MATHEMATICS OF COMPUTATION
Volume 75, Number 254, Pages 903–920
S 0025-5718(06)01785-6
Article electronically published on January 4, 2006

FAST ALGORITHMS FOR COMPONENT-BY-COMPONENT CONSTRUCTION OF RANK-1 LATTICE RULES IN SHIFT-INVARIANT REPRODUCING KERNEL HILBERT SPACES

DIRK NUYENS AND RONALD COOLS

ABSTRACT. We reformulate the original component-by-component algorithm for rank-1 lattices in a matrix-vector notation so as to highlight its structural properties. For function spaces similar to a weighted Korobov space, we derive a technique which has construction cost $O(sn\log(n))$, in contrast with the original algorithm which has construction cost $O(sn^2)$. Herein s is the number of dimensions and n the number of points (taken prime). In contrast to other approaches to speed up construction, our fast algorithm computes exactly the same quantity as the original algorithm. The presented algorithm can also be used to construct randomly shifted lattice rules in weighted Sobolev spaces.

1. INTRODUCTION

We consider s-dimensional integrals over the unit cube,

$$I(f) := \int_{[0,1)^s} f(\mathbf{x})\,\mathrm{d}\mathbf{x},$$

which we want to approximate by an n point rank-1 lattice,

$$Q_n(f) := \frac{1}{n}\sum_{k=0}^{n-1} f\left(\left\{\frac{k\mathbf{z}}{n}\right\}\right),$$

with generating integer vector $\mathbf{z}$. The braces denote the fractional part taken componentwise, i.e., $\{\mathbf{x}\} := \mathbf{x} \pmod 1$, and the components of $\mathbf{z}$ are chosen from the set $\mathcal{Z}_n := \{1 \le z < n : \gcd(z,n) = 1\}$. As is usual, and for simplicity, we take n prime so that $\mathcal{Z}_n = \{1, 2, \ldots, n-1\}$.

The generating vector is chosen in such a way as to minimize a certain discrepancy measure. In the case of lattice rules, the integrand functions are usually supposed to be in a tensor-product reproducing kernel Hilbert space $\mathcal{H}$, with kernel K, and the worst-case cubature error follows naturally as the discrepancy measure (we refer to [7] for a more in-depth view of the theory). The worst-case error is

Received by the editor June 22, 2004 and, in revised form, November 26, 2004.
2000 *Mathematics Subject Classification.* Primary 65D30, 65D32, 68W40.
Key words and phrases. Numerical integration, quasi–Monte Carlo, rank-1 lattice rules, component-by-component construction, fast algorithms.
This research is part of a project financially supported by the Onderzoeksfonds K.U.Leuven / Research Fund K.U.Leuven.

defined as the supremum of all possible errors made when picking functions in the unit ball,

$$e(Q_n, K) := \sup\{|I(f) - Q_n(f)| : f \in \mathcal{H}(K), \|f\| \le 1\}. \tag{1}$$

In this paper we will only consider periodic functions. For this kind of function spaces there exist component-by-component construction algorithms for the generating vector $\mathbf{z}$, which minimize the worst-case error, component by component; see [12, 8], among others, for construction in a Korobov space. These component-by-component algorithms have, in principle, a construction cost of $O(sn^2)$. Since we are interested in constructing rank-1 lattice rules with a large number of points, we would benefit from a reduction of the factor n^2. The presented algorithm in this paper achieves a construction cost of $O(sn\log(n))$, realizing the fast construction of rank-1 lattice rules.

For simplicity and concreteness we will consider functions in a weighted Korobov space $E_{\alpha,\boldsymbol{\gamma}}$. These are functions which have a converging Fourier series, are $[0,1)^s$-periodic, and have suitably decaying Fourier-coefficients, with decay α. Positive weights γ_j are assigned to each dimension j to model functions which are anisotropic, in the sense that the first component x_1 is supposed to be more important than x_2, and so on. This means we have a decaying sequence of positive weights associated with each dimension:

$$\gamma_1 \ge \gamma_2 \ge \cdots \ge \gamma_s \ge 0.$$

The kernel $K_{s,\boldsymbol{\gamma}}$, for such a weighted Korobov space $E_{\alpha,\boldsymbol{\gamma}}$, is given by

$$K_{s,\boldsymbol{\gamma}}(\mathbf{x},\mathbf{y}) = \prod_{j=1}^{s} \left(1 + \gamma_j \sum_{h_j \in \mathbb{Z}\setminus\{0\}} \frac{\exp(2\pi \mathrm{i}\; h_j(x_j - y_j))}{|h_j|^\alpha} \right). \tag{2}$$

This is a product of 1-dimensional kernels K_{1,γ_j},

$$K_{1,\gamma}(x,y) = 1 + \gamma \sum_{h \in \mathbb{Z}\setminus\{0\}} \frac{\exp(2\pi \mathrm{i}\; h(x-y))}{|h|^\alpha},$$

where we can substitute $x \mapsto \{x - y\}$ and $y \mapsto 0$, since the kernel is shift-invariant. These kernels are defined in the Fourier domain, and thus the two summands in the above 1-dimensional kernel denote the constant part and the variable part. For the rest of this paper we will denote the variable part of the kernel as $\omega(x)$, which for a Korobov space is

$$\omega(x) = \sum_{h \in \mathbb{Z}\setminus\{0\}} \frac{\exp(2\pi \mathrm{i}\; hx)}{|h|^\alpha}. \tag{3}$$

The quantity $\omega(x)$ is independent of the function space weighting, and we have that

$$K_{1,\gamma}(x,y) := 1 + \gamma\, \omega(\{x-y\}). \tag{4}$$

Note that the infinite sum in (3) can be written in terms of a Bernoulli polynomial when $\alpha \ge 2$ is even [1, page 805],

$$\omega(x) = \frac{(2\pi)^\alpha}{(-1)^{\alpha/2-1}\alpha!} B_\alpha(x).$$

In Section 2 we explain the component-by-component algorithm for weighted shift-invariant function spaces (such as the weighted Korobov space), which has direct construction cost $O(sn^2)$. We rephrase this algorithm in a slightly different form in Section 3, so as to stress the aspects which will turn out to be the key concepts for speeding up the algorithm. In Section 4 we take a look at the structure of the most important ingredient of the algorithm, namely the kernel matrix K_n, and derive a significantly faster algorithm in Section 5, which has construction cost $O(sn\log(n))$. In Section 6 we will compare some numerical results of the fast algorithm with some previously reported results, and in Section 7 we summarize the important aspects of the new algorithm.

2. The component-by-component algorithm

The worst-case error (1) in an s-dimensional reproducing kernel Hilbert space, with a shift-invariant kernel K, can be written as

$$e(P_n, K) = \left\{ -\int_{[0,1)^s} K(\mathbf{x}, \mathbf{0})\,\mathrm{d}\mathbf{x} + \frac{1}{n^2} \sum_{\mathbf{x},\mathbf{y}\in P_n} K\left(\{\mathbf{x}-\mathbf{y}\}, \mathbf{0}\right) \right\}^{1/2}, \tag{5}$$

with P_n the set of n sample points for the cubature rule Q_n; see [7].

Since we consider a tensor-product space, and since the difference of two lattice points is another lattice point, we can rewrite (5) as

$$e(P_n, K) = \left\{ -1 + \frac{1}{n} \sum_{k=0}^{n-1} \prod_{j=1}^{s} \left(1 + \gamma_j\, \omega\left(\left\{\frac{kz_j}{n}\right\}\right)\right) \right\}^{1/2}, \tag{6}$$

where the double sum over $\mathbf{x}$ and $\mathbf{y}$ is now replaced by a single sum.

In a component-by-component algorithm we calculate the worst-case error for increasing dimension s. In each step we choose a z_s from the set $\mathcal{Z}_n$ which minimizes the worst-case error, and we keep all previous z_j, $j < s$, fixed. Under certain conditions on the weights, this algorithm will construct a lattice rule with optimal rate of convergence [8, 4]. The algorithm is presented as Algorithm 1.

Algorithm 1 CBC for shift-invariant tensor-product RKHS

```
for $s = 1$ to $s_{\max}$ do
  for all $z_s \in \mathcal{Z}_n$ do
    $e_s^2(z_s) = -1 + \frac{1}{n}\sum_{k=0}^{n-1}\prod_{j=1}^{s}\left(1+\gamma_j\,\omega\left(\left\{\frac{kz_j}{n}\right\}\right)\right)$
  end for
  $z_s = \operatorname{argmin}_{z\in\mathcal{Z}_n} e_s^2(z)$
end for
```

The apparent construction cost of the algorithm is dictated by the evaluation of the worst-case error (6), which is $O(ns\phi_\omega)$, where ϕ_ω is the cost of evaluating the function ω, which we assume to be bounded by a constant and so can be neglected. This worst-case error has to be calculated for each of the n possible choices of z_s, and that for $s = 1$ to $s_{\max}$, which brings the total direct construction cost of the algorithm to $O(s_{\max}^2 n^2)$.

Because the function space under consideration is a tensor-product space, the s-dimensional kernel can be written as a product of s 1-dimensional kernels. When iterating over increasing dimensions s it is useful to split this product into two parts,

$$\prod_{j=1}^{s}\left(1+\gamma_j\,\omega\left(\left\{\frac{kz_j}{n}\right\}\right)\right)=\left(\prod_{j=1}^{s-1}\left(1+\gamma_j\,\omega\left(\left\{\frac{kz_j}{n}\right\}\right)\right)\right)\cdot\left(1+\gamma_s\,\omega\left(\left\{\frac{kz_s}{n}\right\}\right)\right), \tag{7}$$

and to store the previous n products to reduce the construction cost of a direct implementation of the algorithm from $O(s_{\max}^2 n^2)$ to $O(s_{\max} n^2)$. Moreover, there is (almost) no reason not to store the previous n products, as the time-penalty for not doing so is significant. This is especially so for large n. See Section 6 for a numerical comparison.

In many publications the weighted function space has additional weights β_j attached to the constant part of the reproducing kernel, where the 1's in (2) are changed into β_j's. We will not consider these weights. Also note that in some publications (e.g., [7]) the γ_j for the variable part are called β_j.

3. Component-by-component rephrased

When inspecting Algorithm 1 it is clear that $\omega(x)$ is only evaluated in n different points, and so these can be calculated beforehand as

$$\omega_\ell := \omega(\ell/n), \qquad \ell = 0,\dots,(n-1),$$

and stored in a vector $\boldsymbol{\omega}$. When the value of $\omega(\{kz/n\})$ is needed, it is sufficient to take element $k\cdot z \pmod{n}$ of the $\boldsymbol{\omega}$ vector, which we will write as

$$\omega_{z\cdot k} := \omega(\{kz/n\}). \tag{8}$$

We are now making the choice of storing the n products from the previous iteration in a vector, as given in (7), explicit, and call this vector $\mathbf{p}$. Since this vector changes for every iteration, we add a subscript to indicate which products it represents. So $\mathbf{p}_{s-1}$ stores the products up to dimensions $(s-1)$. The inner loop of Algorithm 1 then becomes

```
**for all** $z_s \in \mathcal{Z}_n$ **do**
    $e_s^2(z_s) = -1 + \frac{1}{n}\sum_{k=0}^{n-1}\left(1+\gamma_s\,\omega\left(\left\{\frac{kz_s}{n}\right\}\right)\right)\cdot p_{s-1}(k)$
**end for**
```

This loop over z_s, together with the sum over k, expresses a matrix-vector product. Please note that when a vector already has a subscript, e.g., the dimension s, we use parentheses to denote a certain element. So $p_{s-1}(k)$ means element k of vector $\mathbf{p}_{s-1}$. Now by introducing the two $(n-1)\times n$ matrices

$$\Omega_n := \left[\omega_{z,k}\right] = \left[\omega_{z\cdot k}\right]_{\substack{z=1,\dots,(n-1)\\ k=0,\dots,(n-1)}} \tag{9}$$

and

$$K_{n,\gamma_s} := 1 + \gamma_s\,\Omega_n,$$

we can write this loop as

$$\begin{aligned}\mathbf{e}_s^2 &= -1 + \tfrac{1}{n}\, K_{n,\gamma_s} \cdot \mathbf{p}_{s-1} \\ &= -1 + \tfrac{1}{n}\, (1 + \gamma_s\, \Omega_n) \cdot \mathbf{p}_{s-1}.\end{aligned} \tag{10}$$

We call the matrix K_{n,γ_s} the kernel matrix and Ω_n the variable part of the kernel matrix. Note that wherever we write a scalar plus a matrix, as in $1 + \gamma_s\, \Omega_n$, it is meant that this scalar is added to each element of the matrix.

After we pick the value for z_s associated with the minimum of the vector $\mathbf{e}_s^2$, we can update the product vector. Looking at (7) it is clear that we must multiply every element $p_{s-1}(k)$ with $1 + \gamma_s\, \omega_{z_s \cdot k}$ to obtain the new element $p_s(k)$:

$$p_s(k) = (1 + \gamma_s\, \omega_{z_s \cdot k})\, p_{s-1}(k), \qquad \text{for all } k.$$

This is the same as elementwise multiplication with row z_s of K_{n,γ_s}. Let $\operatorname{diag}(\mathbf{x})$ denote the diagonal matrix with the elements of $\mathbf{x}$ on its diagonal and zero elsewhere, and let $\mathbf{v}_{z_s}$ denote a $(n-1)$ selection vector with 1 in position z_s and zero elsewhere. Then we obtain

$$\begin{aligned}\mathbf{p}_s &= \operatorname{diag}\left(\mathbf{v}_{z_s}^T \cdot K_{n,\gamma_s}\right) \cdot \mathbf{p}_{s-1} \\ &= \operatorname{diag}\left(1 + \gamma_s\, \mathbf{v}_{z_s}^T \cdot \Omega_n\right) \cdot \mathbf{p}_{s-1}.\end{aligned} \tag{11}$$

This brings us to the same component-by-component algorithm as in Section 2, but this time rephrased in a matrix-vector notation, presented as Algorithm 2. The construction cost of this algorithm is by consequence the same as that of Algorithm 1 with caching: a (general) matrix-vector product has time complexity $O(n^2)$, the time complexity for finding the minima and updating the product vector is $O(n)$, and so this brings the total construction cost to $O(s_{\max} n^2)$.

Algorithm 2 CBC for shift-invariant tensor-product RKHS (matrix-vector form)

```
for $s = 1$ to $s_{\max}$ do
    $\mathbf{e}_s^2 = -1 + \frac{1}{n}\, K_{n,\gamma_s} \cdot \mathbf{p}_{s-1}$
    $z_s = \operatorname*{argmin}_{z \in \mathcal{Z}_n} e_s^2(z)$
    $\mathbf{p}_s = \operatorname{diag}\left(\mathbf{v}_{z_s}^T \cdot K_{n,\gamma_s}\right) \cdot \mathbf{p}_{s-1}$
end for
```

So, nothing is new here, except from the computational point of view, where we can now choose to use a heavily optimized matrix-vector routine to do the hard work. Even more, we can now use a matrix-vector routine for structured matrices.

4. The structure of Ω_n and $K_{n,\gamma}$

By using the notation introduced in (8) we can clearly demonstrate the structure of the matrix Ω_n defined in (9):

$$\Omega_n = \begin{bmatrix} \omega_{1\cdot 0} & \omega_{1\cdot 1} & \omega_{1\cdot 2} & \cdots & \omega_{1\cdot(n-1)} \\ \omega_{2\cdot 0} & \omega_{2\cdot 1} & \omega_{2\cdot 2} & \cdots & \omega_{2\cdot(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \omega_{(n-1)\cdot 0} & \omega_{(n-1)\cdot 1} & \omega_{(n-1)\cdot 2} & \cdots & \omega_{(n-1)\cdot(n-1)} \end{bmatrix}$$

$$= \begin{bmatrix} \omega_0 & \omega_1 & \omega_2 & \cdots & \omega_{n-1} \\ \omega_0 & \omega_2 & \omega_4 & \cdots & \omega_{n-2} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \omega_0 & \omega_{n-1} & \omega_{n-2} & \cdots & \omega_1 \end{bmatrix}. \tag{12}$$

Obviously the matrix $K_{n,\gamma}$ has the same structure, but it may change in every iteration due to a different γ, whereas the matrix Ω_n is completely defined by the variable kernel part ω and the number of points n. We will therefore concentrate on Ω_n, which has the appealing visual fractal-like representation of Figure 1.

Let us first define the class of circulant matrices: a matrix C_n is circulant if it is specified by its first column $\mathbf{c}$ as

$$C_n = \mathrm{circ}(\mathbf{c}) := \begin{bmatrix} c_0 & c_{n-1} & c_{n-2} & \cdots & c_1 \\ c_1 & c_0 & c_{n-1} & \cdots & c_2 \\ c_2 & c_1 & c_0 & \cdots & c_3 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ c_{n-1} & c_{n-2} & c_{n-3} & \cdots & c_0 \end{bmatrix}.$$

A matrix-vector multiplication of a vector $\mathbf{x}$ with a circulant matrix $C_n = \mathrm{circ}(\mathbf{c})$, can be seen as the convolution of $\mathbf{x}$ and $\mathbf{c}$,

$$C_n \cdot \mathbf{x} = \mathbf{c} * \mathbf{x}.$$

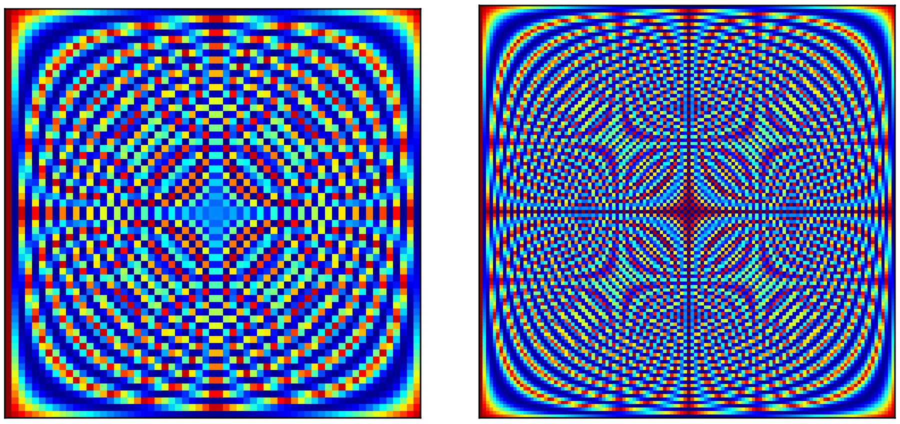

Figure 1. The structure of Ω_n for $n = 61$ and $n = 122$ (using a Korobov kernel with $\alpha = 2$)

In what follows we will write $\mathbf{x}'$ when we need the vector $\mathbf{x}$ without its first entry. Similarly, given a matrix A_n, with

$$A_n = \Big[a_{\ell,k}\Big]_{\substack{\ell=0,\ldots,(n-1)\\ k=0,\ldots,(n-1)}},$$

we denote with A'_n the submatrix

$$A'_n = \Big[a_{\ell,k}\Big]_{\substack{\ell=1,\ldots,(n-1)\\ k=1,\ldots,(n-1)}},$$

i.e., A_n without its first row and column.

We first look closer at a matrix G_n, which looks a lot like the matrix Ω_n, but is a square $(n \times n)$-matrix with the following structure:

$$G_n := \begin{bmatrix} g_0 & g_0 & g_0 & \cdots & g_0 \\ g_0 & g_1 & g_2 & \cdots & g_{n-1} \\ g_0 & g_2 & g_4 & \cdots & g_{n-2} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ g_0 & g_{n-1} & g_{n-2} & \cdots & g_1 \end{bmatrix}.$$

This matrix could be concisely defined, with the notation introduced in (8), as

$$G_n = \Big[g_{\ell,k}\Big] = \Big[g_{\ell\cdot k}\Big]_{\substack{\ell=0,\ldots,(n-1)\\ k=0,\ldots,(n-1)}}. \tag{13}$$

The matrix G_n is fully specified by its second column $\mathbf{g}$. For this kind of structured matrices we can generalize a factorization for discrete Fourier matrices by the name of Rader factorization [10, 13]. This factorization is based on the following number-theoretic permutation (for n prime):

$$\mathbf{z} = \Pi^T_{n,\delta} \cdot \mathbf{x} \quad \Leftrightarrow \quad z_\ell = x_k, \quad \begin{cases} k = \ell & \text{if} \quad \ell = 0, \\ k = \delta^{\ell-1} \pmod{n} & \text{if} \quad \ell = 1, \ldots, (n-1), \end{cases} \tag{14}$$

where δ is a primitive root of n, and so δ is a generator of the cyclic group of order $(n-1)$ with elements $\delta^r \bmod n$, $r = 0, 1, \ldots, n-2$.

Theorem 1 (Rader Factorization)**.** *Given a matrix G_n, of order n, with structure as in* (13)*, and given a primitive root ρ of n, the matrix G_n can be factored as*

$$G_n = \Pi_{n,\rho} \cdot \begin{bmatrix} g_0 & g_0 \mathbf{1}^T_{n-1} \\ g_0 \mathbf{1}_{n-1} & C_{n-1} \end{bmatrix} \cdot \Pi^T_{n,\rho^{-1}}.$$

Here the permutations $\Pi_{n,\rho}$ and $\Pi_{n,\rho^{-1}}$ are defined as in (14)*, the matrix C_{n-1} is a circulant matrix of order $(n-1)$ defined by*

$$C_{n-1} = \operatorname{circ}(\mathbf{c}')$$
$$\mathbf{c} := \Pi^T_{n,\rho} \cdot \mathbf{g},$$

and $\mathbf{1}_{n-1}$ is a one-vector of size $(n-1)$.

Proof. Since row $\ell = 0$ and column $k = 0$ get mapped onto itself, we only have to prove that

$$\Big[C_{n-1}\Big]_{\substack{\ell=1,\ldots,(n-1)\\ k=1,\ldots,(n-1)}} = \Pi'^T_{n,\rho} \cdot G'_n \cdot \Pi'_{n,\rho^{-1}}$$
$$= \Big[G'_n\Big]_{\rho^{\ell-1},(\rho^{-1})^{k-1}},$$

and since the elements of G_n are indexed as $(i \cdot j) \pmod{n}$, we have

$$\rho^{\ell-1} \cdot (\rho^{-1})^{k-1} = \rho^{\ell-k} \pmod{n}.$$

This is a matrix of which the first column, $\ell = 1, \ldots, (n-1)$ and $k = 1$ is given by

$$\Pi'^{T}_{n,\rho} \cdot \mathbf{g}' = \mathbf{c}',$$

as stated in the theorem. To prove this matrix is circulant we note that all diagonal elements are the same, and we are only left to prove that the elements above the main diagonal, i.e., $k > \ell$, match their corresponding element from the first column,

$$\begin{aligned} \rho^{n-k+\ell-1} &= \rho^{n-1}\rho^{\ell-k} \pmod{n} \\ &= \rho^{\ell-k} \pmod{n}, \end{aligned}$$

where we used the fact that $\rho^{n-1} = 1$, since ρ is a primitive root of n. □

Note that since we are considering the case n prime, a primitive root of n which generates $\{1, \ldots, n-1\}$ always exists, and thus the factorization from Theorem 1 is always possible.

Since our matrix Ω_n is just the matrix G_n without its first row, see (9) and (13), we are able to obtain a similar factorization.

Corollary 1. *Given an $(n-1) \times n$ matrix Ω_n with structure as in* (9)*, and given a primitive root ρ of n, the matrix Ω_n can be factored as*

$$\Omega_n = \Pi'_{n,\rho} \cdot \begin{bmatrix} \omega_0 \mathbf{1}_{n-1} & C_{n-1} \end{bmatrix} \cdot \Pi^T_{n,\rho^{-1}}.$$

Here the permutations $\Pi_{n,\rho}$ and $\Pi_{n,\rho^{-1}}$ are defined as in (14)*, and the matrix C_{n-1} is a circulant matrix of order $(n-1)$ defined by*

$$\begin{aligned} C_{n-1} &= \operatorname{circ}(\boldsymbol{\psi}'), \\ \boldsymbol{\psi} &:= \Pi^T_{n,\rho} \cdot \boldsymbol{\omega}, \end{aligned}$$

and $\mathbf{1}_{n-1}$ is a one-vector of size $(n-1)$.

Proof. This is a trivial modification of Theorem 1. □

When the kernel is symmetric, that is $\omega(x) = \omega(1-x)$, there is another computationally important structural property of the matrix Ω_n: its horizontal and vertical symmetry, and this in addition to Ω_n already being centrosymmetric. These structural properties are clearly visible in Figure 1 for a Korobov kernel, where only the first column of Ω_n is nonregular.

When the decay parameter α for the Korobov space is even, the kernel is symmetric, since [1]

$$B_\alpha(1-x) = (-1)^\alpha B_\alpha(x).$$

Due to this symmetry we only have to search half of the possible z values, and we have $e^2_{n-z} = e^2_z$ and $p_{n-k} = p_k$. As a consequence we can halve the number of calculations per z value in (10), winning a constant factor of 4 in total.

If we would transform the matrix Ω_n into a more computationally suitable form, then it would also be nice to keep all of the advantages of its symmetries. We now show that such horizontal and vertical symmetry results in a periodic form when using permutations of the form (14) as used in Corollary 1.

Theorem 2. *Given a vector* $\mathbf{w}$ *of length* $(n-1)$*, which has symmetry* $w_k = w_{n-k}$*, and* δ *a primitive root of* n*, we obtain a periodic vector* $\mathbf{s}$ *after permutation* (14),

$$\mathbf{s} = \Pi'^{T}_{n,\delta} \cdot \mathbf{w} = \begin{bmatrix} \mathbf{t} \\ \mathbf{t} \end{bmatrix},$$

with two copies of a vector $\mathbf{t}$ *with length* $m = (n-1)/2$.

Likewise, given a matrix G'_n *of order* $(n-1)$*, with horizontal and vertical symmetry, and* ρ *a primitive root of* n*, after permutation* (14),

$$C_{n-1} = \Pi'^{T}_{n,\rho} \cdot G'_n \cdot \Pi'_{n,\rho^{-1}} = \begin{bmatrix} C_m & C_m \\ C_m & C_m \end{bmatrix},$$

we obtain a horizontally and vertically periodic matrix C_{n-1} *with four copies of a circulant matrix* C_m *of order* $m = (n-1)/2$.

Proof. Without loss of generality we only prove the periodicity for the vector case. It is also obvious that C_m is circulant if C_{n-1} is circulant, which is proven in Theorem 1.

The periodicity in the vector $\mathbf{s}$ can be expressed as

$$s_\ell = s_{\ell+(n-1)/2},$$

where these indices are generated by permutation (14) from a source index k, and we should prove that

$$\begin{cases} \delta^{\ell-1} = k, \\ \delta^{\ell+(n-1)/2-1} = n-k. \end{cases}$$

The first part is true by definition (14). Using the fact that $\delta^{(n-1)/2} = n-1$ if δ is a primitive root of n, we find

$$\begin{aligned} \delta^{\ell+(n-1)/2-1} &= \delta^{\ell-1}\,\delta^{(n-1)/2} \\ &= k\,(n-1) \\ &= n-k \end{aligned}$$

modulo n, completing the proof. □

5. Component-by-component revisited

With the result of Corollary 1 we can work out the matrix-vector multiplication for the calculation of the worst-case error vector as it appears in (10). We obtain

$$\begin{aligned} K_{n,\gamma} \cdot \mathbf{p} &= (1 + \gamma\,\Omega_n) \cdot \mathbf{p} \\ &= \Pi'_{n,\rho} \cdot \left[\begin{array}{c|c} (1+\gamma\,\psi_0)\,\mathbf{1}_{n-1} & \operatorname{circ}\left(1+\gamma\,\boldsymbol{\psi}'\right) \end{array} \right] \cdot \Pi^{T}_{n,\rho^{-1}} \cdot \mathbf{p}, \end{aligned}$$

and now set $\mathbf{q} := \Pi^{T}_{n,\rho^{-1}} \cdot \mathbf{p}$ and bring $\mathbf{q}$ inside:

$$= \Pi'_{n,\rho} \cdot \Big[(1+\gamma\,\psi_0)\,q_0 + \operatorname{circ}\left(1+\gamma\,\boldsymbol{\psi}'\right) \cdot \mathbf{q}' \Big].$$

The circulant matrix-vector multiplication can be rewritten as

$$\begin{aligned}\operatorname{circ}(1+\gamma\,\boldsymbol{\psi}')\cdot\mathbf{q}' &= (1+\gamma\,\boldsymbol{\psi}')*\mathbf{q}'\\ &= \gamma\,(\boldsymbol{\psi}'*\mathbf{q}') + \sum_{k=1}^{n-1} q_k,\end{aligned}$$

where in the last step we used $(\alpha\mathbf{x}+\beta)*\mathbf{y} = \alpha(\mathbf{x}*\mathbf{y}) + \beta\sum_k y_k$.

Putting all the pieces together, the squared worst-case error vector from (10) can be calculated in the permuted space as

$$\mathbf{E}_s^2 := \Pi_{n,\rho}^{'T}\cdot\mathbf{e}_s^2 = \frac{1}{n}\left(\gamma_s\,(\boldsymbol{\psi}'*\mathbf{q}'_{s-1}) + \gamma_s\,\psi_0\,q_{s-1}(0) + \sum_{k=0}^{n-1} q_{s-1}(k)\right) - 1. \tag{15}$$

In contrast to formula (10) in the original algorithm, which has time-complexity $O(n^2)$, the above formula can be computed in time $O(n\log(n))$, using a fast-convolution algorithm, which calculates the convolution product in the Fourier domain [6, p. 201],

$$\mathbf{x}*\mathbf{y} = \mathcal{F}^{-1}(\operatorname{diag}(\mathcal{F}(\mathbf{x}))\cdot\mathcal{F}(\mathbf{y})),$$

where the discrete Fourier transform $\mathcal{F}$ is done by an FFT. It is advantageous to find the minima in the permuted worst-case error vector $\mathbf{E}_s^2$. We will call the associated generator component in the permuted space w_s,

$$w_s = \operatorname*{argmin}_{w\in\mathcal{Z}_n} E_s^2(w),$$

and the unpermuted component z_s can be found by mapping w_s back:

$$z_s \xleftarrow{\Pi'_{n,\rho}} w_s.$$

Since we are now working with the permuted version of the product vector, we seek an update rule, as in (11), but now for $\mathbf{q}_s$:

$$\begin{aligned}\mathbf{q}_s &= \Pi_{n,\rho^{-1}}^T\cdot\mathbf{p}_s\\ &= \Pi_{n,\rho^{-1}}^T\cdot\operatorname{diag}\left(\mathbf{v}_{z_s}^T\cdot K_{n,\gamma_s}\right)\cdot\mathbf{p}_{s-1}\\ &= \Pi_{n,\rho^{-1}}^T\cdot\operatorname{diag}\left(\mathbf{v}_{z_s}^T\cdot K_{n,\gamma_s}\right)\cdot\Pi_{n,\rho^{-1}}\cdot\mathbf{q}_{s-1}.\end{aligned}$$

Observe that we have the same permutation on the rows and columns of a diagonal matrix. This can be done by applying this same permutation to the vector that defines the diagonal:

$$\begin{aligned}\mathbf{q}_s &= \operatorname{diag}\left(\mathbf{v}_{z_s}^T\cdot K_{n,\gamma_s}\cdot\Pi_{n,\rho^{-1}}\right)\cdot\mathbf{q}_{s-1}\\ &= \operatorname{diag}\left(1+\gamma_s\,\mathbf{v}_{z_s}^T\cdot\Omega_n\cdot\Pi_{n,\rho^{-1}}\right)\cdot\mathbf{q}_{s-1}.\end{aligned}$$

Using Corollary 1 we finally obtain

$$\begin{aligned}\mathbf{q}_s &= \operatorname{diag}\left(1+\gamma_s\,(\mathbf{v}_{z_s}^T\cdot\Pi'_{n,\rho})\cdot\left[\begin{array}{cc}\psi_0\,\mathbf{1}_{n-1} & \operatorname{circ}(\boldsymbol{\psi}')\end{array}\right]\right)\cdot\mathbf{q}_{s-1}\\ &= \operatorname{diag}\left(1+\gamma_s\,\mathbf{v}_{w_s}^T\cdot\left[\begin{array}{cc}\psi_0\,\mathbf{1}_{n-1} & \operatorname{circ}(\boldsymbol{\psi}')\end{array}\right]\right)\cdot\mathbf{q}_{s-1}.\end{aligned} \tag{16}$$

This update rule for $\mathbf{q}$ is very similar to the update rule for $\mathbf{p}$ in (11). Here we recombine the old $\mathbf{q}'$ vector, elementwise with the row of the circulant matrix $\operatorname{circ}(\boldsymbol{\psi}')$ which creates the minimal value in the permuted worst-case error $\mathbf{E}_s^2$. It can be seen that the value for $q_s(0)$ is independent of the choice for z_s, for all s.

Plugging in (15), for the calculation of the worst-case error, together with the above update formula for the product vector (16) into Algorithm 2, gives us the new Algorithm 3: a fast $O(s_{\max} n \log(n))$ component-by-component construction algorithm. In practice only the $\Pi_{n,\rho}^{\prime T}$ permutation must be constructed for the initial calculation of the $\boldsymbol{\psi}$-vector and to convert each w_s value back to a z_s value. Updating $\mathbf{q}$ is a simple case of multiplying with the right elements of $\boldsymbol{\psi}$. Due to the structure of the circulant matrix, this can be done by stepping through the $\boldsymbol{\psi}$-vector in reverse order.

Algorithm 3 Fast CBC for shift-invariant tensor-product RKHS

```
$\widetilde{\boldsymbol{\psi}}' = \mathcal{F}(\boldsymbol{\psi}')$
**for** $s = 1$ **to** $s_{\max}$ **do**
    $\mathbf{E}_s^2 = \frac{1}{n}\left(\gamma_s\, \mathcal{F}^{-1}\left(\operatorname{diag}(\widetilde{\boldsymbol{\psi}}') \cdot \mathcal{F}(\mathbf{q}'_{s-1})\right) + \gamma_s\, \psi_0\, q_{s-1}(0) + \sum_{k=0}^{n-1} q_{s-1}(k)\right) - 1$
    $w_s = \operatorname*{argmin}_{w \in \mathcal{Z}_n} E_s^2(w)$
    $\mathbf{q}_s = \operatorname{diag}\left(1 + \gamma_s\, \mathbf{v}_{w_s}^T \cdot \left[\begin{array}{cc} \psi_0\, \mathbf{1}_{n-1} & \operatorname{circ}(\boldsymbol{\psi}') \end{array}\right]\right) \cdot \mathbf{q}_{s-1}$
**end for**
```

6. Numerical results

6.1. **Implementation details.** In this section we present some numerical results obtained by implementations of Algorithms 1, 2, and 3. The tests were run on a Linux workstation equipped with a 2.4 GHz Pentium 4 processor with hyper-threading and 2 GB of RAM. All implementations used for timings use IEEE double precision floating point and were all compiled with the same level of optimizations. Four versions of the component-by-component algorithm were implemented in C:

- *fastrank1*: This is the fast algorithm using the ideas from this paper and presented in Algorithm 3, obtaining a construction cost of $O(sn\log(n))$. We used the FFTW3 library [5] for shifting back and forth to the frequency domain and the C99 double complex type for calculating the convolution.
- *spmvrank1*: This is an explicit matrix-vector algorithm as given in Algorithm 2, where an optimized level 2 BLAS routine was used for the symmetric matrix-vector multiplication (the *dspmv* routine), obtaining $O(sn^2)$. The BLAS used was the Intel® Math Kernel Library for Linux version 6.
- *rank1*: This is the original algorithm with caching of the product vector and precalculation of the $\boldsymbol{\omega}$-vector as given in Algorithm 2, obtaining $O(sn^2)$.
- *slowrank1*: This is the original algorithm without caching and no precalculations, using the least possible memory,[1] in other words a straight implementation of Algorithm 1, obtaining $O(s^2n^2)$.

All four implementations use the symmetry-trick from Section 4 to search only half the space of z candidates and to do half the number of calculations per z value. For the *fastrank1* algorithm this trick was justified in Theorem 2.

[1]Memory consumption is the reason to mention both the *rank1* and *slowrank1* algorithms, although what follows will surely discourage the usage of the algorithm in the *slowrank1* form.

TABLE 1. Time and memory complexity for s dimensions and n points

algorithm	total time	memory
fastrank1	$O(sn\log(n))$	$O(2n)$
spmvrank1	$O(sn^2)$	$O(n^2/8)$
rank1	$O(sn^2)$	$O(n)$
slowrank1	$O(s^2n^2)$	–

The *spmvrank1* routine was used to obtain an idea of the possible speed of a completely optimized and tuned *rank1* routine. One could use all kinds of (system-dependent) optimizations: instruction-level parallelism, optimize register and cache usage, optimize locality, blocking, ... But all this work is already incorporated in an optimized BLAS implementation. In sacrifice of memory we can use a BLAS2 routine to do the matrix-vector multiplication, in order to get an idea of how fast the *rank1* routine could be made. Consequently, this routine is not of practical interest because the relevant part of the Ω_n matrix must be kept in memory, which is $m(m+1)/2$ doubles, with $m = (n-1)/2$. Note that the extra memory needed for *spmvrank1* is just a side-effect of using the BLAS routine, and would not be needed for a fully optimized *rank1* routine (which would also need much less memory bandwidth).

An overview of the construction cost and memory consumption of all the algorithms is given in Table 1. The constants for the memory complexity are retained to give an approximate idea about the memory limitations of a certain algorithm. Since just counting floating point operations does not always give accurate running time predictions, we also timed the four routines for different n in 20 dimensions. The total running time is illustrated in Figure 2, which also gives a view on the constants for the time complexity. From the figure it can be seen that the complexity difference of the new *fastrank1* algorithm clearly pays off.

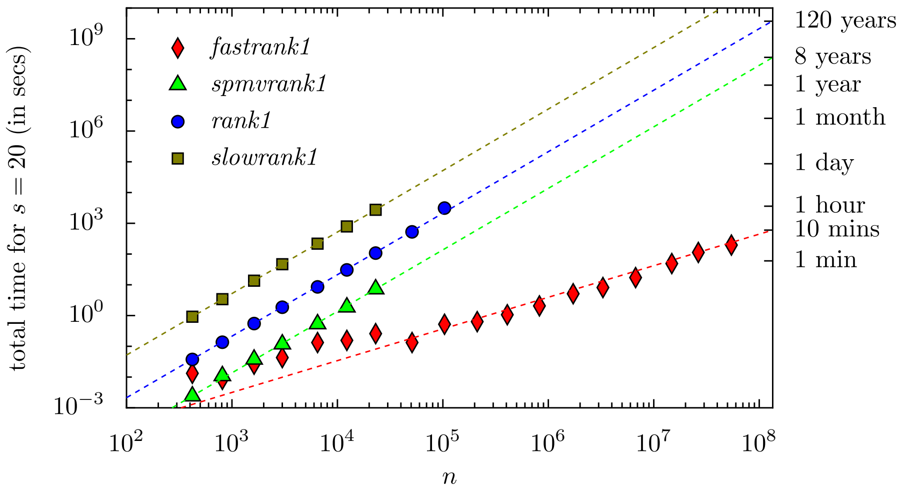

FIGURE 2. Total construction times of the four routines for various point sizes in 20 dimensions

Three of the four routines spent some of their time in a setup phase, and we can thus split the time complexity in two parts: a setup time t_{setup} and an iteration time t_{it}. The total time needed to construct an s-dimensional rule with n points is then given by

$$t_{\text{total}}(s,n) = t_{\text{setup}}(n) + s\, t_{\text{it}}(n).$$

Now we briefly explain the time-complexity of each of these two parts for those routines:

- The *rank1* routine's setup phase consists of precalculating the $\boldsymbol{\omega}$-vector, having a time complexity of $O(n)$ which can be neglected against the further quadratic cost (and for all n presented in this paper was less than our timer resolution of 10 ms).
- The *spmvrank1* routine needs $O(n^2)$ time for constructing the Ω_n matrix, which is in accordance with its iteration cost.
- The iteration time of the *fastrank1* algorithm is clearly $O(n\log(n))$, consisting of two FFTs and n multiplications. In the setup phase the situation is a little bit more complex. Here we need to find a primitive root of n and, because we are using FFTW, prepare a plan for the FFTs in the setup phase. We use a naive algorithm for finding the primitive root which for every primitive root candidate needs r checks, with r the number of factors of $(n-1)$. From a statistical point of view it is clear that not many attempts need to be made to find the generator ρ. The preparation of the FFT plans were done by the heuristic module[2] of FFTW, for which the time is also dependent on the number of factors of $(n-1)$ (actually of m). The time complexity of the setup phase is thereby not transparent but assumed to be almost linear in the number of factors of $(n-1)$.

We can thus conclude that the complexity of the iteration time is always the dominant factor, and this is certainly the case when s is large. This makes the iteration time the preferred measure to compare the different algorithms. In Figure 3 the iteration times for these three routines are shown. From this plot it is clear that the irregularities in the beginning of the data for *fastrank1* in Figure 2 can be accredited to irregularities in the setup phase for these n.

The documentation of FFTW states that the time complexity of its FFT implementation for an m point FFT is always $O(m\log(m))$ (even for prime m, for which the original Rader factorization is used on which Theorem 1 is based) [5]. However the performance of the library degrades for large prime factors. Following the documentation, it is best to choose m of the form

$$m = 2^a\, 3^b\, 5^c\, 7^d\, 11^e\, 13^f,$$

with $e + f \leq 1$. So the choices for n in this paper were made with the previous rule in mind. Starting from 421 and each time picking prime numbers which were approximately twice as large and having factors for $(n-1)$ of at most 7, the largest n we used was 54 454 681. The importance of picking good n values is illustrated in Figure 4, where we also timed the iteration time for the previous five primes and the next five primes for each n.

We now return to our remark about the two original algorithms: *slowrank1* (which is a straightforward implementation of Algorithm 1) and *rank1* (which is an

[2]FFTW also includes a mode to first time different possible plans and then pick the best.

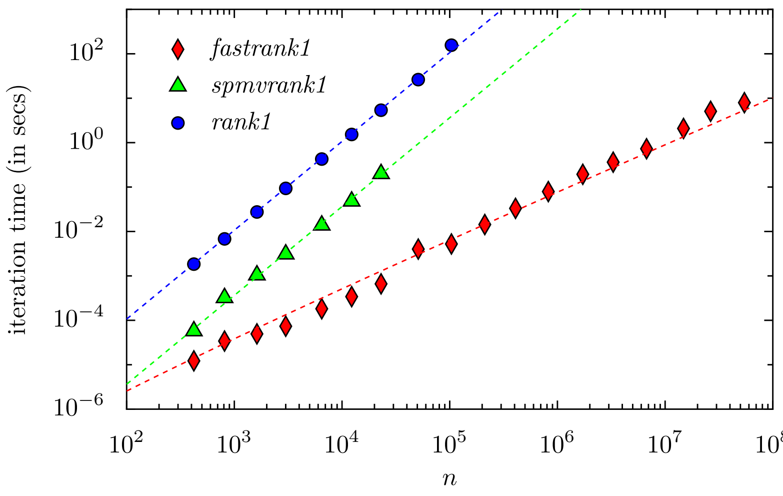

FIGURE 3. Iteration times which are dimension independent

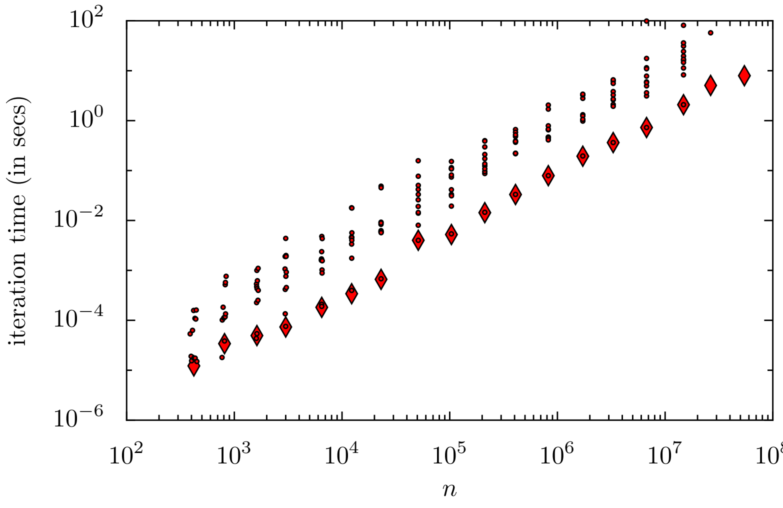

FIGURE 4. Iteration times of the selected points and their environments

implementation of Algorithm 2). As can be seen in Table 1 the memory cost for *rank1* is n doubles, being m cached products and an m-vector to represent half of the $\boldsymbol{\omega}$-vector. Suppose we have available a modest memory size of 1 GB for storing these values; then $n = 2^{30}/2^{3} = 2^{27}$. We can use the least-squares fits of Figure 2 to estimate the time needed for $n = 2^{27} \approx 1.34 \cdot 10^{8}$ points in 20 dimensions. For our implementation of the *rank1* routine this estimate is more than 120 years. Since this time estimate makes *rank1* infeasible even for a modest memory size of 1 GB, we cannot imagine that anyone would want to use the *slowrank1* routine. Note that the time needed for *fastrank1* is 10 minutes.

6.2. **Results discussion.** In the appendix of [9] we list some tables which verify the correct workings of the fast algorithm by comparing its output with some published results. Many of these results match up exactly, while some give a small difference. Two factors are involved: numerical stability and rounding errors, and choice between different z which obtain (almost) the same minima. Concerning the inexactness of the floating point calculations, it is to be expected that the fast algorithm will make less inexactness errors than the original algorithm due to the significantly smaller amount of calculations done. This will be investigated in future work.

In [11] an algorithm for constructing randomly shifted rank-1 lattices in Sobolev space is presented. A shift-invariant kernel can be associated with the kernel of a Sobolev space, satisfying the form and structure for our 1-dimensional kernel. This shift-invariant kernel is the kernel for a weighted Korobov space with parameters

$$\alpha = 2, \qquad \widehat{\beta}_j = \beta_j + \gamma_j\,(a_j^2 - a_j + \frac{1}{3}), \qquad \widehat{\gamma}_j = \frac{\gamma_j}{2\pi^2}.$$

The worst-case error calculated for this shift-invariant kernel then corresponds to the expected value for the worst-case error in the Sobolev space; we refer the reader to [11] for further details. Therefore the presented algorithms can also be used to construct randomly shifted rank-1 lattices in weighted Sobolev spaces.

In Table 2 and Table 3 we present similar tables as in [8] for different weighted Korobov spaces and different weighted Sobolev spaces. In these tables the differences with the tables in [8] are marked with underlines. Note that our results were calculated in double precision (i.e., 53 bit mantissa), while those by Kuo were calculated in long double precision (i.e., 64 bit mantissa).

TABLE 2. Worst-case error e for 100 dimensions for weighted Korobov space with $\alpha = 2$, γ_j as specified above each column, $\beta_j = 1$ and n as specified in the first column

n	0.9^j	0.5^j	0.1^j	j^{-1}	j^{-2}	j^{-6}
4001	2.0242e+02	9.8282e−03	1.9988e−04	1.0759e+01	3.1$\underline{264}$e−02	6.89$\underline{95}$e−04
8009	1.4256e+02	5.9293e−03	1.024$\underline{1}$e−04	7.6069e+00	1.9793e−02	3.57$\underline{72}$e−04
16001	1.01$\underline{51}$e+02	3.5$\underline{558}$e−03	5.1$\underline{961}$e−05	5.3$\underline{817}$e+00	1.2435e−02	1.8223e−04
32003	7.1$\underline{876}$e+01	2.06$\underline{31}$e−03	2.6526e−05	3.$\underline{7939}$e+00	7.$\underline{9071}$e−03	9.369$\underline{5}$e−05
64007	5.0$\underline{634}$e+01	1.1980e−03	1.338$\underline{7}$e−05	2.6762e+00	4.9801e−03	4.7580e−05

TABLE 3. Worst-case error e for 100 dimensions for the shift-invariant kernel constructed for a weighted Sobolev space with $a_j = 1$, γ_j as specified above each column, $\beta_j = 1$ and n as specified in the first column

n	0.9^j	0.5^j	0.1^j	j^{-1}	j^{-2}	j^{-6}
4001	3.20$\underline{60}$e−02	1.9$\underline{776}$e−04	3.472$\underline{7}$e−05	9.2597e−03	3.7846e−04	1.0653e−04
8009	2.0162e−02	1.0388e−04	1.7383e−05	5.$\underline{6899}$e−03	2.0$\underline{379}$e−04	5.340$\underline{2}$e−05
16001	1.28$\underline{24}$e−02	5.4924e−05	8.707$\underline{4}$e−06	3.5744e−03	1.1$\underline{128}$e−04	2.676$\underline{7}$e−05
32003	8.0782e−03	2.868$\underline{5}$e−05	4.3$\underline{617}$e−06	2.2159e−03	6.0764e−05	1.342$\underline{3}$e−05
64007	5.0783e−03	1.480$\underline{0}$e−05	2.18$\underline{03}$e−06	1.38$\underline{17}$e−03	3.295$\underline{1}$e−05	6.7$\underline{183}$e−06

TABLE 4. Worst-case error e for 100 dimensions for the shift-invariant kernel constructed for a weighted Sobolev space with $a_j = 1$, γ_j as specified above each column, $\beta_j = 1$ and n as specified in the first column, values for composite n with r factors are from [2], values for prime n were calculated with *fastrank1*

n	r	0.5^j	j^{-2}	n	r	0.5^j	j^{-2}
2005001		6.1091e−07	1.6863e−06	2022157		6.0392e−07	1.6746e−06
2005007	2	7.1750e−07	1.9173e−06	2022161	4	8.6847e−07	2.4180e−06
2005019		6.1467e−07	1.6871e−06	2022187		5.9751e−07	1.6699e−06
2825567		4.4693e−07	1.2554e−06	2857159		4.3669e−07	1.2442e−06
2825617	2	5.1953e−07	1.4570e−06	2857177	4	6.4611e−07	1.8787e−06
2825639		4.4360e−07	1.2412e−06	2857181		4.3771e−07	1.2406e−06
4003981		3.2042e−07	9.3030e−07	3963161		3.2586e−07	9.3449e−07
4003997	2	3.7002e−07	1.0686e−06	3963181	4	4.9601e−07	1.3965e−06
4003999		3.2266e−07	9.3199e−07	3963209		3.2574e−07	9.3602e−07
5659627		2.3416e−07	6.9021e−07	5699773		2.3282e−07	6.8156e−07
5659637	2	2.7406e−07	8.0221e−07	5699779	4	3.3709e−07	1.0358e−06
5659651		2.3364e−07	6.9164e−07	5699789		2.3068e−07	6.8772e−07
8037191		1.6884e−07	5.1508e−07	7989011		1.7033e−07	5.1630e−07
8037211	2	1.9148e−07	5.9812e−07	7989013	4	2.5473e−07	7.4932e−07
8037229		1.6788e−07	5.1271e−07	7989067		1.6952e−07	5.8935e−07
1966079		6.1410e−07	1.7121e−06	1937207		6.3266e−07	1.7341e−06
1966087	3	7.8342e−07	2.2806e−06	1937221	5	1.0260e−06	2.8180e−06
1966123		6.1996e−07	1.7204e−06	1937227		6.2941e−07	1.7234e−06
2837381		4.4037e−07	1.2499e−06	2956783		4.2416e−07	1.2011e−06
2837407	3	5.6658e−07	1.6320e−06	2956811	5	7.3529e−07	1.9358e−06
2837431		4.4132e−07	1.2503e−06	2956813		4.2714e−07	1.2029e−06
4055927		3.1438e−07	9.1864e−07	4075289		3.1548e−07	9.1508e−07
4055929	3	4.1256e−07	1.2326e−06	4075291	5	4.8902e−07	1.5027e−06
4055957		3.1800e−07	9.1966e−07	4075297		3.1291e−07	9.1487e−07
5604997		2.3561e−07	7.0158e−07	5513623		2.3793e−07	7.0828e−07
5605027	3	3.1262e−07	9.3287e−07	5513623	5	4.1240e−07	1.1734e−06
5605037		2.3412e−07	6.9903e−07	5513663		2.3706e−07	7.0871e−07
8022409		1.6866e−07	5.1711e−07	7971311		1.6928e−07	5.1866e−07
8022431	3	2.3335e−07	6.7881e−07	7971317	5	2.8110e−07	8.1762e−07
8022437		1.6784e−07	5.1246e−07	7971323		1.7056e−07	5.1687e−07

In [3] and [2], Dick and Kuo present a variation on the original algorithm which constructs rank-1 lattices for composite n which constructs the generator per factor of n. They call this algorithm “Partial search” in contrast to the original algorithm which is then dubbed “Full search”. The “Partial search” algorithm has construction cost $O(sn(p_1 + p_2 + \cdots + p_r))$, where n is a product of the distinct prime numbers p_1, p_2, $\ldots$, p_r. In this way they can construct rank-1 lattice rules with “millions of points”. However in the same papers it is also shown that the theoretical convergence of these lattice rules degrades as the number of factors of n increases. In Table 4 we compare the worst-case error of a lattice rule with n prime with those published in [2], calculated in long double precision (i.e., 64-bit mantissa). These numbers were missing in [2] because they could not be calculated.

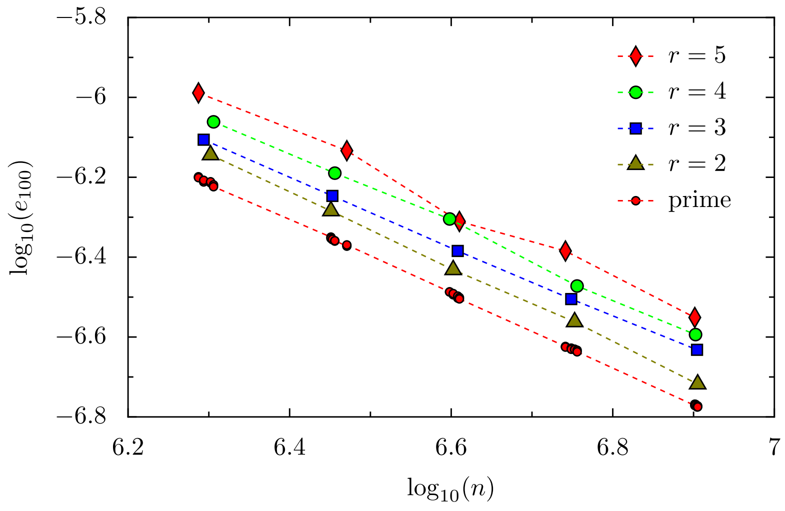

FIGURE 5. Comparison of "Partial search" for composite n with prime n for randomized Sobolev rules with $\gamma_j = 0.5^j$

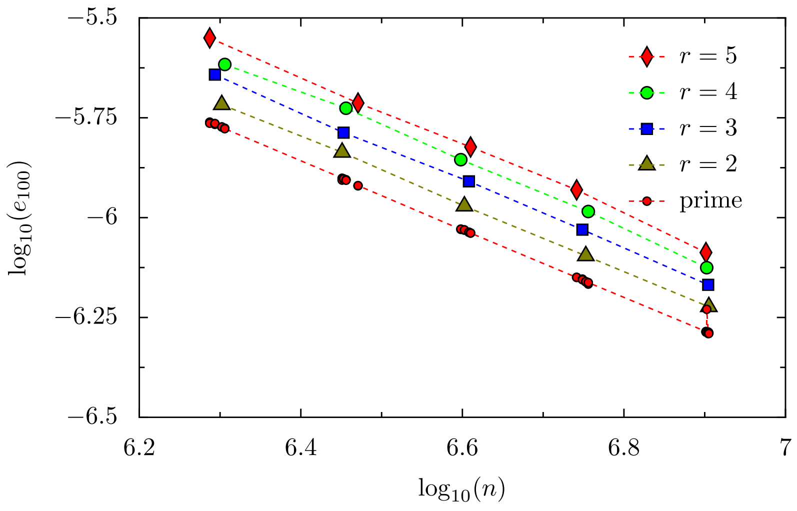

FIGURE 6. Comparison of "Partial search" for composite n with prime n for randomized Sobolev rules with $\gamma_j = j^{-2}$

Figures 5 and 6 show the same information. As can be seen in Figure 6, the worst-case error for $n = 7\,989\,067$ and $\gamma_j = j^{-2}$ suffers from numerical noise. This effect will be investigated at a later time. The other results behave as expected.

In the appendix of [9] we also provide a table for an equally weighted Korobov space with $\alpha = 2$, $\gamma_j = 1/s_{\max}$ and $n = 54\,454\,681$ as a reference point.

7. CONCLUSION

We showed that it is possible to construct rank-1 lattice rules with "millions of points" without using a composite number of points. In [3] and [2] the theoretical rate of convergence is given for composite n consisting of r factors. This theoretical convergence decreases as r increases, so it seems profitable to be able to construct rules with the number of points being prime as is done in this paper. Figures 5 and 6 confirm this. While the construction cost depends on the factors of n in [3] and [2], the construction cost here depends on the factors of $(n-1)$ and does not have a negative influence on the theoretical convergence. As already suggested this algorithm can also be used for the construction of randomly shifted rank-1 lattices in Sobolev spaces.

References

1. Milton Abramowitz and Irene A. Stegun (eds.), *Handbook of mathematical functions with formulas, graphs and mathematical tables*, National Bureau of Standards Applied Mathematics Series, vol. 55, U.S. Government Printing Office, Washington, D.C., 1964. MR0167642 (29:4914)
2. Josef Dick and Frances Kuo, *Constructing good lattice rules with millions of points*, Monte-Carlo and quasi-Monte Carlo Methods - 2002 (Harald Niederreiter, ed.), Springer-Verlag, January 2004, pp. 181–197. MR2076933
3. ______, *Reducing the construction cost of the component-by-component construction of good lattice rules*, Mathematics of Computation **73** (2004), 1967–1988. MR2059746 (2005b:65021)
4. Josef Dick, Ian H. Sloan, Xiaoqun Wang, and Henryk Woźniakowski, *Liberating the weights*, Journal of Complexity **20** (2004), no. 5, 593–623. MR2086942
5. Matteo Frigo and Steven G. Johnson, *FFTW: An adaptive software architecture for the FFT*, Proc. 1998 IEEE Intl. Conf. Acoustics Speech and Signal Processing, Vol. 3, IEEE, 1998, pp. 1381–1384.
6. Gene H. Golub and Charles F. Van Loan, *Matrix computations*, third ed., Studies in the Mathematical Sciences, Johns Hopkins University Press, 1996. MR1417720 (97g:65006)
7. Fred J. Hickernell, *Lattice rules: How well do they measure up?*, Random and Quasi-Random Point Sets (P. Hellekalek and G. Larcher, eds.), Springer-Verlag, 1998, pp. 109–166. MR1662841 (2000b:65007)
8. Frances Kuo, *Component-by-component constructions achieve the optimal rate of convergence for multivariate integration in weighted Korobov and Sobolev spaces*, Journal of Complexity **19** (2003), 301–320. MR1984116 (2004c:41066)
9. Dirk Nuyens and Ronald Cools, *Fast algorithms for component-by-component construction of rank-*1 *lattice rules in shift-invariant reproducing kernel Hilbert spaces*, Report TW392, Dept. of Computer Science, K.U.Leuven, May 2004.
10. Charles M. Rader, *Discrete Fourier transforms when the number of data samples is prime*, Proc. IEEE **5** (1968), 1107–1108.
11. Ian H. Sloan, Frances Kuo, and Stephen Joe, *Constructing randomly shifted lattice rules in weighted Sobolev spaces*, SIAM Journal on Numerical Analysis **40** (2002), no. 5, 1650–1665. MR1950616 (2003m:65031)
12. Ian H. Sloan and Andrew V. Reztsov, *Component-by-component construction of good lattice rules*, Mathematics of Computation **71** (2002), no. 237, 263–273. MR1862999 (2002h:65028)
13. Charles F. Van Loan, *Computational frameworks for the fast Fourier transform*, Frontiers in Applied Mathematics, vol. 10, SIAM, 1992. MR1153025 (93a:65186)

Katholieke Universiteit Leuven, Department of Computer Science, Celestijnenlaan 200A, B-3001 Heverlee, Belgium
E-mail address: `dirk.nuyens@cs.kuleuven.be`

Katholieke Universiteit Leuven, Department of Computer Science, Celestijnenlaan 200A, B-3001 Heverlee, Belgium
E-mail address: `ronald.cools@cs.kuleuven.be`