldots, x_d) = \sum_{i=1}^{d} (-1)^i \prod_{j=1}^{i} x_j \ .$$

The importance of each variable $x_j$ depends on their own rank. More explicitly, if $i < j$, then $x_i$ is more influent than $x_j$.

As for the g-function, Tables 5.4 and 5.5 show averaged estimation errors and total number of evaluations for Variant A.a and Variant A.b.

Table 5.4. Averaged estimation errors $\delta_{\underline{S}_u}$, $\delta_{\overline{S}_u}$ and total number of evaluations for Variant A.a.

| input | $\underline{S}_u$ | $\delta_{\underline{S}_u}$ | Failure rates | $\overline{S}_u$ | $\delta_{\overline{S}_u}$ | Failure rates |
|---|---|---|---|---|---|---|
| $x_1$ | 0.6529 | 0.0004 | 0% | 0.7396 | 0.0003 | 0% |
| $x_2$ | 0.1791 | 0.0006 | 0% | 0.2659 | 0.0004 | 0% |
| $x_3$ | 0.0370 | 0.0058 | 71% | 0.0764 | 0.0012 | 2% |
| $x_4$ | 0.0133 | 0.0022 | 12% | 0.0343 | 0.0020 | 7% |
| $x_5$ | 0.0015 | 0.0024 | 11% | 0.0062 | 0.0014 | 0% |
| $x_6$ | 0.0015 | 0.0021 | 6% | 0.0062 | 0.0010 | 3% |
| Total number of evaluations: 70 129 | | | | | | |

Table 5.5. Averaged estimation errors $\delta_{\underline{S}_u}$, $\delta_{\overline{S}_u}$ and total number of evaluations for Variant A.b.

| input | $\underline{S}_u$ | $\delta_{\underline{S}_u}$ | Failure rates | $\overline{S}_u$ | $\delta_{\overline{S}_u}$ | Failure rates |
|---|---|---|---|---|---|---|
| $x_1$ | 0.6529 | 0.0004 | 0% | 0.7396 | 0.0003 | 0% |
| $x_2$ | 0.1791 | 0.0006 | 0% | 0.2659 | 0.0005 | 0% |
| $x_3$ | 0.0370 | 0.0019 | 4% | 0.0764 | 0.0013 | 1% |
| $x_4$ | 0.0133 | 0.0016 | 1% | 0.0343 | 0.0017 | 2% |
| $x_5$ | 0.0015 | 0.0003 | 0% | 0.0062 | 0.0015 | 4% |
| $x_6$ | 0.0015 | 0.0004 | 0% | 0.0062 | 0.0011 | 0% |
| Total number of evaluations: 68 045 | | | | | | |

Variant A.b gives lower estimation errors and fewer failure rates than Variant A.a on the four small first-indices $\underline{S}_3$, $\underline{S}_4, \underline{S}_5$, $\underline{S}_6$ which highlights the performance of "Correlation 2". The discrepancy is particularly notable for input $x_3$ where Variant A.a reaches a failure rate of 71% against only 4% for Variant A.b. In addition, Variant A.b requires less evaluations than Variant A.a.

Table 5.6 shows averaged estimation errors $\delta_{\underline{S}_u}$ and total number of evaluations for Variant B.

Table 5.6. Averaged estimation errors $\delta_{\underline{S}_u}$ and total number of evaluations for Variant B.

| input | $\underline{S}_u$ | $\delta_{\underline{S}_u}$ | Failure rates |
|---|---|---|---|
| $x_1$ | 0.6529 | 0.0003 | 0% |
| $x_2$ | 0.1791 | 0.0006 | 0% |
| $x_3$ | 0.0370 | 0.0010 | 0% |
| $x_4$ | 0.0133 | 0.0038 | 25% |
| $x_5$ | 0.0015 | 0.0024 | 0% |
| $x_6$ | 0.0015 | 0.0026 | 0% |
| Total number of evaluations: 65 536 | | | |

The estimation errors are lower than those of Variant A.a but higher than those of Variant A.b. Failure rates are similar to those of Variant A.b. Since the total number of evaluations is close to the one of Variant A.b, the conclusion is that Variant B does not bring much improvement for this example.

From the results on these two test functions, the main conclusion is that Variant A.b performs the best. This highlights the efficiency of "Correlation 2" to estimate small first-order indices with quasi-Monte Carlo methods. Even if "Correlation 2" originally requires more function evaluations, this drawback is completely overridden when this estimator is included into an automatic algorithm such as `cubSobol_SI_all_g`. For the case where we are only interested in estimating first-order indices, Variant B should be the best choice as illustrated with the g-function.

**5.4.2 Financial case.** As a real case example, we study the payoff function of an arithmetic mean Asian call option. For a discretized Brownian motion $\boldsymbol{B} = (B_{t_1}, B_{t_2}, \ldots, B_{t_d})$ at times $t_i = T \times i/d$, the payoff of the option is:

$$f_{\text{payoff}}(\boldsymbol{B}) = \mathrm{e}^{-rT} \max\left(\frac{1}{d}\sum_{i=1}^{d} S_0 \mathrm{e}^{(r-\sigma^2/2)t_i+\sigma B_{t_i}} - K, 0\right)$$

where $T = 6/52$ (6 weeks) is the maturity of the option, $S_0 = 36$ is the initial stock price, $\sigma = 50\%$ the volatility of the stock, $r = 6\%$ the interest rate, and $K = 40$ the strike price.

The discretized Brownian motion $\boldsymbol{B}$ follows a multivariate Gaussian distribution with mean $\mathbf{0}$ and covariance matrix $\Sigma$, where $\Sigma_{ij} = \min(t_i, t_j)$. If $\boldsymbol{Y} \sim N(\mathbf{0}, I_d)$, then for any matrix $M$, $M\boldsymbol{Y} \sim N(\mathbf{0}, \Sigma)$ where $\Sigma = MM^t$. Hence, we can generate the discretized Brownian motion using the inverse standard Gaussian distribution function $\Phi^{-1}$: $\boldsymbol{B} \sim M\Phi^{-1}(\boldsymbol{X})$ with $\Phi^{-1}(\boldsymbol{X}) = (\Phi^{-1}(X_1), \ldots, \Phi^{-1}(X_d))^t$ and $\boldsymbol{X} \sim \mathcal{U}[0,1]^d$. Thus, we can estimate the Sobol' indices over $f_{\text{payoff}}(M\Phi^{-1}(\boldsymbol{x}))$. In this particular application, the choice of $M$ is important to reduce the effective dimensionality of the function. In this regard, Sobol' indices can help us assess whether we reduce the effective dimensionality. We will compare the case with either the Cholesky decomposition or the PCA construction of $M$.

For dimension $d = 6$, we propose to analyze the results with averaged Sobol' estimates, averaged number of evaluations, and averaged total number of evaluations over 100 repetitions. The absolute tolerance $\varepsilon_a$ and parameters $\ell^*$, $r$ and $\mathfrak{C}(m)$ are set as previously in Section 5.4.1.

Tables 5.7, 5.8, and 5.9 show the results when $M$ corresponds to the Cholesky decomposition of $\Sigma$.

Table 5.7. Averaged values of $\widehat{\underline{S}}_u$ and $\widehat{\overline{S}}_u$, averaged number of evaluations $n$, and total number of evaluations of Variant A.a with the Cholesky decomposition.

| input | $\widehat{\underline{S}}_u$ | $n$ | $\widehat{\overline{S}}_u$ | $n$ |
|---|---|---|---|---|
| $x_1$ | 0.2120 | 131 072 | 0.5650 | 133 693 |
| $x_2$ | 0.1370 | 65 536 | 0.4420 | 131 072 |
| $x_3$ | 0.0820 | 37 356 | 0.3150 | 65 536 |
| $x_4$ | 0.0408 | 16 742 | 0.1950 | 33 096 |
| $x_5$ | 0.0142 | 5 980 | 0.0944 | 16 384 |
| $x_6$ | 0.0048 | 1 106 | 0.0257 | 1 654 |
| Total number of evaluations: 648 821 | | | | |

Table 5.8. Averaged values of $\widehat{\underline{S}}_u$ and $\widehat{\overline{S}}_u$, averaged number of evaluations $n$, and total number of evaluations of Variant A.b with the Cholesky decomposition.

| input | $\widehat{\underline{S}}_u$ | $n$ | $\widehat{\overline{S}}_u$ | $n$ |
|---|---|---|---|---|
| $x_1$ | 0.2120 | 131 072 | 0.5650 | 133 693 |
| $x_2$ | 0.1370 | 65 536 | 0.4420 | 131 072 |
| $x_3$ | 0.0822 | 34 406 | 0.3150 | 65 536 |
| $x_4$ | 0.0434 | 13 107 | 0.1950 | 33 423 |
| $x_5$ | 0.0169 | 2 371 | 0.0944 | 16 384 |
| $x_6$ | 0.0049 | 532 | 0.0259 | 1 577 |
| Total number of evaluations: 699 487 | | | | |

Table 5.9. Averaged values of $\widehat{\underline{S}}_u$, averaged number of evaluations $n$, and total number of evaluations of Variant B with the Cholesky decomposition.

| input | $\widehat{\underline{S}}_u$ | $n$ |
|---|---|---|
| $x_1$ | 0.2120 | 131 072 |
| $x_2$ | 0.1350 | 129 766 |
| $x_3$ | 0.0774 | 62 290 |
| $x_4$ | 0.0338 | 49 961 |
| $x_5$ | 0.0149 | 48 845 |
| $x_6$ | 0.0028 | 17 905 |
| Total number of evaluations: 262 144 | | |

It turns out that Variant A.b does not perform better than Variant A.a. Averaged Sobol' estimates are similar but the average total number of function evaluations is slightly worse. The explanation is that total effect Sobol' indices require more evaluations than their corresponding first-order indices. As such, "Correlation 2" falls into case 1) explained in Section 5.3.1. Variant B shows a drastic improvement on the total number of evaluations with the drawback of probably a not accurate estimation for input $x_4$ (the result seems to be worse than the other two Variant results).

Tables 5.10, 5.11, and 5.12 show the results when $M$ corresponds to the PCA decomposition of $\Sigma$.

Table 5.10. Averaged values of $\widehat{\underline{S}}_u$ and $\widehat{\overline{S}}_u$, averaged number of evaluations $n$, and total number of evaluations of Variant A.a with the PCA decomposition.

| input | $\widehat{\underline{S}}_u$ | $n$ | $\widehat{\overline{S}}_u$ | $n$ |
|---|---|---|---|---|
| $x_1$ | 0.9800 | 131 072 | 0.9960 | 117 965 |
| $x_2$ | 0.0037 | 901 | 0.0172 | 855 |
| $x_3$ | 0.0016 | 512 | 0.0023 | 512 |
| $x_4$ | 0.0014 | 512 | 0.0005 | 512 |
| $x_5$ | 0.0005 | 512 | 0.0001 | 512 |
| $x_6$ | 0.0002 | 512 | 0.0000 | 512 |
| Total number of evaluations: 396 165 | | | | |

Table 5.11. Averaged values of $\widehat{\underline{S}}_u$ and $\widehat{\overline{S}}_u$, averaged number of evaluations $n$, and total number of evaluations of Variant A.b with the PCA decomposition.

| input | $\widehat{\underline{S}}_u$ | $n$ | $\widehat{\overline{S}}_u$ | $n$ |
|---|---|---|---|---|
| $x_1$ | 0.9800 | 131 072 | 0.9960 | 117 309 |
| $x_2$ | 0.0034 | 512 | 0.0172 | 824 |
| $x_3$ | 0.0004 | 512 | 0.0023 | 512 |
| $x_4$ | 0.0001 | 512 | 0.0005 | 512 |
| $x_5$ | 0.0000 | 512 | 0.0001 | 512 |
| $x_6$ | 0.0000 | 512 | 0.0000 | 512 |
| Total number of evaluations: 398 648 | | | | |

Table 5.12. Averaged values of $\widehat{\underline{S}}_u$, averaged number of evaluations $n$, and total number of evaluations of Variant B with the PCA decomposition.

| input | $\widehat{\underline{S}}_u$ | $n$ |
|---|---|---|
| $x_1$ | 0.9800 | 131 072 |
| $x_2$ | 0.0037 | 18 540 |
| $x_3$ | 0.0029 | 10 173 |
| $x_4$ | 0.0029 | 7 813 |
| $x_5$ | 0.0024 | 4 731 |
| $x_6$ | 0.0028 | 9 723 |
| Total number of evaluations: 262 144 | | |

Variant A.a performs as well as Variant A.b in terms of average total number of evaluations. However, Variant A.a is most probably giving worse estimates for the small first-order indices. Table 5.10 shows that averaged $\widehat{\underline{S}}_4$ is greater than $\widehat{\overline{S}}_4$, $\widehat{\underline{S}}_5$ greater than $\widehat{\overline{S}}_5$, and $\widehat{\underline{S}}_6$ greater than $\widehat{\overline{S}}_6$, which is inconsistent with the property $\underline{S}_u \leq \overline{S}_u$. Once again, Variant B shows an improvement on the total number of evaluations with the drawback of slightly overestimating the small first-order indices.

Overall, each variant captures well how the PCA Brownian motion construction reduces the high effective dimensionality of the payoff function.

# CHAPTER 6
# CONCLUSION

## 6.1 Summary

In this thesis, we focused on the estimation of high dimensional integrals that appear in many applications such as finance, particle physics, or imaging. We proposed to overcome the *curse of dimensionality* by using quasi-Monte Carlo methods, a highly stratified Monte Carlo technique. More specifically, we studied in depth the numerical integration properties of digital sequences and rank-1 lattice node sequences since they have a structure that is helpful to analyze the error of our approximation.

The Koksma-Hlawka inequality provides an error bound for the quasi-Monte Carlo approximation. Nonetheless, it requires the estimation of the variation of the integrand $V(f)$ in the sense of Hardy and Krause, which is computationally expensive to approximate. Instead, we used an alternative error bound based on the discrete Fourier coefficients of the integrand that led to the design of a new adaptive automatic algorithm. To achieve this goal, we proved that for some Fourier basis functions, the average evaluated at any digital net and rank-1 lattice node is 0. Hence, the error can be bounded with the coefficients corresponding to the other Fourier basis functions. Under some assumptions about the decay rate of the Fourier coefficients, we were able to define a data-driven error bound. This new methodology guarantees the error bound for a non-convex cone of integrands. The non-convexity property is important since adaptivity is proven not to help for convex classes of functions [36]. The definition of this cone contains parameters that can be tuned although they may be hard to determine.

In Chapter 4, we proposed a simple modification of our algorithms to allow the introduction of a more general tolerance function. This tolerance function replaces

the usual absolute error tolerance condition and introduces the relative error term. Indeed, this modification can be applied to any functional estimation problem with an estimator that has a parametrized known absolute error bound.

Quasi-Monte Carlo methods estimate the integral of low effective dimensional integrands efficiently [37], even if the nominal dimension is high. However, we still lack of a specific measure that quantifies the effective dimensionality of an integrand. To measure this property, we decided to estimate Sobol' indices by extending our automatic `cubSobol_g` cubature, which required a new estimator to preserve the tolerance guarantees. One assumption about estimating these indices is that the cost of evaluating our integrand is expensive. For that reason, we showed that we can apply our method to Saltelli's idea, and to the replication procedure.

### 6.2 Future Work

The new cubatures proposed in Chapter 3 succeeded in estimating efficiently the integrals of our examples. Both the Owen scrambled Sobol' sequences and Dirk Nuyens lattice generator perform well. However, there are possible improvements involving each cubature:

- `cubSobol_g`: The algorithm is designed for any digital net although it only measures the error based on the integrand's Fourier coefficients. Thus, Josef Dick's [38] high-order nets improved efficiency cannot be captured with `cubSobol_g`. We should design a different error bound for these particular sequences.

- `cubLattice_g`: For this algorithm, the choice of the generating vector is crucial. One possible research project can be the study of a possible extensible rank-1 lattice sequence construction based on the integrand properties, for instance by estimating the optimal dimension weights choice. Another interesting question is how Korobov spaces are related to our cone of functions. By finding this

relationship, we could apply all the known results to `cubLattice_g`.

In this thesis we provided an upper bound on the complexity of the problem but we have not yet obtained a lower bound. We would also like to understand how the cone parameters might depend on the dimension of the problem, and we would like to extend our adaptive algorithm to infinite dimensional problems via multi-level [39] or multivariate decomposition methods.

In addition to the projects proposed above, there is a list of other possible research topics:

- Monte Carlo Efficiency Improvements: Importance sampling and control variates are usual Monte Carlo variance reduction techniques. We can study how to apply them to our quasi-Monte Carlo cubatures, since optimal IID Monte Carlo parameters may behave poorly for quasi-Monte Carlo, as shown in [40].

- Reliability: Following another line of work, we would like to investigate how to automatically modify the inflation factor in (3.10), when data-based necessary conditions are violated ($f \notin \mathcal{C}$).

- General Linear Functional Estimator: We would like to design new algorithms that estimate $\mathcal{L}f$ when $\mathcal{L}$ is a more general operator (not only the integral operator $I$).

- Replication Procedure: We showed that scrambled Sobol' sequences can be used to estimate Sobol' indices using the replicated procedure. The same estimators and algorithms can be applied with rank-1 lattices by noticing that rank-1 lattices are also replicated designs of order 1.

- Variation Measure Definition: Sobol' indices measure what part of the variance is explained by a subset of dimensions. However, in quasi-Monte Carlo the error is related to the variation instead of the variance. Thus, the search for a new definition of effective dimension should be explored.

- Particle Physics: We collaborated with the CDF experiment and the Computing Division at Fermilab. For the fourth top mass measurement, the CDF team plans to use quasi-Monte Carlo methods. Although we developed a C++ package of `cubSobol_g` for this measurement, there remains more work on the study of the structure of the integrands estimated.

- High Performance Computing: For high precision requirements, multidimensional integrals can still require a lot of integration points and be deeply time consuming. Therefore, another line of research is studying the code parallelization of the aforementioned algorithms according to [41] and Fast Fourier Transform constructions, for high scale computing.

# APPENDIX A

## MATLAB CODE FOR VARIANT A: CUBSOBOL_SI_ALL_G

```
function [q,int,out_param] = cubSobol_SI_all_g(varargin)
%CUBSOBOL_SI_ALL_G Quasi-Monte Carlo method using Sobol' cubatures
%to compute all first order and total effect Sobol Indices
%within a specified generalized error tolerance with guarantees under
%Walsh-Fourier coefficients cone decay assumptions.
%
%   [q,out_param] = CUBSOBOL_SI_ALL_G(f,hyperbox) estimates all ...
    first order
%   and total effect Sobol Indices of f, where hyperbox is the ...
    sample space
%   of the uniform distribution (the distribution can also be set to ...
    normal),
%   and the estimation error is guaranteed not to be greater than a ...
    specific
%   generalized error tolerance tolfun:=max(abstol,reltol*| SI(f) |).
%   Input f is a function handle. f should accept an n x d matrix input,
%   where d is the dimension and n is the number of points being ...
    evaluated
%   simultaneously. The input hyperbox is a 2 x d matrix, where the ...
    first
%   row corresponds to the lower limits and the second row ...
    corresponds to
%   the upper limits of the integral. Given the construction of Sobol'
%   sequences, d must be a positive integer with 1≤d≤370.
%
%   q = CUBSOBOL_SI_ALL_G(f,hyperbox,measure,abstol,reltol) ...
    estimates all
%   first order and total effect Sobol Indices of f. The answer
%   is given within the generalized error tolerance tolfun. All ...
    parameters
%   should be input in the order specified above. If an input is not ...
    specified,
%   the default value is used. Note that if an input is not specified,
%   the remaining tail cannot be specified either. Inputs f and hyperbox
%   are required. The other optional inputs are in the correct order:
%   measure,abstol,reltol,mmin,mmax,fudge,toltype and
%   theta.
%
%   q = CUBSOBOL_SI_ALL_G(f,hyperbox,'measure',measure,'abstol',
%       abstol,'reltol',reltol)
%   estimates all first order and total effect Sobol Indices of f. ...
    The answer
%   is given within the generalized error tolerance tolfun. All the ...
    field-value
%   pairs are optional and can be supplied in any order. If an input ...
    is not
%   specified, the default value is used.
%
%   q = CUBSOBOL_SI_ALL_G(f,hyperbox,in_param) estimates all first ...
    order and
%   total effect Sobol Indices of f. The answer is given within the
%   generalized error tolerance tolfun.
%
%   Input Arguments
```

```
41 %
42 %     f ——— the integrand whose input should be a matrix n x d where ...
       n is
43 %     the number of data points and d the dimension, which cannot be
44 %     greater than 370. By default f is f=@ x.^2.
45 %
46 %     hyperbox ——— sample space of the distribution that defines the ...
       random
47 %     input vector. It must be be a 2 x d matrix, where the first row
48 %     corresponds to the lower limits and the second row to the upper
49 %     limits. The default value is [0;1].
50 %
51 %     in_param.measure ——— for f(x)*mu(dx), we can define mu(dx) to ...
       be the
52 %     measure of a uniformly distributed random variable in the hyperbox
53 %     (each dimension is independent) or normally distributed
54 %     with covariance matrix I_d. The only possible values are
55 %     'uniform' or 'normal'. For 'uniform', the hyperbox must have
56 %     a finite volume while for 'normal', the hyperbox can only be ...
       defined as
57 %     (-Inf,Inf)^d. By default it is 'uniform'.
58 %
59 %     in_param.abstol ——— the absolute error tolerance, abstol≥0. By
60 %     default it is 1e-4.
61 %
62 %     in_param.reltol ——— the relative error tolerance, which should be
63 %     in [0,1]. Default value is 1e-2.
64 %
65 %   Optional Input Arguments
66 %
67 %     in_param.mmin ——— the minimum number of points to start is 2^mmin.
68 %     The cone condition on the Fourier coefficients decay requires a
69 %     minimum number of points to start. The advice is to consider ...
       at least
70 %     mmin=10. mmin needs to be a positive integer with mmin≤mmax. By
71 %     default it is 10.
72 %
73 %     in_param.mmax ——— the maximum budget is 2^mmax. By ...
       construction of
74 %     the Sobol' generator, mmax is a positive integer such that
75 %     mmin≤mmax≤53. The default value is 24.
76 %
77 %     in_param.fudge ——— the positive function multiplying the finite
78 %     sum of Fast Walsh Fourier coefficients specified in the cone ...
       of functions.
79 %     This input is a function handle. The fudge should accept an ...
       array of
80 %     nonnegative integers being evaluated simultaneously. For more
81 %     technical information about this parameter, refer to the ...
       references.
82 %     By default it is @(m) 5*2.^-m.
83 %
84 %     in_param.toltype ——— this is the generalized tolerance function.
85 %     There are two choices, 'max' which takes
```

```
86 %     max(abstol,reltol*| integral(f) | ) and 'comb' which is the ...
   linear combination
87 %     theta*abstol+(1-theta)*reltol*| integral(f) | . Theta is another
88 %     parameter to be specified with 'comb'(see below). For pure ...
   absolute
89 %     error, either choose 'max' and set reltol = 0 or choose 'comb' ...
   and set
90 %     theta = 1. For pure relative error, either choose 'max' and set
91 %     abstol = 0 or choose 'comb' and set theta = 0. Note that with ...
   'max',
92 %     the user can not input abstol = reltol = 0 and with 'comb', if ...
   theta = 1
93 %     abstol con not be 0 while if theta = 0, reltol can not be 0.
94 %     By default toltype is 'max'.
95 %
96 %     in_param.theta --- this input is parametrizing the toltype
97 %     'comb'. Thus, it is only active when the toltype
98 %     chosen is 'comb'. It establishes the linear combination weight
99 %     between the absolute and relative tolerances
100 %     theta*abstol+(1-theta)*reltol*| integral(f) |. Note that for ...
   theta = 1,
101 %     we have pure absolute tolerance while for theta = 0, we have pure
102 %     relative tolerance. By default, theta=1.
103 %
104 %     in_param.threshold_small --- this input is the size at which ...
   we use
105 %     the small indices estimator instead of the usual. By default ...
   is 0.1.
106 %
107 %   Output Arguments
108 %
109 %     q --- the estimated value of all Sobol Indices in matrix form. ...
   First
110 %     row corresponds to first order indices, and second row to total
111 %     effect indices. Each column is the dimension for which we estimate
112 %     the indice.
113 %
114 %     out_param.d --- dimension of the domain of f.
115 %
116 %     out_param.n --- number of Sobol' points used to compute q.
117 %
118 %     out_param.bound_err --- predicted error bound of q based on ...
   the cone
119 %     conditions. If the function lies in the cone, the real error ...
   will be
120 %     smaller than generalized tolerance.
121 %
122 %     out_param.time --- time elapsed in seconds when calling ...
   cubSobol_SI_all_g.
123 %
124 %     out_param.exitflag --- for each q, this is a binary vector stating
125 %     whether warning flags arise. This is a triple array element with
126 %     flags stored in the first component, and indice specified by the
127 %     other two. These flags tell about which conditions make the
```

```
128 %     final result certainly not guaranteed. One flag is considered ...
      arisen
129 %     when its value is 1. The following list explains the flags in the
130 %     respective vector order:
131 %
132 %                         1 : If reaching overbudget. It states whether
133 %                         the max budget is attained without reaching the
134 %                         guaranteed error tolerance.
135 %
136 %                         2 : If the function lies outside the cone. In
137 %                         this case, results are not guaranteed. For more
138 %                         information about the cone definition, check the
139 %                         article mentioned below.
140 %
141 %     out_param.small ---- Boolean indicating if we changed our estimator
142 %     for small first order Sobol Indices. This improves the ...
      estimation of
143 %     the indices when they are small.
144 %
145 %  Guarantee
146 % This algorithm computes first order and total effect Sobol Indices of
147 % real valued functions in [0,1)^d with a prescribed generalized error
148 % tolerance. The Walsh-Fourier coefficients of the integrand are assumed
149 % to be absolutely convergent. If the algorithm terminates without ...
      warning
150 % messages, the output is given with guarantees under the assumption ...
      that
151 % the integrand lies inside a cone of functions. The guarantee is ...
      based on
152 % the decay rate of the Walsh-Fourier coefficients. For more details ...
      on how
153 % the cone is defined, please refer to the references below.
154 %
155 %  Examples
156 %
157 % Example 1:
158 % Ishigami example:
159 %
160 % >> f = @(x) sin(x(:,1)).*(1+1/10*(x(:,3).^4))+7*sin(x(:,2)).^2; ...
      hyperbox = pi*[-ones(1,3) ; ones(1,3)];
161 % >> q = cubSobol_SI_all_g(f,hyperbox,'uniform',1e-1,0); exactsol = ...
      [.3139051827, .4424111333, 0.; .5575888667, .4424111326, ...
      .2436836832];
162 % >> check = sum(sum(abs(exactsol-q) < ...
      gail.tolfun(1e-1,0,1,exactsol,'max')))
163 % check = 6
164 %
165 %
166 % Example 2:
167 % Bratley example:
168 %
169 % >> f = @(x) sum(bsxfun(@times, cumprod(x,2), (-1).^(1:6)),2); ...
      hyperbox = [-zeros(1,6) ; ones(1,6)];
170 % >> q = cubSobol_SI_all_g(f,hyperbox,'uniform',1e-1,1e-1); exactsol ...
```

```
    = [.6528636616, .1791303924, 0.3701041165e-1, 0.1332374820e-1, ...
    0.1480416466e-2, 0.1480416466e-2; .7396477462, .2659144770, ...
    0.764211693e-1, 0.343115454e-1, 0.62384628e-2, 0.62384628e-2];
% >> check = sum(sum(abs(exactsol-q) < ...
    gail.tolfun(1e-1,1e-1,1,exactsol,'max')))
% check = 12
%
%
% Example 3:
% Sobol' g-function example, first order indice for dimension 5:
%
% >> a = [0 1/2 3 9 99 99]; hyperbox = [zeros(1,6) ; ones(1,6)];
% >> f = @(x) prod(bsxfun(@(x,a) (abs(4*x-2)+a)./(1+a), x , a),2);
% >> q = cubSobol_SI_all_g(f,hyperbox,'uniform',1e-2,1e-1); exactsol ...
    = [.5867811893, .2607916397, 0.3667382378e-1, 0.5867811312e-2, ...
    0.5867753221e-4, 0.5867753221e-4; .6900858920, .3561733634, ...
    0.563335432e-1, 0.91705767e-2, 0.920079e-4, 0.920079e-4];
% >> check = sum(sum(abs(exactsol-q) < ...
    gail.tolfun(1e-2,1e-1,1,exactsol,'max')))
% check = 12
%
%
% Example 4:
% Morokoff and Caflish example, total effect indice for dimension 4:
%
% >> f = @(x) (1+1/6)^6*prod(x,2).^(1/6); hyperbox = [zeros(1,6) ; ...
    ones(1,6)];
% >> q = cubSobol_SI_all_g(f,hyperbox,'uniform',1e-2,1e-1); exactsol ...
    = [.1581948744, .1581948744, .1581948744, .1581948744, ...
    .1581948744, .1581948744; .1753745708, .1753745708, .1753745708, ...
    .1753745708, .1753745708, .1753745708];
% >> check = sum(sum(abs(exactsol-q) < ...
    gail.tolfun(1e-2,1e-1,1,exactsol,'max')))
% check = 12
%
%
%   See also CUBSOBOL_G
%
%  References
%
%   [1] Fred J. Hickernell and Lluis Antoni Jimenez Rugama, ...
    "Reliable adaptive
%   cubature using digital sequences," 2014. Submitted for publication:
%   arXiv:1410.8615.
%
%   [2] Art B. Owen, "Better Estimation of Small Sobol' Sensitivity
%   Indices," ACM Trans. Model. Comput. Simul., 23, 2, Article 11 ...
    (May 2013).
%
%   [3] Sou-Cheng T. Choi, Fred J. Hickernell, Yuhan Ding, Lan Jiang,
%   Lluis Antoni Jimenez Rugama, Xin Tong, Yizhi Zhang and Xuan Zhou,
%   GAIL: Guaranteed Automatic Integration Library (Version 2.1)
%   [MATLAB Software], 2015. Available from ...
    http://code.google.com/p/gail/
```

```
209 %
210 %   [4] Sou-Cheng T. Choi, "MINRES-QLP Pack and Reliable Reproducible
211 %   Research via Supportable Scientific Software," Journal of Open ...
    Research
212 %   Software, Volume 2, Number 1, e22, pp. 1-7, 2014.
213 %
214 %   [5] Sou-Cheng T. Choi and Fred J. Hickernell, "IIT MATH-573 Reliable
215 %   Mathematical Software" [Course Slides], Illinois Institute of
216 %   Technology, Chicago, IL, 2013. Available from
217 %   http://code.google.com/p/gail/
218 %
219 %   [6] Daniel S. Katz, Sou-Cheng T. Choi, Hilmar Lapp, Ketan ...
    Maheshwari,
220 %   Frank Loffler, Matthew Turk, Marcus D. Hanwell, Nancy Wilkins-Diehr,
221 %   James Hetherington, James Howison, Shel Swenson, Gabrielle D. Allen,
222 %   Anne C. Elster, Bruce Berriman, Colin Venters, "Summary of the First
223 %   Workshop On Sustainable Software for Science: Practice And ...
    Experiences
224 %   (WSSSPE1)," Journal of Open Research Software, Volume 2, Number ...
    1, e6,
225 %   pp. 1-21, 2014.
226 %
227 %   If you find GAIL helpful in your work, please support us by ...
    citing the
228 %   above papers, software, and materials.
229 %
230
231 t_start = tic;
232 %% Initial important cone factors and Check-initialize parameters
233 r_lag = 4; %distance between coefficients summed and those computed
234 [f,hyperbox,out_param] = cubSobol_SI_all_g_param(r_lag,varargin{:});
235 out_param.small = false(2, out_param.d); % Boolean that states ...
    whether we are dealing with small indices estimators
236 converged = false(2, out_param.d); % We flag the indices that converged
237 l_star = out_param.mmin - r_lag; % Minimum gathering of points for ...
    the sums of DFWT
238 omg_circ = @(m) 2.^(-m);
239 omg_hat = @(m) ...
    out_param.fudge(m)/((1+out_param.fudge(r_lag))*omg_circ(r_lag));
240
241 if strcmp(out_param.measure,'normal')
242    f=@(x) f(gail.stdnorminv(x));
243 elseif strcmp(out_param.measure,'uniform')
244    Cnorm = prod(hyperbox(2,:)-hyperbox(1,:));
245    f=@(x) Cnorm*f(bsxfun(@plus,hyperbox(1,:), ...
246        bsxfun(@times,(hyperbox(2,:)-hyperbox(1,:)),x))); % a + ...
            (b-a)x = u
247 end
248
249 % First and total order indices estimators and function evaluations
250 % The S estimators use numerator and denominator from left to right ...
    order,
251 % for instance, down(1,2,3,4) = [a, b, c, d] ---> (a-b^2)/ (c-d^2)
252 % Down and up are for the inferior and superior bounds of our integral
```

```
% estimates.

% Below, both numerator and denominator need to be positive. In addition,
% the final value needs to be between 0 and 1.
numerator_size = 1;
% Sfo_min = @(down, up) estimate_min(down, up); % S function for first order
% Sfo_max = @(down, up) estimate_max(down, up); % S function for first order
% ffo1 = @(xpts,u,fx,fy,fxy,fzx) fx.*fxy;
% ffo2 = @(xpts,u,fx,fy,fxy,fzx) 1/2*(fx + fxy);
% Sfo_min = @(down, up) estimate_min([down(1) down(end) down(2:end)], [up(1) up(end) up(2:end)]); % S function for first order
% Sfo_max = @(down, up) estimate_max([down(1) down(end) down(2:end)], [up(1) up(end) up(2:end)]); % S function for first order
% ffo = @(xpts,u,fx,fy,fxy,fzx) fx.*fxy;
Sfo_min = @(down, up) estimate_min([down(1) 0 down(2:end)], [up(1) 0 up(2:end)]); % S function for first order
Sfo_max = @(down, up) estimate_max([down(1) 0 down(2:end)], [up(1) 0 up(2:end)]); % S function for first order
ffo = @(xpts,u,fx,fy,fxy,fzx) fx.*(fxy - fy); % ffo = @(xpts,u,fx,fy,fxy,fzx) (fx - 1/2*mean(fx + fxy)).*(fxy - 1/2*mean(fx + fxy));

Sfo_s_min = @(down,up) estimate_min([down(1) 0 down(2:end)], [up(1) 0 up(2:end)]); % S function for first order small
Sfo_s_max = @(down,up) estimate_max([down(1) 0 down(2:end)], [up(1) 0 up(2:end)]); % S function for first order small
ffo_s = @(xpts,u,fx,fy,fxy,fzx) (fx - fzx).*(fxy - fy); % We redefine the non normalized estimator
    % fzx is only used for the small indices. But we needed as input for
    % all estimators to generalize the algorithm. If we give it as an
    % input, we can only evaluate it one time and use it to several small
    % indices.

Stot_min = Sfo_s_min; % S function for total effect
Stot_max = Sfo_s_max;
ftot = @(xpts,u,fx,fy,fxy,fzx) 1/2*(fy - fxy).^2;


%% Main algorithm (we added 2xd dimensions for each index and and additional 2 for mean of f and f^2)
sobstr = sobolset(3*out_param.d); %generate a Sobol' sequence 3*d to consider the changing to the estimator for some smaller size indices
sobstr = scramble(sobstr,'MatousekAffineOwen'); %scramble it
kappanumap_fx2_fx = bsxfun(@times,(1:2^out_param.mmin)', [1 1]); %initialize map
Stilde_fx2_fx = zeros(out_param.mmax-out_param.mmin+1, 2); %initialize sum of DFWT terms for fx2 and fx
CStilde_low_fx2_fx = -inf(out_param.mmax-l_star+1, 2); %initialize ...
```

```
    various sums of DFWT terms for necessary conditions for fx2 and fx
288 CStilde_up_fx2_fx = inf(out_param.mmax-l_star+1, 2); %initialize ...
    various sums of DFWT terms for necessary conditions for fx2 and fx
289 err_bound_int_fx2_fx = inf(out_param.mmax-out_param.mmin+1, 2); ...
    %initialize error estimates for fx2 and fx
290 est_int_fx2_fx = zeros(out_param.mmax-out_param.mmin+1, 2); % ...
    Estimate of the mean for fx2 and fx
291 exit_len = 2;
292 out_param.exit = false(exit_len, 2, out_param.d); %we start the ...
    algorithm with all warning flags down
293
294 for u = 1:out_param.d
295     for r = 1:2 % r = 1 for first order, r = 2 for total effect
296         if r == 1 % If we use correl 2 first order ind estimator, we ...
                need
297                 % to estimate 2 integrals in the numerator
298             INDICES(r,u).kappanumap = kappanumap_fx2_fx(:, ...
                    1:numerator_size);
299             INDICES(r,u).Stilde = ...
                    zeros(out_param.mmax-out_param.mmin+1, numerator_size);
300             INDICES(r,u).CStilde_low = CStilde_low_fx2_fx(:, ...
                    1:numerator_size);
301             INDICES(r,u).CStilde_up = CStilde_up_fx2_fx(:, ...
                    1:numerator_size);
302             INDICES(r,u).Smin = Sfo_min;
303             INDICES(r,u).Smax = Sfo_max;
304             if numerator_size == 1
305                 INDICES(r,u).f = {ffo}; % {ffo1, ffo2} {ffo}
306             elseif numerator_size == 2
307                 INDICES(r,u).f = {ffo1, ffo2}; % {ffo1, ffo2} {ffo}
308             else
309                 error('Check first order Sobol indices, number of ...
                        functions in numerator')
310             end
311             INDICES(r,u).est_int = est_int_fx2_fx(:, ...
                    1:numerator_size); %initialize mean estimates for ...
                    each integral in the numerator
312             INDICES(r,u).err_bound_int = err_bound_int_fx2_fx(:, ...
                    1:numerator_size); %initialize error estimates for ...
                    each integral in the numerator
313         else
314             INDICES(r,u).kappanumap = kappanumap_fx2_fx(:,1);
315             INDICES(r,u).Stilde = ...
                    zeros(out_param.mmax-out_param.mmin+1,1);
316             INDICES(r,u).CStilde_low = CStilde_low_fx2_fx(:,1);
317             INDICES(r,u).CStilde_up = CStilde_up_fx2_fx(:,1);
318             INDICES(r,u).Smin = Stot_min;
319             INDICES(r,u).Smax = Stot_max;
320             INDICES(r,u).f = {ftot};
321             INDICES(r,u).est_int = est_int_fx2_fx(:,1); %initialize ...
                    mean estimates for each integral in the numerator
322             INDICES(r,u).err_bound_int = err_bound_int_fx2_fx(:,1); ...
                    %initialize error estimates for each integral in the ...
                    numerator
```

```
323         end
324         INDICES(r,u).errest = inf(out_param.mmax-out_param.mmin+1); ...
               %initialize error estimates for each Sobol' indice
325     end
326 end
327
328
329 %% Initial points and FWT
330 out_param.n = 2^out_param.mmin*ones(2, out_param.d); %total number ...
    of points to start with
331 n0 = out_param.n(1,1); %initial number of points
332 xpts = sobstr(1:n0,1:3*out_param.d); %grab Sobol' points
333 fx = f(xpts(:,1:out_param.d)); %evaluate integrands y
334 fx2 = fx.^2; %evaluate integrands y2
335 fxval = fx; % We store fx because fx will later contain the fast ...
    transform
336 fx2val = fx2; % We store fx2 because fx2 will later contain the fast ...
    transform
337 fy = f(xpts(:,out_param.d+1:2*out_param.d)); % We evaluate the f at ...
    the replicated points
338 fzx = []; % We set it empty. So if it is empty, we only evaluate it ...
    1 time.
339
340 % We check which indices are small and evaluate the numerator of all
341 % indices for the first time
342
343 for u = 1:out_param.d
344     fxy = f([xpts(:,out_param.d+1:out_param.d+u-1) xpts(:,u) ...
           xpts(:,out_param.d+u+1:2*out_param.d)]);
345         % fxy is the only one we evaluate as a function of u. This ...
               is the d
346         % from the evaluation cost (d+1)n
347     INDICES(1,u).y = [];
348     for p = 1:numerator_size
349             % y values for integrals in the numerator
350         INDICES(1,u).y = [INDICES(1,u).y ...
               INDICES(1,u).f{p}(xpts,u,fx,fy,fxy,0)];
351     end
352     aux_double = INDICES(1,u).Smin([mean(INDICES(1,u).y, 1) ...
           mean(fx2) mean(fx)],[mean(INDICES(1,u).y, 1) mean(fx2) ...
           mean(fx)]);
353     if aux_double < out_param.threshold_small % If the normalized ...
           first order index is small, we change to a better estimator ...
           estimator
354         if isempty(fzx)
355             fzx = f([xpts(:,1:u-1) xpts(:,2*out_param.d + u) ...
                   xpts(:,u+1:out_param.d)]);
356         end
357         INDICES(1,u).kappanumap = kappanumap_fx2_fx(:,1);
358         INDICES(1,u).Stilde = zeros(out_param.mmax-out_param.mmin+1,1);
359         INDICES(1,u).CStilde_low = CStilde_low_fx2_fx(:,1);
360         INDICES(1,u).CStilde_up = CStilde_up_fx2_fx(:,1);
361         INDICES(1,u).Smin = Sfo_s_min; % We redefine the S function ...
               for the small estimator
```

```
362         INDICES(1,u).Smax = Sfo_s_max;
363         INDICES(1,u).f = {ffo_s};
364         INDICES(1,u).est_int = est_int_fx2_fx(:,1); %initialize mean ...
                estimates for each integral in the numerator
365         INDICES(1,u).err_bound_int = err_bound_int_fx2_fx(:,1); ...
                %initialize error estimates for each integral in the ...
                numerator
366         INDICES(1,u).y = INDICES(1,u).f{1}(xpts,u,fx,fy,fxy,fzx); % ...
                We reevaluate the points if we change the estimator
367         out_param.small(1,u) = 1;
368         fzx = [];
369     end
370     INDICES(2,u).y = INDICES(2,u).f{1}(xpts,u,fx,fy,fxy,0);
371     INDICES(1,u).est_int = mean(INDICES(1,u).y, 1); % Estimate the ...
            integral
372     INDICES(2,u).est_int = mean(INDICES(2,u).y, 1); % Estimate the ...
            integral
373 end
374
375
376 %% Compute initial FWT
377 nllstart = int64(2^(out_param.mmin-r_lag-1));
378 for l=0:out_param.mmin-1 % We need the FWT for fx, fx^2, and the y ...
        values (a1/[a2-a3])
379     nl=2^l;
380     nmminlm1=2^(out_param.mmin-l-1);
381     ptind=repmat([true(nl,1); false(nl,1)],nmminlm1,1);
382     for u = 1:out_param.d
383         for r = 1:2
384             evenval=INDICES(r,u).y(ptind, :);
385             oddval=INDICES(r,u).y(~ptind, :);
386             INDICES(r,u).y(ptind, :)=(evenval+oddval)/2;
387             INDICES(r,u).y(~ptind, :)=(evenval-oddval)/2;
388         end
389     end
390     % Just above we compute the numerator integrals, below we ...
            compute the
391     % numerator integrals
392     evenval=fx(ptind);
393     oddval=fx(~ptind);
394     fx(ptind)=(evenval+oddval)/2;
395     fx(~ptind)=(evenval-oddval)/2;
396     evenval=fx2(ptind);
397     oddval=fx2(~ptind);
398     fx2(ptind)=(evenval+oddval)/2;
399     fx2(~ptind)=(evenval-oddval)/2;
400 end
401 %y now contains the FWT coefficients
402
403 for u = 1:out_param.d + 1 % u == d+1 is the kappanumap for fx2 and fx
404     for r = 1:2
405         %% Create kappanumap implicitly from the data
406         for l=out_param.mmin-1:-1:1
407            nl=2^l;
```

```
408             % for u = d+1 we compute the denominator which is the ...
                    same for
409             % all indices. That is why we store the y values and ...
                    kappanumap
410             % separetly for the denominator
411             if u < out_param.d + 1
412                 for p = 1:size(INDICES(r,u).y, 2)
413                     oldone=abs(INDICES(r,u). ...
414                         y(INDICES(r,u).kappanumap(2:nl, p), p)); ...
                            %earlier values of kappa, don't touch ...
                            first one
415                     newone=abs(INDICES(r,u). ...
416                         y(INDICES(r,u).kappanumap(nl+2:2*nl, p), p)); ...
                            %later values of kappa
417                     flip = find(newone > oldone); %which in the pair ...
                            are the larger ones
418                     if ¬isempty(flip)
419                         flipall = bsxfun(@plus, flip, ...
                                0:2^(l+1):2^out_param.mmin-1);
420                         temp = INDICES(r,u).kappanumap(nl+1+flipall, ...
                                p); %then flip
421                         INDICES(r,u).kappanumap(nl+1+flipall, p) = ...
                                INDICES(r,u).kappanumap(1+flipall, p); %them
422                         INDICES(r,u).kappanumap(1+flipall, p) = temp; ...
                                %around
423                     end
424                 end
425             elseif r == 1 % We keep the kappamap for int fx2
426                 oldone=abs(fx2(kappanumap_fx2_fx(2:nl,1))); %earlier ...
                        values of kappa, don't touch first one
427                 newone=abs(fx2(kappanumap_fx2_fx(nl+2:2*nl,1))); ...
                        %later values of kappa
428                 flip=find(newone>oldone); %which in the pair are the ...
                        larger ones
429                 if ¬isempty(flip)
430                     flipall=bsxfun(@plus, flip, ...
                            0:2^(l+1):2^out_param.mmin-1);
431 %                     flipall=flipall(:);
432                     temp = kappanumap_fx2_fx(nl+1+flipall,1); %then flip
433                     kappanumap_fx2_fx(nl+1+flipall,1) = ...
                            kappanumap_fx2_fx(1+flipall,1); %them
434                     kappanumap_fx2_fx(1+flipall,1) = temp; %around
435                 end
436             else % We keep the kappamap for int fx
437                 oldone=abs(fx(kappanumap_fx2_fx(2:nl,2))); %earlier ...
                        values of kappa, don't touch first one
438                 newone=abs(fx(kappanumap_fx2_fx(nl+2:2*nl,2))); ...
                        %later values of kappa
439                 flip=find(newone>oldone); %which in the pair are the ...
                        larger ones
440                 if ¬isempty(flip)
441                     flipall=bsxfun(@plus, flip, ...
                            0:2^(l+1):2^out_param.mmin-1);
442 %                     flipall=flipall(:);
```

```
443                     temp=kappanumap_fx2_fx(nl+1+flipall,2); %then flip
444                     kappanumap_fx2_fx(nl+1+flipall,2) = ...
                            kappanumap_fx2_fx(1+flipall,2); %them
445                     kappanumap_fx2_fx(1+flipall,2) = temp; %around
446                 end
447             end
448         end
449     end
450 end
451
452 %% Compute Stilde
453 % We keep the error estimates for int fx2
454 Stilde_fx2_fx(1,1) = ...
        sum(abs(fx2(kappanumap_fx2_fx(nllstart+1:2*nllstart,1))));
455 est_int_fx2_fx(1,1,end) = mean(fx2val); % Estimate the integral
456 err_bound_int_fx2_fx(1,1) = ...
        out_param.fudge(out_param.mmin)*Stilde_fx2_fx(1,1);
457 % We keep the error estimates for int fx
458 Stilde_fx2_fx(1,2) = ...
        sum(abs(fx(kappanumap_fx2_fx(nllstart+1:2*nllstart,2))));
459 est_int_fx2_fx(1,2) = mean(fxval); % Estimate the integral
460 err_bound_int_fx2_fx(1,2) = ...
        out_param.fudge(out_param.mmin)*Stilde_fx2_fx(1,2);
461 int = est_int_fx2_fx(1,2); % Estimate of the expectation of the function
462
463 for u = 1:out_param.d
464     for r = 1:2
465         INDICES(r,u).Stilde(1, :) = ...
                sum(abs(INDICES(r,u).y(INDICES(r,u). ...
466              kappanumap(nllstart+1:2*nllstart, :))), 1);
467         INDICES(r,u).err_bound_int(1, :) = ...
                out_param.fudge(out_param.mmin)*INDICES(r,u).Stilde(1, :);
468
469         down = [INDICES(r,u).est_int(1, :) - ...
                INDICES(r,u).err_bound_int(1, :), est_int_fx2_fx(1,1) - ...
                err_bound_int_fx2_fx(1,1), est_int_fx2_fx(1,2) - ...
                err_bound_int_fx2_fx(1,2)];
470         up = [INDICES(r,u).est_int(1, :) + ...
                INDICES(r,u).err_bound_int(1, :), est_int_fx2_fx(1,1) + ...
                err_bound_int_fx2_fx(1,1), est_int_fx2_fx(1,2) + ...
                err_bound_int_fx2_fx(1,2)];
471         q(r,u) = 1/2*(INDICES(r,u).Smax(down,up) + ...
                INDICES(r,u).Smin(down,up));
472         out_param.bound_err(r,u) = 1/2*(INDICES(r,u).Smax(down,up) - ...
                INDICES(r,u).Smin(down,up));
473         INDICES(r,u).errest(1) = out_param.bound_err(r,u);
474     end
475 end
476
477 % Necessary conditions for all indices integrals and fx and fx2
478 for l = l_star:out_param.mmin % Storing the information for the ...
        necessary conditions
479     C_low = 1/(1+omg_hat(out_param.mmin-l)*omg_circ(out_param.mmin-l));
480     C_up = 1/(1-omg_hat(out_param.mmin-l)*omg_circ(out_param.mmin-l));
```

```
481     for u = 1:out_param.d
482         for r = 1:2
483             INDICES(r,u).CStilde_low(l-l_star+1, :) = ...
                    max(INDICES(r,u). ...
484                 CStilde_low(l-l_star+1, :), ...
                        C_low*sum(abs(INDICES(r,u). ...
485                 y(INDICES(r,u).kappanumap(2^(l-1)+1:2^l, :))), 1));
486             if (omg_hat(out_param.mmin-l)* ...
487                     omg_circ(out_param.mmin-l) < 1)
488                 INDICES(r,u).CStilde_up(l-l_star+1, :) = ...
                        min(INDICES(r,u). ...
489                     CStilde_up(l-l_star+1, :), ...
                            C_up*sum(abs(INDICES(r,u). ...
490                     y(INDICES(r,u).kappanumap(2^(l-1)+1:2^l, :))), 1));
491             end
492         end
493     end
494     CStilde_low_fx2_fx(l-l_star+1,1) = ...
            max(CStilde_low_fx2_fx(l-l_star+1,1), ...
495         C_low*sum(abs(fx2(kappanumap_fx2_fx(2^(l-1)+1:2^l,1)))));
496     CStilde_low_fx2_fx(l-l_star+1,2) = ...
            max(CStilde_low_fx2_fx(l-l_star+1,2), ...
497         C_low*sum(abs(fx(kappanumap_fx2_fx(2^(l-1)+1:2^l,2)))));
498     if (omg_hat(out_param.mmin-l)*omg_circ(out_param.mmin-l) < 1)
499         CStilde_up_fx2_fx(l-l_star+1,1) = ...
                min(CStilde_up_fx2_fx(l-l_star+1,1), ...
500             C_up*sum(abs(fx2(kappanumap_fx2_fx(2^(l-1)+1:2^l,1)))));
501         CStilde_up_fx2_fx(l-l_star+1,2) = ...
                min(CStilde_up_fx2_fx(l-l_star+1,2), ...
502             C_up*sum(abs(fx(kappanumap_fx2_fx(2^(l-1)+1:2^l,2)))));
503     end
504 end
505 aux_bool = any(any(CStilde_low_fx2_fx > CStilde_up_fx2_fx)); % ...
        Variable that checks conditions violated for fx2 and fx
506 for u = 1:out_param.d
507     for r = 1:2
508         if any(any(INDICES(r,u).CStilde_low > ...
                INDICES(r,u).CStilde_up)) || aux_bool
509             out_param.exit(2,r,u) = true;
510         end
511     end
512 end
513
514 for u = 1:out_param.d
515     for r = 1:2
516         % Check the end of the algorithm
517         Δplus = 0.5*(gail.tolfun(out_param.abstol,...
518             out_param.reltol,out_param.theta,abs(q(r,u)- ...
519             INDICES(r,u).errest(1)), out_param.toltype)+ ...
520             gail.tolfun(out_param.abstol,out_param.reltol,...
521             out_param.theta,abs(q(r,u)+INDICES(r,u).errest(1)), ...
522             out_param.toltype));
523         Δminus = 0.5*(gail.tolfun(out_param.abstol,...
524             out_param.reltol,out_param.theta,abs(q(r,u)- ...
```

```
525             INDICES(r,u).errest(1)), out_param.toltype)- ...
526             gail.tolfun(out_param.abstol,out_param.reltol,...
527             out_param.theta,abs(q(r,u)+INDICES(r,u).errest(1)), ...
528             out_param.toltype));
529
530         q(r,u) = q(r,u)+Δminus;
531         if out_param.bound_err(r,u) ≤ Δplus
532            converged(r,u) = true;
533         elseif out_param.mmin == out_param.mmax % We are on our max ...
               budget and did not meet the error condition => overbudget
534            out_param.exit(1,r,u) = true;
535         end
536     end
537 end
538 out_param.time=toc(t_start);
539
540 %% Loop over m
541 for m = out_param.mmin+1:out_param.mmax
542     if all(all(converged))
543        break;
544     end
545     mnext=m-1;
546     nnext=2^mnext;
547     xnext=sobstr(n0+(1:nnext),1:3*out_param.d);
548     n0=n0+nnext;
549     fxnext = f(xnext(:,1:out_param.d)); %evaluate integrands y3
550     fy = f(xnext(:,out_param.d+1:2*out_param.d)); % We evaluate the ...
           f at the replicated points
551     fx2next = fxnext.^2; %evaluate integrands y2
552     fxval = [fxval; fxnext];
553     fx2val = [fx2val; fx2next];
554
555     %% Compute initial FWT on next points only for fx and fx2
556     % We need to split fx2 and fx from the indices because the ...
           number of
557     % data points used will vary depending on the indice but we will ...
           always
558     % still need fx and fx2. Keeping altogehter requires too many if's.
559     nllstart=int64(2^(m-r_lag-1));
560     meff=m-out_param.mmin+1;
561     for l=0:mnext-1
562         nl=2^l;
563         nmminlm1=2^(mnext-l-1);
564         ptind=repmat([true(nl,1); false(nl,1)],nmminlm1,1);
565         evenval=fxnext(ptind);
566         oddval=fxnext(¬ptind);
567         fxnext(ptind)=(evenval+oddval)/2;
568         fxnext(¬ptind)=(evenval-oddval)/2;
569         evenval=fx2next(ptind);
570         oddval=fx2next(¬ptind);
571         fx2next(ptind)=(evenval+oddval)/2;
572         fx2next(¬ptind)=(evenval-oddval)/2;
573     end
574     %% Compute FWT on all points only for fx and fx2
```

```
575     fx = [fx; fxnext];
576     fx2 = [fx2; fx2next];
577     nl=2^mnext;
578     ptind=[true(nl,1); false(nl,1)];
579     evenval=fx(ptind);
580     oddval=fx(¬ptind);
581     fx(ptind)=(evenval+oddval)/2;
582     fx(¬ptind)=(evenval-oddval)/2;
583     evenval=fx2(ptind);
584     oddval=fx2(¬ptind);
585     fx2(ptind)=(evenval+oddval)/2;
586     fx2(¬ptind)=(evenval-oddval)/2;
587     %% Update kappanumap only for fx and fx2
588     kappanumap_fx2_fx = [kappanumap_fx2_fx ; 2^(m-1) + ...
            kappanumap_fx2_fx]; %initialize map only for fx and fx2
589     for l=m-1:-1:m-r_lag
590         nl=2^l;
591         oldone=abs(fx2(kappanumap_fx2_fx(2:nl,1))); %earlier values ...
                of kappa, don't touch first one
592         newone=abs(fx2(kappanumap_fx2_fx(nl+2:2*nl,1))); %later ...
                values of kappa
593         flip=find(newone>oldone);
594         if ¬isempty(flip)
595           flipall=bsxfun(@plus,flip,0:2^(l+1):2^m-1);
596           temp=kappanumap_fx2_fx(nl+1+flipall,1);
597           kappanumap_fx2_fx(nl+1+flipall,1)= ...
598               kappanumap_fx2_fx(1+flipall,1);
599           kappanumap_fx2_fx(1+flipall,1)=temp;
600         end
601         oldone=abs(fx(kappanumap_fx2_fx(2:nl,2))); %earlier values ...
                of kappa, don't touch first one
602         newone=abs(fx(kappanumap_fx2_fx(nl+2:2*nl,2))); %later ...
                values of kappa
603         flip=find(newone>oldone);
604         if ¬isempty(flip)
605           flipall=bsxfun(@plus,flip,0:2^(l+1):2^m-1);
606           temp=kappanumap_fx2_fx(nl+1+flipall,2);
607           kappanumap_fx2_fx(nl+1+flipall,2)= ...
608               kappanumap_fx2_fx(1+flipall,2);
609           kappanumap_fx2_fx(1+flipall,2)=temp;
610         end
611     end
612
613     %% Compute Stilde for fx and fx2 only
614     % We keep the error estimates and integrals only for int fx2
615     Stilde_fx2_fx(meff,1) = ...
            sum(abs(fx2(kappanumap_fx2_fx(nllstart+1:2*nllstart,1))));
616     est_int_fx2_fx(meff,1) = mean(fx2val); % Estimate the integral ...
            of f^2
617     err_bound_int_fx2_fx(meff,1) = ...
            out_param.fudge(m)*Stilde_fx2_fx(meff,1);
618     % We keep the error estimates and integrals only for int fx
619     Stilde_fx2_fx(meff,2) = ...
            sum(abs(fx(kappanumap_fx2_fx(nllstart+1:2*nllstart,2))));
```

```
620     est_int_fx2_fx(meff,2) = mean(fxval); % Estimate the integral of f
621     err_bound_int_fx2_fx(meff,2) = ...
            out_param.fudge(m)*Stilde_fx2_fx(meff,2);
622     int = est_int_fx2_fx(meff,2); % Estimate of the expectation of ...
            the function
623
624     %% We start computing everything for the numerator. fx and fx2 ...
            were for
625     % the denominator only, which is common with all indices
626     fzx = [];
627     for u = 1:out_param.d
628         for r = 1:2
629             if ¬converged(r,u)
630                 if r == 1 % For a given u, fxy is the same for the first
631                           % order and total effect. Thus, we only ...
                              need to
632                           % compute it one time for each u.
633                     fxy = f([xnext(:,out_param.d+1:out_param.d+u-1) ...
                            xnext(:,u) ...
                            xnext(:,out_param.d+u+1:2*out_param.d)]);
634                 % If we did not evaluate it for r == 1 because it ...
                      has already
635                 % converged, we evaluate it for r == 2 not converged
636                 elseif r == 2 && converged(1,u)
637                     fxy = f([xnext(:,out_param.d+1:out_param.d+u-1) ...
                            xnext(:,u) ...
                            xnext(:,out_param.d+u+1:2*out_param.d)]);
638                 end
639                 % If we are working on a small indice, we also need to
640                 % evaluate fzx
641                 if r == 1 && out_param.small(1,u) == 1 && isempty(fzx)
642                     % We only evaluate fzx if isempty and we found 1 ...
                          small
643                     fzx = f([xnext(:,1:u-1) xnext(:,2*out_param.d + ...
                            u) xnext(:,u+1:out_param.d)]);
644                 end
645                 % Evaluation y only
646                 INDICES(r,u).n = 2^m;
647                 out_param.n(r,u) = 2^m;
648                 % We evaluate all functions in the numerator
649                 ynext = [];
650                 for func = 1:size(INDICES(r,u).f, 2)
651                     ynext = [ynext ...
                            INDICES(r,u).f{func}(xnext,u,fxval(end/2 + ...
                            1:end),fy,fxy,fzx)];
652                 end
653                 fzx = [];
654                 INDICES(r,u).est_int(meff, :) = ...
                        1/2*(INDICES(r,u).est_int(meff-1, :) + ...
                        mean(ynext, 1)); % Estimate the integral
655
656                 %% Compute initial FWT on next points only for y
657                 nllstart=int64(2^(m-r_lag-1));
658                 meff=m-out_param.mmin+1;
```

```
659                 for l=0:mnext-1
660                     nl=2^l;
661                     nmminlm1=2^(mnext-l-1);
662                     ptind=repmat([true(nl,1); false(nl,1)],nmminlm1,1);
663                     evenval=ynext(ptind, :);
664                     oddval=ynext(¬ptind, :);
665                     ynext(ptind, :)=(evenval+oddval)/2;
666                     ynext(¬ptind, :)=(evenval-oddval)/2;
667                 end
668                 %% Compute FWT on all points only for y
669                 INDICES(r,u).y = [INDICES(r,u).y; ynext];
670                 nl=2^mnext;
671                 ptind=[true(nl,1); false(nl,1)];
672                 evenval = INDICES(r,u).y(ptind, :);
673                 oddval = INDICES(r,u).y(¬ptind, :);
674                 INDICES(r,u).y(ptind, :) = (evenval+oddval)/2;
675                 INDICES(r,u).y(¬ptind, :) = (evenval-oddval)/2;
676
677                 %% Update kappanumap only for indices
678                 INDICES(r,u).kappanumap = [INDICES(r,u).kappanumap ; ...
                        2^(m-1)+INDICES(r,u).kappanumap];
679                 for l=m-1:-1:m-r_lag
680                   nl=2^l;
681                   for p = 1:size(INDICES(r,u).y, 2)
682                       oldone=abs(INDICES(r,u).y(INDICES(r,u). ...
683                           kappanumap(2:nl, p), p)); %earlier values ...
                                of kappa, don't touch first one
684                       newone=abs(INDICES(r,u).y(INDICES(r,u). ...
685                           kappanumap(nl+2:2*nl, p), p)); %later ...
                                values of kappa
686                       flip = find(newone > oldone);
687                       if ¬isempty(flip)
688                           flipall = bsxfun(@plus, flip, ...
                                0:2^(l+1):2^m-1);
689                           temp = ...
                                INDICES(r,u).kappanumap(nl+1+flipall, p);
690                           INDICES(r,u).kappanumap(nl+1+flipall, p) = ...
                                INDICES(r,u).kappanumap(1+flipall, p);
691                           INDICES(r,u).kappanumap(1+flipall, p) = temp;
692                       end
693                   end
694                 end
695
696                 %% Compute Stilde for y only
697                 INDICES(r,u).Stilde(meff, :) = sum(abs(INDICES(r,u). ...
698                     y(INDICES(r,u).kappanumap(nllstart+ ...
699                     1:2*nllstart, :))), 1);
700                 INDICES(r,u).err_bound_int(meff, :) = ...
701                     out_param.fudge(m)*INDICES(r,u).Stilde(meff, :); ...
                            % Only error bound for the integral on the ...
                            numerator
702
703                 down = [INDICES(r,u).est_int(meff, :) - ...
                        INDICES(r,u).err_bound_int(meff, :), ...
```

```
                    est_int_fx2_fx(meff, :) - ...
                    err_bound_int_fx2_fx(meff, :)];
704                 up = [INDICES(r,u).est_int(meff, :) + ...
                    INDICES(r,u).err_bound_int(meff, :), ...
                    est_int_fx2_fx(meff, :) + ...
                    err_bound_int_fx2_fx(meff, :)];
705
706                 q(r,u) = 1/2*(INDICES(r,u).Smax(down,up) + ...
                    INDICES(r,u).Smin(down,up));
707                 out_param.bound_err(r,u) = ...
                    1/2*(INDICES(r,u).Smax(down,up) - ...
                    INDICES(r,u).Smin(down,up));
708                 INDICES(r,u).errest(meff) = out_param.bound_err(r,u);
709             end
710         end
711     end
712
713     % Necessary conditions for all indices integrals and fx and fx2
714     for l = l_star:m % Storing the information for the necessary ...
        conditions
715         C_low = 1/(1+omg_hat(out_param.mmin-l)* ...
716             omg_circ(out_param.mmin-l));
717         C_up = 1/(1-omg_hat(out_param.mmin-l)* ...
718             omg_circ(out_param.mmin-l));
719         for u = 1:out_param.d
720             for r = 1:2
721                 if ¬converged(r,u)
722                     INDICES(r,u).CStilde_low(l-l_star+1, :) = max( ...
723                         INDICES(r,u).CStilde_low(l-l_star+1, :), ...
724                         C_low*sum(abs(INDICES(r,u).y(INDICES(r,u). ...
725                         kappanumap(2^(l-1)+1:2^l, :))), 1));
726                     if (omg_hat(out_param.mmin-l)* ...
727                             omg_circ(out_param.mmin-l) < 1)
728                         INDICES(r,u).CStilde_up(l-l_star+1, :) = ...
729                             min(INDICES(r,u).CStilde_up(l- ...
730                             l_star+1,:),C_up*sum(abs(INDICES(r,u). ...
731                             y(INDICES(r,u).kappanumap(2^(l-1)+ ...
732                             1:2^l, :))), 1));
733                     end
734                 end
735             end
736         end
737         CStilde_low_fx2_fx(l-l_star+1,1) = max( ...
738             CStilde_low_fx2_fx(l-l_star+1,1),C_low* ...
739             sum(abs(fx2(kappanumap_fx2_fx(2^(l-1)+1:2^l,1))))));
740         CStilde_low_fx2_fx(l-l_star+1,2) = max( ...
741             CStilde_low_fx2_fx(l-l_star+1,2),C_low* ...
742             sum(abs(fx(kappanumap_fx2_fx(2^(l-1)+1:2^l,2))))));
743         if (omg_hat(out_param.mmin-l)*omg_circ(out_param.mmin-l) < 1)
744             CStilde_up_fx2_fx(l-l_star+1,1) = min( ...
745                 CStilde_up_fx2_fx(l-l_star+1,1),C_up* ...
746                 sum(abs(fx2(kappanumap_fx2_fx(2^(l-1)+1:2^l,1))))));
747             CStilde_up_fx2_fx(l-l_star+1,2) = min( ...
748                 CStilde_up_fx2_fx(l-l_star+1,2),C_up* ...
```

```
749                 sum(abs(fx(kappanumap_fx2_fx(2^(l-1)+1:2^l,2))));
750         end
751     end
752     aux_bool = any(any(CStilde_low_fx2_fx > CStilde_up_fx2_fx)); % ...
            Variable that checks conditions violated for fx2 and fx
753     for u = 1:out_param.d
754         for r = 1:2
755             if ¬converged(r,u) && (any(any(INDICES(r,u).CStilde_low ...
                > INDICES(r,u).CStilde_up)) || aux_bool)
756                 out_param.exit(2,r,u) = true;
757             end
758         end
759     end
760
761     for u = 1:out_param.d
762         for r = 1:2
763             if ¬converged(r,u)
764                 % Check the end of the algorithm
765                 Δplus = 0.5*(gail.tolfun(out_param.abstol,...
766                     out_param.reltol,out_param.theta,abs(q(r,u)- ...
767                     INDICES(r,u).errest(meff)), out_param.toltype)+ ...
768                     gail.tolfun(out_param.abstol,out_param.reltol,...
769                     out_param.theta,abs(q(r,u)+INDICES(r,u). ...
770                     errest(meff)),out_param.toltype));
771                 Δminus = 0.5*(gail.tolfun(out_param.abstol,...
772                     out_param.reltol,out_param.theta,abs(q(r,u)- ...
773                     INDICES(r,u).errest(meff)),out_param.toltype)- ...
774                     gail.tolfun(out_param.abstol,out_param.reltol,...
775                     out_param.theta,abs(q(r,u)+INDICES(r,u). ...
776                     errest(meff)),out_param.toltype));
777
778                 q(r,u) = q(r,u) + Δminus;
779                 if out_param.bound_err(r,u) ≤ Δplus
780                    converged(r,u) = true;
781                 elseif m == out_param.mmax % We are on our max ...
                       budget and did not meet the error condition => ...
                       overbudget
782                    out_param.exit(1,r,u) = true;
783                 end
784             end
785         end
786     end
787     out_param.time=toc(t_start);
788 end
789
790 % Decode the exit structure
791 for u = 1:out_param.d
792     for r = 1:2
793         out_param.exitflag(:,r,u) = ...
              (2.^(0:exit_len-1))'.*out_param.exit(:,r,u);
794     end
795 end
796 out_param = rmfield(out_param,'exit');
797
```

```
798 out_param.time=toc(t_start);
799 end
800
801
802 %% Parsing for the input of cubSobol_SI_all_g
803 function [f,hyperbox, out_param] = ...
        cubSobol_SI_all_g_param(r_lag,varargin)
804
805 % Default parameter values
806 default.hyperbox = [zeros(1,1);ones(1,1)];% default hyperbox
807 default.measure  = 'uniform';
808 default.abstol  = 1e-4;
809 default.reltol  = 1e-2;
810 default.mmin  = 10;
811 default.mmax  = 24;
812 default.fudge = @(m) 5*2.^-m;
813 default.toltype  = 'max';
814 default.theta  = 1;
815 default.threshold_small = 0.1;
816
817 if numel(varargin)<2
818     help cubSobol_SI_all_g
819     warning('GAIL:cubSobol_SI_all_g:fdnotgiven',...
820         'At least, function f and hyperbox need to be specified. ...
               Example for f(x)=x^2:')
821     f = @(x) x.^2;
822     out_param.f=f;
823     hyperbox = default.hyperbox;
824 else
825     f = varargin{1};
826     if ~gail.isfcn(f)
827         warning('GAIL:cubSobol_SI_all_g:fnotfcn',...
828             'The given input f was not a function. Example for ...
                   f(x)=x^2:')
829         f = @(x) x.^2;
830         out_param.f=f;
831         hyperbox = default.hyperbox;
832     else
833         out_param.f=f;
834         hyperbox = varargin{2};
835         if ~isnumeric(hyperbox) || ~(size(hyperbox,1)==2) || ~...
               (size(hyperbox,2)<370)
836             warning('GAIL:cubSobol_SI_all_g:hyperbox_error1',...
837                 'The hyperbox must be a real matrix of size 2xd ...
                       where d can not be greater than 370. Example for ...
                       f(x)=x^2:')
838             f = @(x) x.^2;
839             out_param.f=f;
840             hyperbox = default.hyperbox;
841         end
842     end
843 end
844
845 validvarargin=numel(varargin)>2;
```

```
846 if validvarargin
847     in3=varargin(3:end);
848     for j=1:numel(varargin)-2
849     validvarargin=validvarargin && (isnumeric(in3{j}) ...
850         || ischar(in3{j}) || isstruct(in3{j}) || gail.isfcn(in3{j}));
851     end
852     if ¬validvarargin
853         warning('GAIL:cubSobol_SI_all_g:validvarargin','Optional ...
                parameters must be numeric or strings. We will use the ...
                default optional parameters.')
854     end
855     in3=varargin{3};
856 end
857
858 MATLABVERSION = gail.matlab_version;
859 if MATLABVERSION ≥ 8.3
860   f_addParamVal = @addParameter;
861 else
862   f_addParamVal = @addParamValue;
863 end
864
865 if ¬validvarargin
866     out_param.measure = default.measure;
867     out_param.abstol = default.abstol;
868     out_param.reltol = default.reltol;
869     out_param.mmin = default.mmin;
870     out_param.mmax = default.mmax;
871     out_param.fudge = default.fudge;
872     out_param.toltype = default.toltype;
873     out_param.theta = default.theta;
874     out_param.threshold_small = default.threshold_small;
875 else
876     p = inputParser;
877     addRequired(p,'f',@gail.isfcn);
878     addRequired(p,'hyperbox',@isnumeric);
879     if isnumeric(in3) || ischar(in3)
880         addOptional(p,'measure',default.measure,...
881             @(x) any(validatestring(x, {'uniform','normal'})));
882         addOptional(p,'abstol',default.abstol,@isnumeric);
883         addOptional(p,'reltol',default.reltol,@isnumeric);
884         addOptional(p,'mmin',default.mmin,@isnumeric);
885         addOptional(p,'mmax',default.mmax,@isnumeric);
886         addOptional(p,'fudge',default.fudge,@gail.isfcn);
887         addOptional(p,'toltype',default.toltype,...
888             @(x) any(validatestring(x, {'max','comb'})));
889         addOptional(p,'theta',default.theta,@isnumeric);
890         addOptional(p,'threshold_small', default.threshold_small, ...
                @isnumeric);
891     else
892         if isstruct(in3) %parse input structure
893             p.StructExpand = true;
894             p.KeepUnmatched = true;
895         end
896         f_addParamVal(p,'measure',default.measure,...
```

```
897             @(x) any(validatestring(x, {'uniform','normal'})));
898         f_addParamVal(p,'abstol',default.abstol,@isnumeric);
899         f_addParamVal(p,'reltol',default.reltol,@isnumeric);
900         f_addParamVal(p,'mmin',default.mmin,@isnumeric);
901         f_addParamVal(p,'mmax',default.mmax,@isnumeric);
902         f_addParamVal(p,'fudge',default.fudge,@gail.isfcn);
903         f_addParamVal(p,'toltype',default.toltype,...
904             @(x) any(validatestring(x, {'max','comb'})));
905         f_addParamVal(p,'theta',default.theta,@isnumeric);
906         f_addParamVal(p,'threshold_small',default. ...
907             threshold_small,@isnumeric);
908     end
909     parse(p,f,hyperbox,varargin{3:end})
910     out_param = p.Results;
911 end
912
913 out_param.d = size(hyperbox,2);
914
915 % fdgyes = 0; % We store how many functions are in varargin. There ...
      can only
916 %             % two functions as input, the function f and the fudge ...
      factor.
917 % for j = 1:size(varargin,2)
918 %     fdgyes = gail.isfcn(varargin{j})+fdgyes;
919 % end
920 % if fdgyes < 2 % No fudge factor given as input
921 %     default.fudge = @(m) 5*2.^-(m/d);
922 % end
923
924 %hyperbox should be 2 x dimension
925 if ¬isnumeric(hyperbox) || ¬(size(hyperbox,1)==2) || ¬(out_param.d<370)
926     warning('GAIL:cubSobol_SI_all_g:hyperbox_error2',...
927         'The hyperbox must be a real matrix of size 2 x d where d ...
              can not be greater than 370. Example for f(x)=x^2:')
928     f = @(x) x.^2;
929     out_param.f=f;
930     hyperbox = default.hyperbox;
931 end
932
933 % Force measure to be uniform or normal only
934 if ¬(strcmp(out_param.measure,'uniform') || ...
    strcmp(out_param.measure,'normal') )
935     warning('GAIL:cubSobol_SI_all_g:notmeasure',['The measure can ...
          only be uniform or normal.' ...
936             ' Using default measure ' num2str(default.measure)])
937     out_param.measure = default.measure;
938 end
939
940 % Force absolute tolerance greater than 0
941 if (out_param.abstol < 0 )
942     warning('GAIL:cubSobol_SI_all_g:abstolnonpos',['Absolute ...
          tolerance cannot be negative.' ...
943             ' Using default absolute tolerance ' ...
                num2str(default.abstol)])
```

```
944     out_param.abstol = default.abstol;
945 end
946
947 % Force relative tolerance greater than 0 and smaller than 1
948 if (out_param.reltol < 0) || (out_param.reltol > 1)
949     warning('GAIL:cubSobol_SI_all_g:reltolnonunit',['Relative ...
            tolerance should be chosen in [0,1].' ...
950             ' Using default relative tolerance ' ...
                    num2str(default.reltol)])
951     out_param.reltol = default.reltol;
952 end
953
954 % Force mmin to be integer greater than 0
955 if (¬gail.isposint(out_param.mmin) || ¬(out_param.mmin < ...
        out_param.mmax+1))
956     warning('GAIL:cubSobol_SI_all_g:lowmmin',['The minimum starting ...
            exponent ' ...
957             'should be an integer greater than 0 and smaller or ...
                    equal than the maxium.' ...
958             ' Using default mmin ' num2str(default.mmin)])
959     out_param.mmin = default.mmin;
960 end
961
962 % Force mmin to be integer greater than r_lag (so that ...
        l_star=mmin-r_lag≥0)
963 if out_param.mmin < r_lag
964     warning('GAIL:cubSobol_SI_all_g:lowmminrlag',['The minimum ...
            starting exponent ' ...
965             'should be at least ' num2str(r_lag) '.' ...
966             ' Using default mmin ' num2str(default.mmin)])
967     out_param.mmin = default.mmin;
968 end
969
970 % Force exponent budget number of points be a positive integer ...
        greater than
971 % or equal to mmin an smaller than 54
972 if ¬(gail.isposint(out_param.mmax) && out_param.mmax≥out_param.mmin ...
        && out_param.mmax≤53)
973     warning('GAIL:cubSobol_SI_all_g:wrongmmax',['The maximum ...
            exponent for the budget should be an integer biger than mmin ...
            and smaller than 54.' ...
974             ' Using default mmax ' num2str(default.mmax)])
975     out_param.mmax = default.mmax;
976 end
977
978 % Force fudge factor to be greater than 0
979 if ¬((gail.isfcn(out_param.fudge) && (out_param.fudge(1)>0)))
980     warning('GAIL:cubSobol_SI_all_g:fudgenonpos',['The fudge factor ...
            should be a positive function.' ...
981             ' Using default fudge factor ' func2str(default.fudge)])
982     out_param.fudge = default.fudge;
983 end
984
985 % Force toltype to be max or comb
```

```
986 if ¬(strcmp(out_param.toltype,'max') || ...
    strcmp(out_param.toltype,'comb') )
987     warning('GAIL:cubSobol_SI_all_g:nottoltype',['The error type can ...
            only be max or comb.' ...
988             ' Using default toltype ' num2str(default.toltype)])
989     out_param.toltype = default.toltype;
990 end
991
992 % Force theta to be in [0,1]
993 if (out_param.theta < 0) || (out_param.theta > 1)
994     warning('GAIL:cubSobol_SI_all_g:thetanonunit',['Theta should be ...
            chosen in [0,1].' ...
995             ' Using default theta ' num2str(default.theta)])
996     out_param.theta = default.theta;
997 end
998
999 % Force threshold_small to be in [0,1]
1000 if (out_param.threshold_small < 0) || (out_param.threshold_small > 1)
1001     warning('GAIL:cubSobol_SI_all_g:threshold_smallnonunit', ...
1002         ['Threshold_small should be chosen in [0,1].' ...
1003             ' Using default threshold_small ' ...
                 num2str(default.threshold_small)])
1004     out_param.threshold_small = default.threshold_small;
1005 end
1006
1007 % Checking on pure absolute/relative error
1008 if (out_param.abstol==0) && (out_param.reltol==0)
1009     warning('GAIL:cubSobol_SI_all_g:tolzeros',['Absolute and ...
            relative error tolerances can not be simultaniusly 0.' ...
1010             ' Using default absolute tolerance ' ...
                 num2str(default.abstol) ' and relative tolerance ' ...
                 num2str(default.reltol)])
1011     out_param.abstol = default.abstol;
1012     out_param.reltol = default.reltol;
1013 end
1014 if (strcmp(out_param.toltype,'comb')) && (out_param.theta==1) && ...
    (out_param.abstol==0)
1015     warning('GAIL:cubSobol_SI_all_g:abstolzero',['When choosing ...
            toltype comb, if theta=1 then abstol>0.' ...
1016             ' Using default absolute tolerance ' ...
                 num2str(default.abstol) ])
1017     out_param.abstol = default.abstol;
1018 end
1019 if (strcmp(out_param.toltype,'comb')) && (out_param.theta==0) && ...
    (out_param.reltol==0)
1020     warning('GAIL:cubSobol_SI_all_g:reltolzero',['When choosing ...
            toltype comb, if theta=0 then reltol>0.' ...
1021             ' Using default relative tolerance ' ...
                 num2str(default.reltol) ])
1022     out_param.reltol = default.reltol;
1023 end
1024
1025 % Checking on the hyperbox given the measure
1026 if (strcmp(out_param.measure,'uniform')) && ¬...
```

```
    all(all(isfinite(hyperbox)))
1027     warning('GAIL:cubSobol_SI_all_g:hyperboxnotfinite',['If uniform ...
            measure, hyperbox must be of finite volume.' ...
1028             ' Using default hyperbox:'])
1029     disp([zeros(1,out_param.d);ones(1,out_param.d)])
1030     hyperbox = [zeros(1,out_param.d);ones(1,out_param.d)];
1031 end
1032 if (strcmp(out_param.measure,'normal')) && ...
    (sum(sum(isfinite(hyperbox)))>0)
1033     warning('GAIL:cubSobol_SI_all_g:hyperboxfinite',['If normal ...
            measure, hyperbox must be defined as (-Inf,Inf)^d.' ...
1034             ' Using default hyperbox:'])
1035     disp([-inf*ones(1,out_param.d);inf*ones(1,out_param.d)])
1036     hyperbox = [-inf*ones(1,out_param.d);inf*ones(1,out_param.d)];
1037 end
1038 if (strcmp(out_param.measure,'normal')) && ...
    (any(hyperbox(1,:)==hyperbox(2,:)) || ...
    any(hyperbox(1,:)>hyperbox(2,:)))
1039     warning('GAIL:cubSobol_SI_all_g:hyperboxnormalwrong',['If normal ...
            measure, hyperbox must be defined as (-Inf,Inf)^d.' ...
1040             ' Using default hyperbox:'])
1041     disp([-inf*ones(1,out_param.d);inf*ones(1,out_param.d)])
1042     hyperbox = [-inf*ones(1,out_param.d);inf*ones(1,out_param.d)];
1043 end
1044
1045 end
1046
1047
1048 function y = estimate_min(down, up)
1049     if down(4)*up(4) <= 0
1050         y = min(max(max(down(1)-max(down(2).^2, up(2).^2)...
1051         ,0)./max(up(3), eps), 0),1);
1052     else
1053         y = min(max(max(down(1)-max(down(2).^2, up(2).^2)...
1054         ,0)./max(up(3)-min(down(4).^2, up(4).^2), eps), 0),1);
1055     end
1056 end
1057
1058 function y = estimate_max(down, up)
1059     if down(2)*up(2) <= 0
1060         y = min(max(max(up(1)...
1061     ,0)./max(down(3)-max(down(4).^2, up(4).^2), eps), 0),1);
1062     else
1063         y = min(max(max(up(1)-min(down(2).^2, up(2).^2)...
1064     ,0)./max(down(3)-max(down(4).^2, up(4).^2), eps), 0),1);
1065     end
1066 end
```

# APPENDIX B
# MATLAB CODE FOR VARIANT B: `CUBSOBOL_SI_FO_G`

```
function [q,int,out_param] = cubSobol_SI_fo_g(varargin)
%CUBSOBOL_SI_FO_G Quasi-Monte Carlo method using Sobol' cubatures
%to compute all first order Sobol Indices whitin a specified
%generalized error tolerance with guarantees under Walsh-Fourier
%coefficients cone decay assumptions. It uses the replicated procedure.
%
%   [q,out_param] = CUBSOBOL_SI_FO_G(f,hyperbox) estimates all first ...
    order
%   Sobol Indices of f, where hyperbox is the sample space
%   of the uniform distribution (the distribution can also be set to ...
    normal),
%   and the estimation error is guaranteed not to be greater than a ...
    specific
%   generalized error tolerance tolfun:=max(abstol,reltol*| SI(f) |).
%   Input f is a function handle. f should accept an n x d matrix input,
%   where d is the dimension and n is the number of points being ...
    evaluated
%   simultaneously. The input hyperbox is a 2 x d matrix, where the ...
    first
%   row corresponds to the lower limits and the second row ...
    corresponds to
%   the upper limits of the integral. Given the construction of Sobol'
%   sequences, d must be a positive integer with 1≤d≤370.
%
%   q = CUBSOBOL_SI_FO_G(f,hyperbox,measure,abstol,reltol) estimates all
%   first order Sobol Indices of f. The answer is given within
%   the generalized error tolerance tolfun. All parameters should be
%   input in the order specified above. If an input is not specified,
%   the default value is used. Note that if an input is not specified,
%   the remaining tail cannot be specified either. Inputs f and hyperbox
%   are required. The other optional inputs are in the correct order:
%   measure,abstol,reltol,mmin,mmax,fudge,toltype and
%   theta.
%
%   q = CUBSOBOL_SI_FO_G(f,hyperbox,'measure',measure,'abstol',abstol,
%       'reltol',reltol)
%   estimates all first order Sobol Indices of f. The answer is given
%   within the generalized error tolerance tolfun. All the field-value
%   pairs are optional and can be supplied in any order. If an input ...
    is not
%   specified, the default value is used.
%
%   q = CUBSOBOL_SI_FO_G(f,hyperbox,in_param) estimates all first order
%   Sobol Indices of f. The answer is given within the generlized error
%   tolerance tolfun.
%
%   Input Arguments
%
%     f --- the integrand whose input should be a matrix n x d where ...
    n is
%     the number of data points and d the dimension, which cannot be
%     greater than 370. By default f is f=@ x.^2.
%
%     hyperbox --- sample space of the distribution that defines the ...
```

```
    random
47  %     input vector. It must be be a 2 x d matrix, where the first row
48  %     corresponds to the lower limits and the second row to the upper
49  %     limits. The default value is [0;1].
50  %
51  %     in_param.measure ---- for f(x)*mu(dx), we can define mu(dx) to ...
        be the
52  %     measure of a uniformly distributed random variable in the hyperbox
53  %     (each dimension is independent) or normally distributed
54  %     with covariance matrix I_d. The only possible values are
55  %     'uniform' or 'normal'. For 'uniform', the hyperbox must have
56  %     a finite volume while for 'normal', the hyperbox can only be ...
        defined as
57  %     (-Inf,Inf)^d. By default it is 'uniform'.
58  %
59  %     in_param.abstol ---- the absolute error tolerance, abstol≥0. By
60  %     default it is 1e-4.
61  %
62  %     in_param.reltol ---- the relative error tolerance, which should be
63  %     in [0,1]. Default value is 1e-2.
64  %
65  %    Optional Input Arguments
66  %
67  %     in_param.mmin ---- the minimum number of points to start is 2^mmin.
68  %     The cone condition on the Fourier coefficients decay requires a
69  %     minimum number of points to start. The advice is to consider ...
        at least
70  %     mmin=10. mmin needs to be a positive integer with mmin≤mmax. By
71  %     default it is 10.
72  %
73  %     in_param.mmax ---- the maximum budget is 2^mmax. By ...
        construction of
74  %     the Sobol' generator, mmax is a positive integer such that
75  %     mmin≤mmax≤53. The default value is 24.
76  %
77  %     in_param.fudge ---- the positive function multiplying the finite
78  %     sum of Fast Walsh Fourier coefficients specified in the cone ...
        of functions.
79  %     This input is a function handle. The fudge should accept an ...
        array of
80  %     nonnegative integers being evaluated simultaneously. For more
81  %     technical information about this parameter, refer to the ...
        references.
82  %     By default it is @(m) 5*2.^-m.
83  %
84  %     in_param.toltype ---- this is the generalized tolerance function.
85  %     There are two choices, 'max' which takes
86  %     max(abstol,reltol*| integral(f) | ) and 'comb' which is the ...
        linear combination
87  %     theta*abstol+(1-theta)*reltol*| integral(f) | . Theta is another
88  %     parameter to be specified with 'comb'(see below). For pure ...
        absolute
89  %     error, either choose 'max' and set reltol = 0 or choose 'comb' ...
        and set
```

```
90 %     theta = 1. For pure relative error, either choose 'max' and set
91 %     abstol = 0 or choose 'comb' and set theta = 0. Note that with ...
       'max',
92 %     the user can not input abstol = reltol = 0 and with 'comb', if ...
       theta = 1
93 %     abstol con not be 0 while if theta = 0, reltol can not be 0.
94 %     By default toltype is 'max'.
95 %
96 %     in_param.theta ---- this input is parametrizing the toltype
97 %     'comb'. Thus, it is only active when the toltype
98 %     chosen is 'comb'. It establishes the linear combination weight
99 %     between the absolute and relative tolerances
100 %    theta*abstol+(1-theta)*reltol*| integral(f) |. Note that for ...
       theta = 1,
101 %    we have pure absolute tolerance while for theta = 0, we have pure
102 %    relative tolerance. By default, theta=1.
103 %
104 %  Output Arguments
105 %
106 %    q ---- the estimated value of all Sobol Indices in matrix form.
107 %    Each column is the dimension for which we estimate the indice.
108 %
109 %    out_param.d ---- dimension of the domain of f.
110 %
111 %    out_param.n ---- number of Sobol' points used to compute q.
112 %
113 %    out_param.bound_err ---- predicted error bound of q based on ...
       the cone
114 %    conditions. If the function lies in the cone, the real error ...
       will be
115 %    smaller than generalized tolerance.
116 %
117 %    out_param.time ---- time elapsed in seconds when calling ...
       cubSobol_SI_fo_g.
118 %
119 %    out_param.exitflag ---- for each q, this is a binary vector stating
120 %    whether warning flags arise. This is a triple array element with
121 %    flags stored in the first component, and indice specified by the
122 %    other two. These flags tell about which conditions make the
123 %    final result certainly not guaranteed. One flag is considered ...
       arisen
124 %    when its value is 1. The following list explains the flags in the
125 %    respective vector order:
126 %
127 %                        1 : If reaching overbudget. It states whether
128 %                        the max budget is attained without reaching the
129 %                        guaranteed error tolerance.
130 %
131 %                        2 : If the function lies outside the cone. In
132 %                        this case, results are not guaranteed. For more
133 %                        information about the cone definition, check the
134 %                        article mentioned below.
135 %
136 %  Guarantee
```

```
137 % This algorithm computes first order and total effect Sobol Indices of
138 % real valued functions in [0,1)^d with a prescribed generalized error
139 % tolerance. The Walsh-Fourier coefficients of the integrand are assumed
140 % to be absolutely convergent. If the algorithm terminates without ...
      warning
141 % messages, the output is given with guarantees under the assumption ...
      that
142 % the integrand lies inside a cone of functions. The guarantee is ...
      based on
143 % the decay rate of the Walsh-Fourier coefficients. For more details ...
      on how
144 % the cone is defined, please refer to the references below.
145 %
146 %  Examples
147 %
148 % Example 1:
149 % Ishigami example:
150 %
151 % >> f = @(x) sin(x(:,1)).*(1+1/10*(x(:,3).^4))+7*sin(x(:,2)).^2; ...
      hyperbox = pi*[-ones(1,3) ; ones(1,3)];
152 % >> q = cubSobol_SI_fo_g(f,hyperbox,'uniform',1e-1,0); exactsol = ...
      [.3139051827, .4424111333, 0.];
153 % >> check = sum(sum(abs(exactsol-q) < ...
      gail.tolfun(1e-1,0,1,exactsol,'max')))
154 % check = 3
155 %
156 %
157 % Example 2:
158 % Bratley example:
159 %
160 % >> f = @(x) sum(bsxfun(@times, cumprod(x,2), (-1).^(1:6)),2); ...
      hyperbox = [-zeros(1,6) ; ones(1,6)];
161 % >> q = cubSobol_SI_fo_g(f,hyperbox,'uniform',1e-1,1e-1); exactsol ...
      = [.6528636616, .1791303924, 0.3701041165e-1, 0.1332374820e-1, ...
      0.1480416466e-2, 0.1480416466e-2];
162 % >> check = sum(sum(abs(exactsol-q) < ...
      gail.tolfun(1e-1,1e-1,1,exactsol,'max')))
163 % check = 6
164 %
165 %
166 % Example 3:
167 % Sobol' g-function example, first order indice for dimension 5:
168 %
169 % >> a = [0 1/2 3 9 99 99]; hyperbox = [zeros(1,6) ; ones(1,6)];
170 % >> f = @(x) prod(bsxfun(@(x,a) (abs(4*x-2)+a)./(1+a), x , a),2);
171 % >> q = cubSobol_SI_fo_g(f,hyperbox,'uniform',1e-2,1e-1); exactsol ...
      = [.5867811893, .2607916397, 0.3667382378e-1, 0.5867811312e-2, ...
      0.5867753221e-4, 0.5867753221e-4];
172 % >> check = sum(sum(abs(exactsol-q) < ...
      gail.tolfun(1e-2,1e-1,1,exactsol,'max')))
173 % check = 6
174 %
175 %
176 % Example 4:
```

```
177 % Morokoff and Caflish example, total effect indice for dimension 4:
178 %
179 % >> f = @(x) (1+1/6)^6*prod(x,2).^(1/6); hyperbox = [zeros(1,6) ; ...
      ones(1,6)];
180 % >> q = cubSobol_SI_fo_g(f,hyperbox,'uniform',1e-2,1e-1); exactsol ...
      = [.1581948744, .1581948744, .1581948744, .1581948744, ...
      .1581948744, .1581948744];
181 % >> check = sum(sum(abs(exactsol-q) < ...
      gail.tolfun(1e-2,1e-1,1,exactsol,'max')))
182 % check = 6
183 %
184 %
185 %   See also CUBSOBOL_G
186 %
187 %  References
188 %
189 %   [1] Fred J. Hickernell and Lluis Antoni Jimenez Rugama, ...
      "Reliable adaptive
190 %   cubature using digital sequences," 2014. Submitted for publication:
191 %   arXiv:1410.8615.
192 %
193 %   [2] Art B. Owen, "Better Estimation of Small Sobol' Sensitivity
194 %   Indices," ACM Trans. Model. Comput. Simul., 23, 2, Article 11 ...
      (May 2013).
195 %
196 %   [3] Sou-Cheng T. Choi, Fred J. Hickernell, Yuhan Ding, Lan Jiang,
197 %   Lluis Antoni Jimenez Rugama, Xin Tong, Yizhi Zhang and Xuan Zhou,
198 %   GAIL: Guaranteed Automatic Integration Library (Version 2.1)
199 %   [MATLAB Software], 2015. Available from ...
      http://code.google.com/p/gail/
200 %
201 %   [4] Sou-Cheng T. Choi, "MINRES-QLP Pack and Reliable Reproducible
202 %   Research via Supportable Scientific Software," Journal of Open ...
      Research
203 %   Software, Volume 2, Number 1, e22, pp. 1-7, 2014.
204 %
205 %   [5] Sou-Cheng T. Choi and Fred J. Hickernell, "IIT MATH-573 Reliable
206 %   Mathematical Software" [Course Slides], Illinois Institute of
207 %   Technology, Chicago, IL, 2013. Available from
208 %   http://code.google.com/p/gail/
209 %
210 %   [6] Daniel S. Katz, Sou-Cheng T. Choi, Hilmar Lapp, Ketan ...
      Maheshwari,
211 %   Frank Loffler, Matthew Turk, Marcus D. Hanwell, Nancy Wilkins-Diehr,
212 %   James Hetherington, James Howison, Shel Swenson, Gabrielle D. Allen,
213 %   Anne C. Elster, Bruce Berriman, Colin Venters, "Summary of the First
214 %   Workshop On Sustainable Software for Science: Practice And ...
      Experiences
215 %   (WSSSPE1)," Journal of Open Research Software, Volume 2, Number ...
      1, e6,
216 %   pp. 1-21, 2014.
217 %
218 %   If you find GAIL helpful in your work, please support us by ...
      citing the
```

```
%   above papers, software, and materials.
%

t_start = tic;
%% Initial important cone factors and Check—initialize parameters
r_lag = 4; %distance between coefficients summed and those computed
[f,hyperbox,out_param] = cubSobol_SI_fo_g_param(r_lag,varargin{:});
converged = false(1,out_param.d); % We flag the indices that converged
l_star = out_param.mmin − r_lag; % Minimum gathering of points for ...
    the sums of DFWT
omg_circ = @(m) 2.^(−m);
omg_hat = @(m) ...
    out_param.fudge(m)/((1+out_param.fudge(r_lag))*omg_circ(r_lag));

if strcmp(out_param.measure,'normal')
   f=@(x) f(gail.stdnorminv(x));
elseif strcmp(out_param.measure,'uniform')
   Cnorm = prod(hyperbox(2,:)−hyperbox(1,:));
   f=@(x) Cnorm*f(bsxfun(@plus,hyperbox(1,:),bsxfun(@times, ...
       (hyperbox(2,:)−hyperbox(1,:)),x))); % a + (b−a)x = u
end

% Below, both numerator and denominator need to be positive. In ...
    addition,
% the final value needs to be between 0 and 1.
numerator_size = 1;
% Sfo_min = @(down, up) estimate_min(down, up); % S function for ...
    first order
% Sfo_max = @(down, up) estimate_max(down, up); % S function for ...
    first order
% ffo1 = @(fx,fy,fxy) fx.*fxy;
% ffo2 = @(fx,fy,fxy) 1/2*(fx + fxy);
Sfo_min = @(down, up) estimate_min([down(1) 0 down(2:end)], [up(1) 0 ...
    up(2:end)]); % S function for first order
Sfo_max = @(down, up) estimate_max([down(1) 0 down(2:end)], [up(1) 0 ...
    up(2:end)]); % S function for first order
ffo = @(fx,fy,fxy) fx.*(fxy − fy);

%% Main algorithm (we added 2xd dimensions for each index and and ...
    additional 2 for mean of f and f^2)
sobstr = sobolset(2*out_param.d); %generate a Sobol' sequence 2*d
sobstr_scr = sobolset(out_param.d); % This will be the first ...
    scrambled sequence...
     % The replicated sequence will have to use the same scrambling
sobstr_scr = scramble(sobstr_scr,'MatousekAffineOwen'); %scramble it
kappanumap_fx2_fx = bsxfun(@times,(1:2^out_param.mmin)', [1 1]); ...
    %initialize map
Stilde_fx2_fx = zeros(out_param.mmax−out_param.mmin+1, 2); ...
    %initialize sum of DFWT terms for fx2 and fx
CStilde_low_fx2_fx = −inf(out_param.mmax−l_star+1, 2); %initialize ...
    various sums of DFWT terms for necessary conditions for fx2 and fx
CStilde_up_fx2_fx = inf(out_param.mmax−l_star+1, 2); %initialize ...
    various sums of DFWT terms for necessary conditions for fx2 and fx
err_bound_int_fx2_fx = inf(out_param.mmax−out_param.mmin+1, 2); ...
```

```
    %initialize error estimates for fx2 and fx
est_int_fx2_fx = zeros(out_param.mmax-out_param.mmin+1, 2); % ...
    Estimate of the mean for fx2 and fx
exit_len = 2;
out_param.exit = false(exit_len, out_param.d); %we start the ...
    algorithm with all warning flags down

for u = 1:out_param.d
    INDICES(u).kappanumap = kappanumap_fx2_fx(:, 1:numerator_size);
    INDICES(u).Stilde = zeros(out_param.mmax-out_param.mmin+1, ...
        numerator_size);
    INDICES(u).CStilde_low = CStilde_low_fx2_fx(:, 1:numerator_size);
    INDICES(u).CStilde_up = CStilde_up_fx2_fx(:, 1:numerator_size);
    INDICES(u).Smin = Sfo_min;
    INDICES(u).Smax = Sfo_max;
    if numerator_size == 1
        INDICES(u).f = {ffo}; % {ffo1, ffo2} {ffo}
    elseif numerator_size == 2
        INDICES(u).f = {ffo1, ffo2}; % {ffo1, ffo2} {ffo}
    else
        error('Check first order Sobol indices, number of functions ...
            in numerator')
    end
    INDICES(u).est_int = est_int_fx2_fx(:, 1:numerator_size); ...
        %initialize mean estimates for each integral in the numerator
    INDICES(u).err_bound_int = err_bound_int_fx2_fx(:, ...
        1:numerator_size); %initialize error estimates for each ...
        integral in the numerator
    INDICES(u).errest = inf(out_param.mmax-out_param.mmin+1); ...
        %initialize error estimates for each indice
end


%% Initial points and FWT

%% In this section we apply the replicated method
% For the replication method, we want to sort dimension u+d with the ...
    same
% order as dimension u. In other words, if dimension u is sorted ...
    according
% to permutation tau and u+d according to sigma, we want to find psi ...
    such
% that sigma(psi) = tau. Therefore, psi = sigma^-1(tau). To find ...
    sigma^-1 we
% use the sorting Matlab function. The original order is sigma, so ...
    we only
% need to call the function values at order psi.
% Note that with scrambled points, to keep the replicated property,
% dimensions u and u+d need to share the same scrambling for all u (for
% each u we can choose a different scrambling). This means that the ...
    sorting
% psi is always the same that can be found with non-scr Sobol points ...
    (S for
% the Scrambling + shift): S(sigma(psi)) = S(tau) -> psi =
```

```
% sigma^-1(S^-1(S(tau))) = sigma^-1(tau). Therefore, the sorting can be
% found with non-scr Sobol points and applied to the scrambled points.
out_param.n = 2^out_param.mmin*ones(1,out_param.d); %total number of ...
    points to start with
n0 = out_param.n(1); %initial number of points
xpts = sobstr(1:n0,1:2*out_param.d); %grab Sobol' points
tau = 2^(out_param.mmin)*xpts(:, 1:out_param.d) + 1;
sigma = 2^(out_param.mmin)*xpts(:, out_param.d+1:2*out_param.d) + 1;

%We create the replicated points
xpts_1 = sobstr_scr(1:n0, 1:out_param.d); xpts_2 = zeros(size(xpts_1));
for u = 1:out_param.d
    [~, tau_inv] = sort(tau(:, u));
    xpts_2(:, u) = xpts_1(tau_inv(sigma(:, u)), u);
end

% We perform some evaluations
fx = f(xpts_1); %evaluate integrands y3
fx2 = fx.^2; %evaluate integrands y2
fxval = fx; % We store fx because fx will later contain the fast ...
    transform
fx2val = fx2; % We store fx2 because fx2 will later contain the fast ...
    transform
% We only need fy if we use the estimator that includes ffo
fy = [];
if numerator_size == 1
    fy = f(xpts_2); % We evaluate the f at the replicated points
end

% We apply the replicated method
fxy_base = f(xpts_2); % In sigma order
for u = 1:out_param.d
    [~, sigma_inv] = sort(sigma(:, u));
    fxy = fxy_base(sigma_inv(tau(:, u)));
    INDICES(u).y = [];
    for p = 1:numerator_size
        INDICES(u).y = [INDICES(u).y INDICES(u).f{p}(fx,fy,fxy)];
    end
    INDICES(u).est_int(1, :) = mean(INDICES(u).y, 1); % Estimate the ...
        integral
end


%% Compute initial FWT
nllstart = int64(2^(out_param.mmin-r_lag-1));
for l=0:out_param.mmin-1 % We need the FWT for fx, fx^2, and the y ...
    values (a1/[a2-a3])
    nl=2^l;
    nmminlm1=2^(out_param.mmin-l-1);
    ptind=repmat([true(nl,1); false(nl,1)],nmminlm1,1);
    for u = 1:out_param.d
        evenval=INDICES(u).y(ptind, :);
        oddval=INDICES(u).y(~ptind, :);
        INDICES(u).y(ptind, :)=(evenval+oddval)/2;
```

```
347          INDICES(u).y(¬ptind, :)=(evenval−oddval)/2;
348      end
349      evenval=fx(ptind);
350      oddval=fx(¬ptind);
351      fx(ptind)=(evenval+oddval)/2;
352      fx(¬ptind)=(evenval−oddval)/2;
353      evenval=fx2(ptind);
354      oddval=fx2(¬ptind);
355      fx2(ptind)=(evenval+oddval)/2;
356      fx2(¬ptind)=(evenval−oddval)/2;
357  end
358  %y now contains the FWT coefficients
359
360  for u = 1:out_param.d + 1 % u == d+1 is the kappanumap for fx2 and fx
361      %% Create kappanumap implicitly from the data
362      for l=out_param.mmin−1:−1:1
363         nl=2^l;
364         % for u = d+1 we compute the denominator which is the same for
365         % all indices. That is why we store the y values and kappanumap
366         % separetly for the denominator
367         if u < out_param.d + 1
368             for p = 1:numerator_size
369                  oldone=abs(INDICES(u).y(INDICES(u). ...
370                      kappanumap(2:nl, p), p)); %earlier values of ...
                           kappa, don't touch first one
371                  newone=abs(INDICES(u).y(INDICES(u). ...
372                      kappanumap(nl+2:2*nl, p), p)); %later values of ...
                           kappa
373                  flip=find(newone>oldone); %which in the pair are the ...
                        larger ones
374                  if ¬isempty(flip)
375                     flipall=bsxfun(@plus,flip, ...
376                         0:2^(l+1):2^out_param.mmin−1);
377                     temp = INDICES(u).kappanumap(nl+1+flipall, p); ...
                           %then flip
378                     INDICES(u).kappanumap(nl+1+flipall, p) = ...
                           INDICES(u). ...
379                         kappanumap(1+flipall, p); %them
380                     INDICES(u).kappanumap(1+flipall, p) = temp; %around
381                  end
382             end
383         else % We keep the kappamap for int fx2 and fx
384             oldone=abs(fx2(kappanumap_fx2_fx(2:nl,1))); %earlier ...
                    values of kappa, don't touch first one
385             newone=abs(fx2(kappanumap_fx2_fx(nl+2:2*nl,1))); %later ...
                    values of kappa
386             flip=find(newone>oldone); %which in the pair are the ...
                    larger ones
387             if ¬isempty(flip)
388                 flipall=bsxfun(@plus,flip, ...
389                     0:2^(l+1):2^out_param.mmin−1);
390                 temp = kappanumap_fx2_fx(nl+1+flipall,1); %then flip
391                 kappanumap_fx2_fx(nl+1+flipall,1) = ...
                        kappanumap_fx2_fx(1+flipall,1); %them
```

```
392                 kappanumap_fx2_fx(1+flipall,1) = temp; %around
393             end
394             oldone=abs(fx(kappanumap_fx2_fx(2:nl,2))); %earlier ...
                    values of kappa, don't touch first one
395             newone=abs(fx(kappanumap_fx2_fx(nl+2:2*nl,2))); %later ...
                    values of kappa
396             flip=find(newone>oldone); %which in the pair are the ...
                    larger ones
397             if ¬isempty(flip)
398                 flipall=bsxfun(@plus,flip, ...
399                     0:2^(l+1):2^out_param.mmin-1);
400                 temp=kappanumap_fx2_fx(nl+1+flipall,2); %then flip
401                 kappanumap_fx2_fx(nl+1+flipall,2) = ...
                        kappanumap_fx2_fx(1+flipall,2); %them
402                 kappanumap_fx2_fx(1+flipall,2) = temp; %around
403             end
404         end
405     end
406 end
407
408 %% Compute Stilde
409 % We keep the error estimates for int fx2
410 Stilde_fx2_fx(1,1) = ...
        sum(abs(fx2(kappanumap_fx2_fx(nllstart+1:2*nllstart,1))));
411 est_int_fx2_fx(1,1,end) = mean(fx2val); % Estimate the integral
412 err_bound_int_fx2_fx(1,1) = ...
        out_param.fudge(out_param.mmin)*Stilde_fx2_fx(1,1);
413 % We keep the error estimates for int fx
414 Stilde_fx2_fx(1,2) = ...
        sum(abs(fx(kappanumap_fx2_fx(nllstart+1:2*nllstart,2))));
415 est_int_fx2_fx(1,2) = mean(fxval); % Estimate the integral
416 err_bound_int_fx2_fx(1,2) = ...
        out_param.fudge(out_param.mmin)*Stilde_fx2_fx(1,2);
417 int = est_int_fx2_fx(1,2); % Estimate of the expectation of the function
418
419 for u = 1:out_param.d
420     INDICES(u).Stilde(1, :) = sum(abs(INDICES(u).y(INDICES(u). ...
421         kappanumap(nllstart+1:2*nllstart, :))), 1);
422     INDICES(u).err_bound_int(1, :) = ...
            out_param.fudge(out_param.mmin)*INDICES(u).Stilde(1, :);
423
424     down = [INDICES(u).est_int(1, :) - INDICES(u).err_bound_int(1, ...
            :), est_int_fx2_fx(1, :) - err_bound_int_fx2_fx(1, :)];
425     up = [INDICES(u).est_int(1, :) + INDICES(u).err_bound_int(1, :), ...
            est_int_fx2_fx(1, :) + err_bound_int_fx2_fx(1, :)];
426     q(u) = 1/2*(INDICES(u).Smax(down,up) + INDICES(u).Smin(down,up));
427     out_param.bound_err(u) = 1/2*(INDICES(u).Smax(down,up) - ...
            INDICES(u).Smin(down,up));
428     INDICES(u).errest(1) = out_param.bound_err(u);
429 end
430
431 % Necessary conditions for all indices integrals and fx and fx2
432 for l = l_star:out_param.mmin % Storing the information for the ...
        necessary conditions
```

```
433     C_low = 1/(1+omg_hat(out_param.mmin-l)*omg_circ(out_param.mmin-l));
434     C_up = 1/(1-omg_hat(out_param.mmin-l)*omg_circ(out_param.mmin-l));
435     for u = 1:out_param.d
436         INDICES(u).CStilde_low(l-l_star+1, :)=max( ...
437             INDICES(u).CStilde_low(l-l_star+1, :),C_low*sum(abs( ...
438             INDICES(u).y(INDICES(u).kappanumap(2^(l-1)+1:2^l, :))), 1));
439         if (omg_hat(out_param.mmin-l)*omg_circ(out_param.mmin-l) < 1)
440             INDICES(u).CStilde_up(l-l_star+1, :)=min( ...
441                 INDICES(u).CStilde_up(l-l_star+1, :),C_up*sum(abs( ...
442                 INDICES(u).y(INDICES(u).kappanumap(2^(l-1)+1:2^l, ...
                        :))), 1));
443         end
444     end
445     CStilde_low_fx2_fx(l-l_star+1,1) = ...
            max(CStilde_low_fx2_fx(l-l_star+ ...
446         1,1),C_low*sum(abs(fx2(kappanumap_fx2_fx(2^(l-1)+1:2^l, ...
447         1)))));
448     CStilde_low_fx2_fx(l-l_star+1,2) = ...
            max(CStilde_low_fx2_fx(l-l_star+ ...
449         1,2),C_low*sum(abs(fx(kappanumap_fx2_fx(2^(l-1)+1:2^l, ...
450         2)))));
451     if (omg_hat(out_param.mmin-l)*omg_circ(out_param.mmin-l) < 1)
452         CStilde_up_fx2_fx(l-l_star+1,1) = ...
                min(CStilde_up_fx2_fx(l-l_star+ ...
453             1,1),C_up*sum(abs(fx2(kappanumap_fx2_fx(2^(l-1)+1:2^l, ...
454             1)))));
455         CStilde_up_fx2_fx(l-l_star+1,2) = ...
                min(CStilde_up_fx2_fx(l-l_star+ ...
456             1,2),C_up*sum(abs(fx(kappanumap_fx2_fx(2^(l-1)+1:2^l, ...
457             2)))));
458     end
459 end
460 aux_bool = any(any(CStilde_low_fx2_fx > CStilde_up_fx2_fx)); % ...
        Variable that checks conditions violated for fx2 and fx
461 for u = 1:out_param.d
462     if any(any(INDICES(u).CStilde_low > INDICES(u).CStilde_up)) || ...
            aux_bool
463         out_param.exit(2,u) = true;
464     end
465 end
466
467 for u = 1:out_param.d
468     % Check the end of the algorithm
469     aplus = 0.5*(gail.tolfun(out_param.abstol,out_param.reltol, ...
470         out_param.theta,abs(q(u)-INDICES(u).errest(1)),out_param. ...
471         toltype)+gail.tolfun(out_param.abstol,out_param.reltol, ...
472         out_param.theta,abs(q(u)+INDICES(u).errest(1)), ...
473         out_param.toltype));
474     aminus = 0.5*(gail.tolfun(out_param.abstol,out_param.reltol, ...
475         out_param.theta,abs(q(u)-INDICES(u).errest(1)),out_param. ...
476         toltype)-gail.tolfun(out_param.abstol,out_param.reltol, ...
477         out_param.theta,abs(q(u)+INDICES(u).errest(1)), ...
478         out_param.toltype));
479
```

```
480     q(u) = q(u)+Δminus;
481     if out_param.bound_err(u) ≤ Δplus
482        converged(u) = true;
483     elseif out_param.mmin == out_param.mmax % We are on our max ...
           budget and did not meet the error condition => overbudget
484        out_param.exit(1,u) = true;
485     end
486 end
487 out_param.time=toc(t_start);
488
489 %% Loop over m
490 for m = out_param.mmin+1:out_param.mmax
491     if all(converged)
492        break;
493     end
494     mnext=m-1;
495     nnext=2^mnext;
496     xnext=sobstr(n0+(1:nnext),1:2*out_param.d); % The following Sobol
497                     % points have the nice property that can give us the
498                     % order doing this operation. The +1 is because in
499                     % Matlab, vectors start at 1, not 0.
500     tau = (2^m*xnext(:, 1:out_param.d) - 1)/2 + 1;
501     sigma = (2^m*xnext(:, out_param.d+1:2*out_param.d) - 1)/2 + 1;
502     %We create the replicated points
503     xnext_1 = sobstr_scr(n0+(1:nnext),1:out_param.d); xnext_2 = ...
           zeros(size(xnext_1));
504     for u = 1:out_param.d
505         [¬, tau_inv] = sort(tau(:, u));
506         xnext_2(:, u) = xnext_1(tau_inv(sigma(:, u)), u);
507     end
508
509     n0=n0+nnext;
510     fxnext = f(xnext_1); %evaluate integrands y3
511     if numerator_size == 1
512         fy = f(xnext_2); % We evaluate the f at the replicated points
513     end
514     fx2next = fxnext.^2; %evaluate integrands y2
515     fxval = [fxval; fxnext];
516     fx2val = [fx2val; fx2next];
517
518     %% Compute initial FWT on next points only for fx and fx2
519     % We need to split fx2 and fx from the indices because the ...
           number of
520     % data points used will vary depending on the indice but we will ...
           always
521     % still need fx and fx2. Keeping altogehter requires too many if's.
522     nllstart=int64(2^(m-r_lag-1));
523     meff=m-out_param.mmin+1;
524     for l=0:mnext-1
525         nl=2^l;
526         nmminlm1=2^(mnext-l-1);
527         ptind=repmat([true(nl,1); false(nl,1)],nmminlm1,1);
528         evenval=fxnext(ptind);
529         oddval=fxnext(¬ptind);
```

```
530         fxnext(ptind)=(evenval+oddval)/2;
531         fxnext(¬ptind)=(evenval−oddval)/2;
532         evenval=fx2next(ptind);
533         oddval=fx2next(¬ptind);
534         fx2next(ptind)=(evenval+oddval)/2;
535         fx2next(¬ptind)=(evenval−oddval)/2;
536     end
537     %% Compute FWT on all points only for fx and fx2
538     fx = [fx; fxnext];
539     fx2 = [fx2; fx2next];
540     nl=2^mnext;
541     ptind=[true(nl,1); false(nl,1)];
542     evenval=fx(ptind);
543     oddval=fx(¬ptind);
544     fx(ptind)=(evenval+oddval)/2;
545     fx(¬ptind)=(evenval−oddval)/2;
546     evenval=fx2(ptind);
547     oddval=fx2(¬ptind);
548     fx2(ptind)=(evenval+oddval)/2;
549     fx2(¬ptind)=(evenval−oddval)/2;
550     %% Update kappanumap only for fx and fx2
551     kappanumap_fx2_fx = [kappanumap_fx2_fx ; 2^(m−1) + ...
            kappanumap_fx2_fx]; %initialize map only for fx and fx2
552     for l=m−1:−1:m−r_lag
553         nl=2^l;
554         oldone=abs(fx2(kappanumap_fx2_fx(2:nl,1))); %earlier values ...
                of kappa, don't touch first one
555         newone=abs(fx2(kappanumap_fx2_fx(nl+2:2*nl,1))); %later ...
                values of kappa
556         flip=find(newone>oldone);
557         if ¬isempty(flip)
558           flipall=bsxfun(@plus,flip,0:2^(l+1):2^m−1);
559           temp=kappanumap_fx2_fx(nl+1+flipall,1);
560           kappanumap_fx2_fx(nl+1+flipall,1)= ...
561               kappanumap_fx2_fx(1+flipall,1);
562           kappanumap_fx2_fx(1+flipall,1)=temp;
563         end
564         oldone=abs(fx(kappanumap_fx2_fx(2:nl,2))); %earlier values ...
                of kappa, don't touch first one
565         newone=abs(fx(kappanumap_fx2_fx(nl+2:2*nl,2))); %later ...
                values of kappa
566         flip=find(newone>oldone);
567         if ¬isempty(flip)
568           flipall=bsxfun(@plus,flip,0:2^(l+1):2^m−1);
569           temp=kappanumap_fx2_fx(nl+1+flipall,2);
570           kappanumap_fx2_fx(nl+1+flipall,2)= ...
571               kappanumap_fx2_fx(1+flipall,2);
572           kappanumap_fx2_fx(1+flipall,2)=temp;
573         end
574     end
575
576     %% Compute Stilde for fx and fx2 only
577     % We keep the error estimates and integrals only for int fx2
578     Stilde_fx2_fx(meff,1) = ...
```

```
        sum(abs(fx2(kappanumap_fx2_fx(nllstart+1:2*nllstart,1))));
579     est_int_fx2_fx(meff,1) = mean(fx2val); % Estimate the integral ...
            of f^2
580     err_bound_int_fx2_fx(meff,1) = ...
            out_param.fudge(m)*Stilde_fx2_fx(meff,1);
581     % We keep the error estimates and integrals only for int fx
582     Stilde_fx2_fx(meff,2) = ...
            sum(abs(fx(kappanumap_fx2_fx(nllstart+1:2*nllstart,2))));
583     est_int_fx2_fx(meff,2) = mean(fxval); % Estimate the integral of f
584     err_bound_int_fx2_fx(meff,2) = ...
            out_param.fudge(m)*Stilde_fx2_fx(meff,2);
585     int = est_int_fx2_fx(meff,2); % Estimate of the expectation of ...
            the function
586
587     %% We start computing everything for the numerator. fx and fx2 ...
            were for
588     % the denominator only, which is common with all indices
589     fxy_base = f(xnext_2); % In sigma order,
590         % and at this point, we now that at least 1 indice did not ...
              meet the
591         % error tolerance so we need to evaluate fxy_base. However, the
592         % sorting, FFWT, etc, only needs to be done for those that ...
              did not
593         % converge.
594     for u = 1:out_param.d
595             if ¬converged(u)
596                 INDICES(u).n = 2^m;
597                 out_param.n(u) = 2^m;
598                 [¬, sigma_inv] = sort(sigma(:, u));
599                 fxy = fxy_base(sigma_inv(tau(:, u)));
600                 ynext = [];
601                 % fxval already was computed for fx and fx2 above, ...
                      so fx
602                 % contains FFWT coefficients of fx.
603                 for p = 1:numerator_size
604                     ynext = [ynext INDICES(u).f{p}(fxval(end/2 + ...
                          1:end),fy,fxy)];
605                 end
606                 INDICES(u).est_int(meff, :) = ...
                        1/2*(INDICES(u).est_int(meff-1, :) + mean(ynext, ...
                        1)); % Estimate the integral
607
608                 %% Compute initial FWT on next points only for y
609                 nllstart=int64(2^(m-r_lag-1));
610                 meff=m-out_param.mmin+1;
611                  for l=0:mnext-1
612                     nl=2^l;
613                     nmminlm1=2^(mnext-l-1);
614                     ptind=repmat([true(nl,1); false(nl,1)],nmminlm1,1);
615                     evenval=ynext(ptind, :);
616                     oddval=ynext(¬ptind, :);
617                     ynext(ptind, :)=(evenval+oddval)/2;
618                     ynext(¬ptind, :)=(evenval-oddval)/2;
619                  end
```

```
620                 %% Compute FWT on all points only for y
621                 INDICES(u).y = [INDICES(u).y; ynext];
622                 nl=2^mnext;
623                 ptind=[true(nl,1); false(nl,1)];
624                 evenval = INDICES(u).y(ptind, :);
625                 oddval = INDICES(u).y(¬ptind, :);
626                 INDICES(u).y(ptind, :) = (evenval+oddval)/2;
627                 INDICES(u).y(¬ptind, :) = (evenval-oddval)/2;
628
629                 %% Update kappanumap only for indices
630                 INDICES(u).kappanumap = [INDICES(u).kappanumap ; ...
                        2^(m-1)+INDICES(u).kappanumap];
631                 for l=m-1:-1:m-r_lag
632                   nl=2^l;
633                   for p = 1:numerator_size
634                       oldone=abs(INDICES(u).y(INDICES(u). ...
635                           kappanumap(2:nl, p), p)); %earlier values ...
                                of kappa, don't touch first one
636                       newone=abs(INDICES(u).y(INDICES(u). ...
637                           kappanumap(nl+2:2*nl, p), p)); %later ...
                                values of kappa
638                       flip = find(newone>oldone);
639                       if ¬isempty(flip)
640                           flipall = bsxfun(@plus,flip,0:2^(l+1):2^m-1);
641                           temp = INDICES(u).kappanumap(nl+1+flipall, p);
642                           INDICES(u).kappanumap(nl+1+flipall, p) = ...
                                INDICES(u).kappanumap(1+flipall, p);
643                           INDICES(u).kappanumap(1+flipall, p) = temp;
644                       end
645                   end
646                 end
647
648                 %% Compute Stilde for y only
649                 INDICES(u).Stilde(meff, :) = sum(abs(INDICES(u). ...
650                     y(INDICES(u).kappanumap(nllstart+1:2*nllstart, ...
                            :))), 1);
651                 INDICES(u).err_bound_int(meff, :) = ...
                        out_param.fudge(m)*INDICES(u).Stilde(meff, :); % ...
                        Only error bound for the integral on the numerator
652
653                 down = [INDICES(u).est_int(meff, :) - ...
                        INDICES(u).err_bound_int(meff, :), ...
                        est_int_fx2_fx(meff, :) - ...
                        err_bound_int_fx2_fx(meff, :)];
654                 up = [INDICES(u).est_int(meff, :) + ...
                        INDICES(u).err_bound_int(meff, :), ...
                        est_int_fx2_fx(meff, :) + ...
                        err_bound_int_fx2_fx(meff, :)];
655
656                 q(u) = 1/2*(INDICES(u).Smax(down,up) + ...
                        INDICES(u).Smin(down,up));
657                 out_param.bound_err(u) = ...
                        1/2*(INDICES(u).Smax(down,up) - ...
                        INDICES(u).Smin(down,up));
```

```
658                 INDICES(u).errest(meff) = out_param.bound_err(u);
659             end
660         end
661
662         % Necessary conditions for all indices integrals and fx and fx2
663         for l = l_star:m % Storing the information for the necessary ...
                conditions
664             C_low = 1/(1+omg_hat(out_param.mmin-l)* ...
665                 omg_circ(out_param.mmin-l));
666             C_up = 1/(1-omg_hat(out_param.mmin-l)* ...
667                 omg_circ(out_param.mmin-l));
668             for u = 1:out_param.d
669                 if ~converged(u)
670                     INDICES(u).CStilde_low(l-l_star+1, :) = ...
                            max(INDICES(u). ...
671                         CStilde_low(l-l_star+1, ...
                                :),C_low*sum(abs(INDICES(u). ...
672                         y(INDICES(u).kappanumap(2^(l-1)+1:2^l, :))), 1));
673                     if (omg_hat(out_param.mmin-l)* ...
674                             omg_circ(out_param.mmin-l) < 1)
675                         INDICES(u).CStilde_up(l-l_star+1, :) = min( ...
676                             INDICES(u).CStilde_up(l-l_star+1, ...
                                    :),C_up*sum( ...
677                             abs(INDICES(u).y(INDICES(u).kappanumap( ...
678                             2^(l-1)+1:2^l, :))), 1));
679                     end
680                 end
681             end
682             CStilde_low_fx2_fx(l-l_star+1,1) = max(CStilde_low_fx2_fx(l- ...
683                 l_star+1,1),C_low*sum(abs(fx2(kappanumap_fx2_fx( ...
684                 2^(l-1)+1:2^l,1)))));
685             CStilde_low_fx2_fx(l-l_star+1,2) = max(CStilde_low_fx2_fx(l- ...
686                 l_star+1,2),C_low*sum(abs(fx(kappanumap_fx2_fx( ...
687                 2^(l-1)+1:2^l,2)))));
688             if (omg_hat(out_param.mmin-l)*omg_circ(out_param.mmin-l) < 1)
689                 CStilde_up_fx2_fx(l-l_star+1,1) = ...
                        min(CStilde_up_fx2_fx(l- ...
690                         l_star+1,1),C_up*sum(abs(fx2(kappanumap_fx2_fx( ...
691                         2^(l-1)+1:2^l,1)))));
692                 CStilde_up_fx2_fx(l-l_star+1,2) = ...
                        min(CStilde_up_fx2_fx(l- ...
693                         l_star+1,2),C_up*sum(abs(fx(kappanumap_fx2_fx(2^(l ...
694                         -1)+1:2^l,2)))));
695             end
696         end
697         aux_bool = any(any(CStilde_low_fx2_fx > CStilde_up_fx2_fx)); % ...
                Variable that checks conditions violated for fx2 and fx
698         for u = 1:out_param.d
699             if ~converged(u) & (any(any(INDICES(u).CStilde_low > ...
                    INDICES(u).CStilde_up)) || aux_bool)
700                 out_param.exit(2,u) = true;
701             end
702         end
703
```

```
704     for u = 1:out_param.d
705             if ¬converged(u)
706                 % Check the end of the algorithm
707                 Δplus = 0.5*(gail.tolfun(out_param.abstol,...
708                     out_param.reltol,out_param.theta,abs(q(u)− ...
709                     INDICES(u).errest(meff)),out_param.toltype)+ ...
710                     gail.tolfun(out_param.abstol,out_param. ...
711                     reltol,out_param.theta,abs(q(u)+INDICES(u). ...
712                     errest(meff)), out_param.toltype));
713                 Δminus = 0.5*(gail.tolfun(out_param.abstol, ...
714                     out_param.reltol,out_param.theta,abs(q(u)− ...
715                     INDICES(u).errest(meff)),out_param.toltype)− ...
716                     gail.tolfun(out_param.abstol,out_param. ...
717                     reltol,out_param.theta,abs(q(u)+INDICES(u). ...
718                     errest(meff)),out_param.toltype));
719
720                 q(u) = q(u) + Δminus;
721                 if out_param.bound_err(u) ≤ Δplus
722                    converged(u) = true;
723                 elseif m == out_param.mmax % We are on our max ...
                       budget and did not meet the error condition => ...
                       overbudget
724                    out_param.exit(1,u) = true;
725                 end
726             end
727     end
728
729     out_param.time=toc(t_start);
730 end
731
732 % Decode the exit structure
733 for u = 1:out_param.d
734         out_param.exitflag(:,u) = ...
               (2.^(0:exit_len−1))'.*out_param.exit(:,u);
735 end
736 out_param = rmfield(out_param,'exit');
737
738 out_param.time=toc(t_start);
739 end
740
741
742 %% Parsing for the input of cubSobol_SI_fo_g
743 function [f,hyperbox, out_param] = ...
       cubSobol_SI_fo_g_param(r_lag,varargin)
744
745 % Default parameter values
746 default.hyperbox = [zeros(1,1);ones(1,1)];% default hyperbox
747 default.measure  = 'uniform';
748 default.abstol  = 1e−4;
749 default.reltol  = 1e−2;
750 default.mmin  = 10;
751 default.mmax  = 24;
752 default.fudge = @(m) 5*2.^−m;
753 default.toltype  = 'max';
```

```
754 default.theta  = 1;
755
756 if numel(varargin)<2
757     help cubSobol_SI_fo_g
758     warning('GAIL:cubSobol_SI_fo_g:fdnotgiven',...
759         'At least, function f and hyperbox need to be specified. ...
               Example for f(x)=x^2:')
760     f = @(x) x.^2;
761     out_param.f=f;
762     hyperbox = default.hyperbox;
763 else
764     f = varargin{1};
765     if ¬gail.isfcn(f)
766         warning('GAIL:cubSobol_SI_fo_g:fnotfcn',...
767             'The given input f was not a function. Example for ...
                   f(x)=x^2:')
768         f = @(x) x.^2;
769         out_param.f=f;
770         hyperbox = default.hyperbox;
771     else
772         out_param.f=f;
773         hyperbox = varargin{2};
774         if ¬isnumeric(hyperbox) || ¬(size(hyperbox,1)==2) || ¬...
               (size(hyperbox,2)<370)
775             warning('GAIL:cubSobol_SI_fo_g:hyperbox_error1',...
776                 'The hyperbox must be a real matrix of size 2xd ...
                       where d can not be greater than 370. Example for ...
                       f(x)=x^2:')
777             f = @(x) x.^2;
778             out_param.f=f;
779             hyperbox = default.hyperbox;
780         end
781     end
782 end
783
784 validvarargin=numel(varargin)>2;
785 if validvarargin
786     in3=varargin(3:end);
787     for j=1:numel(varargin)-2
788     validvarargin=validvarargin && (isnumeric(in3{j}) ...
789         || ischar(in3{j}) || isstruct(in3{j}) || gail.isfcn(in3{j}));
790     end
791     if ¬validvarargin
792         warning('GAIL:cubSobol_SI_fo_g:validvarargin','Optional ...
               parameters must be numeric or strings. We will use the ...
               default optional parameters.')
793     end
794     in3=varargin{3};
795 end
796
797 MATLABVERSION = gail.matlab_version;
798 if MATLABVERSION ≥ 8.3
799   f_addParamVal = @addParameter;
800 else
```

```
801   f_addParamVal = @addParamValue;
802 end
803
804 if ¬validvarargin
805     out_param.measure = default.measure;
806     out_param.abstol = default.abstol;
807     out_param.reltol = default.reltol;
808     out_param.mmin = default.mmin;
809     out_param.mmax = default.mmax;
810     out_param.fudge = default.fudge;
811     out_param.toltype = default.toltype;
812     out_param.theta = default.theta;
813 else
814     p = inputParser;
815     addRequired(p,'f',@gail.isfcn);
816     addRequired(p,'hyperbox',@isnumeric);
817     if isnumeric(in3) || ischar(in3)
818         addOptional(p,'measure',default.measure,...
819             @(x) any(validatestring(x, {'uniform','normal'})));
820         addOptional(p,'abstol',default.abstol,@isnumeric);
821         addOptional(p,'reltol',default.reltol,@isnumeric);
822         addOptional(p,'mmin',default.mmin,@isnumeric);
823         addOptional(p,'mmax',default.mmax,@isnumeric);
824         addOptional(p,'fudge',default.fudge,@gail.isfcn);
825         addOptional(p,'toltype',default.toltype,...
826             @(x) any(validatestring(x, {'max','comb'})));
827         addOptional(p,'theta',default.theta,@isnumeric);
828     else
829         if isstruct(in3) %parse input structure
830             p.StructExpand = true;
831             p.KeepUnmatched = true;
832         end
833         f_addParamVal(p,'measure',default.measure,...
834             @(x) any(validatestring(x, {'uniform','normal'})));
835         f_addParamVal(p,'abstol',default.abstol,@isnumeric);
836         f_addParamVal(p,'reltol',default.reltol,@isnumeric);
837         f_addParamVal(p,'mmin',default.mmin,@isnumeric);
838         f_addParamVal(p,'mmax',default.mmax,@isnumeric);
839         f_addParamVal(p,'fudge',default.fudge,@gail.isfcn);
840         f_addParamVal(p,'toltype',default.toltype,...
841             @(x) any(validatestring(x, {'max','comb'})));
842         f_addParamVal(p,'theta',default.theta,@isnumeric);
843     end
844     parse(p,f,hyperbox,varargin{3:end})
845     out_param = p.Results;
846 end
847
848 out_param.d = size(hyperbox,2);
849
850 fdgyes = 0; % We store how many functions are in varargin. There can ...
       only
851             % two functions as input, the function f and the fudge ...
                  factor.
852 for j = 1:size(varargin,2)
```

```
853     fdgyes = gail.isfcn(varargin{j})+fdgyes;
854 end
855 if fdgyes < 2 % No fudge factor given as input
856     default.fudge = @(m) 5*2.^-(m/d);
857 end
858
859 %hyperbox should be 2 x dimension
860 if ¬isnumeric(hyperbox) || ¬(size(hyperbox,1)==2) || ¬(out_param.d<370)
861     warning('GAIL:cubSobol_SI_fo_g:hyperbox_error2',...
862         'The hyperbox must be a real matrix of size 2 x d where d ...
                can not be greater than 370. Example for f(x)=x^2:')
863     f = @(x) x.^2;
864     out_param.f=f;
865     hyperbox = default.hyperbox;
866 end
867
868 % Force measure to be uniform or normal only
869 if ¬(strcmp(out_param.measure,'uniform') || ...
    strcmp(out_param.measure,'normal') )
870     warning('GAIL:cubSobol_SI_fo_g:notmeasure',['The measure can ...
        only be uniform or normal.' ...
871             ' Using default measure ' num2str(default.measure)])
872     out_param.measure = default.measure;
873 end
874
875 % Force absolute tolerance greater than 0
876 if (out_param.abstol < 0 )
877     warning('GAIL:cubSobol_SI_fo_g:abstolnonpos',['Absolute ...
        tolerance cannot be negative.' ...
878             ' Using default absolute tolerance ' ...
                num2str(default.abstol)])
879     out_param.abstol = default.abstol;
880 end
881
882 % Force relative tolerance greater than 0 and smaller than 1
883 if (out_param.reltol < 0) || (out_param.reltol > 1)
884     warning('GAIL:cubSobol_SI_fo_g:reltolnonunit',['Relative ...
        tolerance should be chosen in [0,1].' ...
885             ' Using default relative tolerance ' ...
                num2str(default.reltol)])
886     out_param.reltol = default.reltol;
887 end
888
889 % Force mmin to be integer greater than 0
890 if (¬gail.isposint(out_param.mmin) || ¬(out_param.mmin < ...
    out_param.mmax+1))
891     warning('GAIL:cubSobol_SI_fo_g:lowmmin',['The minimum starting ...
        exponent ' ...
892             'should be an integer greater than 0 and smaller or ...
                equal than the maxium.' ...
893             ' Using default mmin ' num2str(default.mmin)])
894     out_param.mmin = default.mmin;
895 end
896
```

```
897 % Force mmin to be integer greater than r_lag (so that ...
      l_star=mmin-r_lag>0)
898 if out_param.mmin < r_lag
899     warning('GAIL:cubSobol_SI_fo_g:lowmminrlag',['The minimum ...
          starting exponent ' ...
900              'should be at least ' num2str(r_lag) '.' ...
901              ' Using default mmin ' num2str(default.mmin)])
902     out_param.mmin = default.mmin;
903 end
904
905 % Force exponent budget number of points be a positive integer ...
      greater than
906 % or equal to mmin an smaller than 54
907 if ¬(gail.isposint(out_param.mmax) && out_param.mmax≥out_param.mmin ...
      && out_param.mmax≤53)
908     warning('GAIL:cubSobol_SI_fo_g:wrongmmax',['The maximum exponent ...
          for the budget should be an integer biger than mmin and ...
          smaller than 54.' ...
909              ' Using default mmax ' num2str(default.mmax)])
910     out_param.mmax = default.mmax;
911 end
912
913 % Force fudge factor to be greater than 0
914 if ¬((gail.isfcn(out_param.fudge) && (out_param.fudge(1)>0)))
915     warning('GAIL:cubSobol_SI_fo_g:fudgenonpos',['The fudge factor ...
          should be a positive function.' ...
916              ' Using default fudge factor ' func2str(default.fudge)])
917     out_param.fudge = default.fudge;
918 end
919
920 % Force toltype to be max or comb
921 if ¬(strcmp(out_param.toltype,'max') || ...
      strcmp(out_param.toltype,'comb') )
922     warning('GAIL:cubSobol_SI_fo_g:nottoltype',['The error type can ...
          only be max or comb.' ...
923              ' Using default toltype ' num2str(default.toltype)])
924     out_param.toltype = default.toltype;
925 end
926
927 % Force theta to be in [0,1]
928 if (out_param.theta < 0) || (out_param.theta > 1)
929     warning('GAIL:cubSobol_SI_fo_g:thetanonunit',['Theta should be ...
          chosen in [0,1].' ...
930              ' Using default theta ' num2str(default.theta)])
931     out_param.theta = default.theta;
932 end
933
934 % Checking on pure absolute/relative error
935 if (out_param.abstol==0) && (out_param.reltol==0)
936     warning('GAIL:cubSobol_SI_fo_g:tolzeros',['Absolute and relative ...
          error tolerances can not be simultaniusly 0.' ...
937              ' Using default absolute tolerance ' ...
                   num2str(default.abstol) ' and relative tolerance ' ...
                   num2str(default.reltol)])
```

```
938     out_param.abstol = default.abstol;
939     out_param.reltol = default.reltol;
940 end
941 if (strcmp(out_param.toltype,'comb')) && (out_param.theta==1) && ...
    (out_param.abstol==0)
942     warning('GAIL:cubSobol_SI_fo_g:abstolzero',['When choosing ...
        toltype comb, if theta=1 then abstol>0.' ...
943             ' Using default absolute tolerance ' ...
                num2str(default.abstol) ])
944     out_param.abstol = default.abstol;
945 end
946 if (strcmp(out_param.toltype,'comb')) && (out_param.theta==0) && ...
    (out_param.reltol==0)
947     warning('GAIL:cubSobol_SI_fo_g:reltolzero',['When choosing ...
        toltype comb, if theta=0 then reltol>0.' ...
948             ' Using default relative tolerance ' ...
                num2str(default.reltol) ])
949     out_param.reltol = default.reltol;
950 end
951
952 % Checking on the hyperbox given the measure
953 if (strcmp(out_param.measure,'uniform')) && ¬...
    all(all(isfinite(hyperbox)))
954     warning('GAIL:cubSobol_SI_fo_g:hyperboxnotfinite',['If uniform ...
        measure, hyperbox must be of finite volume.' ...
955             ' Using default hyperbox:'])
956     disp([zeros(1,out_param.d);ones(1,out_param.d)])
957     hyperbox = [zeros(1,out_param.d);ones(1,out_param.d)];
958 end
959 if (strcmp(out_param.measure,'normal')) && ...
    (sum(sum(isfinite(hyperbox)))>0)
960     warning('GAIL:cubSobol_SI_fo_g:hyperboxfinite',['If normal ...
        measure, hyperbox must be defined as (-Inf,Inf)^d.' ...
961             ' Using default hyperbox:'])
962     disp([-inf*ones(1,out_param.d);inf*ones(1,out_param.d)])
963     hyperbox = [-inf*ones(1,out_param.d);inf*ones(1,out_param.d)];
964 end
965 if (strcmp(out_param.measure,'normal')) && ...
    (any(hyperbox(1,:)==hyperbox(2,:)) || ...
    any(hyperbox(1,:)>hyperbox(2,:)))
966     warning('GAIL:cubSobol_SI_fo_g:hyperboxnormalwrong',['If normal ...
        measure, hyperbox must be defined as (-Inf,Inf)^d.' ...
967             ' Using default hyperbox:'])
968     disp([-inf*ones(1,out_param.d);inf*ones(1,out_param.d)])
969     hyperbox = [-inf*ones(1,out_param.d);inf*ones(1,out_param.d)];
970 end
971
972 end
973
974
975 function y = estimate_min(down, up)
976     if down(4)*up(4) ≤ 0
977         y = min(max(max(down(1)-max(down(2).^2, up(2).^2)...
978         ,0)./max(up(3), eps), 0),1);
```

```
979     else
980         y = min(max(max(down(1)—max(down(2).^2, up(2).^2)...
981         ,0)./max(up(3)—min(down(4).^2, up(4).^2), eps), 0),1);
982     end
983 end
984
985 function y = estimate_max(down, up)
986     if down(2)*up(2) ≤ 0
987         y = min(max(max(up(1)...
988     ,0)./max(down(3)—max(down(4).^2, up(4).^2), eps), 0),1);
989     else
990         y = min(max(max(up(1)—min(down(2).^2, up(2).^2)...
991     ,0)./max(down(3)—max(down(4).^2, up(4).^2), eps), 0),1);
992     end
993 end
```

# BIBLIOGRAPHY


[1] I. Sobol', "Global sensitivity indices for nonlinear mathematical models and their Monte Carlo estimates," *Mathematics and Computers in Simulation (MATCOM)*, vol. 55, no. 1, pp. 271–280, 2001.

[2] P. Glasserman, *Monte Carlo Methods in Financial Engineering*, vol. 53 of *Applications of Mathematics*. New York: Springer-Verlag, 2004.

[3] P. J. Lujan, "Top Quark Mass Measurement Using a Matrix Element Method with Quasi-Monte Carlo Integration," Tech. Rep. arXiv:0810.3739, Lawrence Berkeley National Laboratory, Oct 2008. Comments: 3 pages. Proceedings for the 34th International Conference on High Energy Physics (ICHEP08), Philadelphia, 2008. Minor text fixes.

[4] I. Volobouev, "Matrix Element Method in HEP: Transfer functions, efficiencies, and likelihood normalization," tech. rep., Texas Tech University, 2011. arXiv:1101.2259 [physics.data-an].

[5] A. Keller, "Quasi-Monte Carlo image synthesis in a nutshell," in *Monte Carlo and Quasi-Monte Carlo Methods 2012* (J. Dick, F. Y. Kuo, G. W. Peters, and I. H. Sloan, eds.), vol. 65 of *Springer Proceedings in Mathematics and Statistics*, pp. 213–249, Springer Berlin Heidelberg, 2013.

[6] C. Wächter and A. Keller, "Quasi-Monte Carlo light transport simulation by efficient ray tracing," May 31 2011. US Patent 7,952,583.

[7] A. Genz, "Numerical computation of multivariate normal probabilities," *J. Comput. Graph. Statist.*, vol. 1, pp. 141–150, 1992.

[8] F. J. Hickernell and H. S. Hong, "Computing multivariate normal probabilities using rank-1 lattice sequences," in *Proceedings of the Workshop on Scientific Computing* (G. H. Golub, S. H. Lui, F. T. Luk, and R. J. Plemmons, eds.), (Hong Kong), pp. 209–215, Springer-Verlag, Singapore, 1997.

[9] F. J. Hickernell and Ll. A. Jiménez Rugama, "Reliable adaptive cubature using digital sequences," in *Monte Carlo and Quasi-Monte Carlo Methods: MCQMC, Leuven, Belgium, April 2014* (R. Cools and D. Nuyens, eds.), (Cham), pp. 367–383, Springer International Publishing, 2016.

[10] Ll. A. Jiménez Rugama and F. J. Hickernell, "Adaptive multidimensional integration based on rank-1 lattices," in *Monte Carlo and Quasi-Monte Carlo Methods: MCQMC, Leuven, Belgium, April 2014* (R. Cools and D. Nuyens, eds.), (Cham), pp. 407–422, Springer International Publishing, 2016.

[11] F. J. Hickerenll and Ll. A. Jiménez Rugama and Da Li, "Adaptive quasi-Monte Carlo methods," 2016+. Manuscript in preparation.

[12] L. Gilquin and Ll. A. Jiménez Rugama, "Reliable error estimation for Sobol' indices," *Statistics and Computing*, 2016+. Submitted for publication: https://hal.inria.fr/hal-01358067/document.

[13] A. B. Owen, "Scrambling Sobol' and Niederreiter-Xing points," *J. Complexity*, vol. 14, pp. 466–489, 1998.

[14] H. Niederreiter and C. Xing, "Quasirandom points and global function fields," in *Finite Fields and Applications* (S. Cohen and H. Niederreiter, eds.), no. 233 in London Math. Soc. Lecture Note Ser., pp. 269–296, Cambridge University Press, 1996.

[15] H. Niederreiter, *Random Number Generation and Quasi-Monte Carlo Methods*. CBMS-NSF Regional Conference Series in Applied Mathematics, Philadelphia: SIAM, 1992.

[16] I. H. Sloan and S. Joe, *Lattice Methods for Multiple Integration*. Oxford: Oxford University Press, 1994.

[17] J. Dick, F. Y. Kuo, and I. H. Sloan, "High-dimensional integration: The quasi-Monte Carlo way," *Acta Numerica*, vol. 22, pp. 133–288, 005 2013.

[18] P. L'Ecuyer and D. Munger, "Algorithm xxx: A general software tool for constructing rank-1 lattice rules," *ACM Transactions on Mathematical Software*, 2016+.

[19] E. Hlawka, "Funktionen von beschränkter Variation in der Theorie der Gleichverteilung," *Ann. Mat. Pura Appl.*, vol. 54, pp. 325–333, 1961.

[20] I. H. Sloan, "Lattice methods for multiple integration," *J. Comput. Appl. Math.*, vol. 12 & 13, pp. 131–143, 1985.

[21] F. J. Hickernell, "Obtaining $O(N^{-2+\epsilon})$ convergence for lattice quadrature rules," in *Monte Carlo and Quasi-Monte Carlo Methods 2000* (K. T. Fang, F. J. Hickernell, and H. Niederreiter, eds.), pp. 274–289, Springer-Verlag, Berlin, 2002.

[22] N. Clancy, Y. Ding, C. Hamilton, F. J. Hickernell, and Y. Zhang, "The cost of deterministic, adaptive, automatic algorithms: Cones, not balls," *J. Complexity*, vol. 30, pp. 21–45, 2014.

[23] A. B. Owen, "Randomly permuted $(t, m, s)$-nets and $(t, s)$-sequences," in *Monte Carlo and Quasi-Monte Carlo Methods in Scientific Computing* (H. Niederreiter and P. J.-S. Shiue, eds.), vol. 106 of *Lecture Notes in Statistics*, pp. 299–317, Springer-Verlag, New York, 1995.

[24] S.-C. T. Choi, Y. Ding, F. J. Hickernell, L. Jiang, Ll. A. Jiménez Rugama, X. Tong, Y. Zhang, and X. Zhou, "GAIL: Guaranteed Automatic Integration Library (versions 1.0–2.1)." MATLAB software, 2013–2015.

[25] B. D. Keister, "Multidimensional quadrature algorithms," *Computers in Physics*, vol. 10, pp. 119–122, 1996.

[26] I. M. Sobol', "On sensitivity estimation for nonlinear mathematical models," *Matem. Mod.*, vol. 2, no. 1, pp. 112–118, 1990.

[27] W. Hoeffding, "A class of statistics with asymptotically normal distributions," *Annals of Mathematical Statistics*, vol. 19, no. 3, pp. 293–325, 1948.

[28] A. Saltelli, "Making best use of model evaluations to compute sensitivity indices," *Computer Physics Communications*, vol. 145, no. 2, pp. 280 – 297, 2002.

[29] T. A. Mara and O. R. Joseph, "Comparison of some efficient methods to evaluate the main effect of computer model factors," *Journal of Statistical Computation and Simulation*, vol. 78, no. 2, pp. 167–178, 2008.

[30] M. D. McKay, J. D. Morrison, and S. C. Upton, "Evaluating prediction uncertainty in simulation models," *Computer Physics Communications*, vol. 117, no. 1, pp. 44 – 51, 1999.

[31] L. Gilquin, Ll. A. Jiménez Rugama, E. Arnaud, F. J. Hickernell, H. Monod, and C. Prieur, "Iterative construction of replicated designs based on Sobol' sequences," *Comptes Rendus Mathematique*, 2016+. Submitted for publication: https://hal.inria.fr/hal-01349444/document.

[32] J.-Y. Tissot and C. Prieur, "A randomized orthogonal array-based procedure for the estimation of first- and second-order Sobol' indices," *Journal of Statistical Computation and Simulation*, vol. 85, no. 7, pp. 1358–1381, 2015.

[33] P. Bratley, B. L. Fox, and H. Niederreiter, "Implementation and tests of low-discrepancy sequences," *ACM Trans. Model. Comput. Simul.*, vol. 2, pp. 195–213, 1992.

[34] A. B. Owen, "Better estimation of small Sobol' sensitivity indices," *ACM Trans. Model. Comput. Simul.*, vol. 23, pp. 11:1–11:17, May 2013.

[35] H. S. Hong and F. J. Hickernell, "Algorithm 823: Implementing scrambled digital nets," *ACM Trans. Math. Software*, vol. 29, pp. 95–109, 2003.

[36] N. Bakhvalov, "On the optimality of linear methods for operator approximation in convex classes of functions," *USSR Computational Mathematics and Mathematical Physics*, vol. 11, no. 4, pp. 244 – 249, 1971.

[37] I. H. Sloan and H. Woźniakowski, "When are quasi-Monte Carlo algorithms efficient for high dimensional integrals," *J. Complexity*, vol. 14, pp. 1–33, 1998.

[38] J. Dick, "On quasi-Monte Carlo rules achieving higher order convergence," in *Monte Carlo and Quasi-Monte Carlo Methods 2008* (P. L'Ecuyer and A. Owen, eds.), pp. 73–96, Springer-Verlag, Berlin, 2010.

[39] M. B. Giles, "Multilevel Monte Carlo path simulation," *Oper. Res.*, vol. 56, pp. 607—617, 2008.

[40] F. J. Hickernell, C. Lemieux, and A. B. Owen, "Control variates for quasi-Monte Carlo," *Statist. Sci.*, vol. 20, pp. 1–31, 2005.

[41] A. Keller and L. Grünschloß, "Parallel quasi-Monte Carlo integration by partitioning low discrepancy sequences," in *Monte Carlo and Quasi-Monte Carlo Methods 2010* (L. Plaskota and H. Woźniakowski, eds.), vol. 23 of *Springer Proceedings in Mathematics and Statistics*, pp. 487–498, Springer Berlin Heidelberg, 2012.