
Fast Component-by-Component Construction, a Reprise for Different Kernels

Dirk Nuyens and Ronald Cools

Dept. of Computer Science, K.U.Leuven, B-3001 Heverlee, Belgium
dirk.nuyens@cs.kuleuven.ac.be, ronald.cools@cs.kuleuven.ac.be

Summary. In [16] (Nuyens and Cools) it was shown that it is possible to generate rank-1 lattice rules with n points, n being prime, in a fast way. The construction cost in shift-invariant tensor-product reproducing kernel Hilbert spaces was reduced from $O(sn^2)$ to $O(sn \log(n))$, with s the number of dimensions. This reduction in construction cost was made possible by exploiting the algebraic structure of multiplication modulo a prime.

Here we look at the applications of the fast algorithm from a practical point of view. Although the choices for the function space are arbitrary, in practice only few kernels are used for the construction of rank-1 lattices. We will discuss component-by-component construction for the worst-case Korobov space, the average-case Sobolev space, the weighted lattice criterion $R_{n,\gamma}$ and polynomial lattice rules based on the digital Walsh kernel, of which the last two were presented at MC²QMC 2004 by Joe [11] and Dick, Leobacher and Pillichshammer, see e.g. [7]. We also give an example implementation of the algorithm in Matlab.

Keywords. Numerical integration, Quasi-Monte Carlo, Rank-1 lattice rules, Component-by-component construction, Fast algorithms, Digital nets

1 Introduction

We want to approximate an s -dimensional integral over the unit cube by an equal weight cubature rule,

$$I(f) = \int_{[0,1]^s} f(\mathbf{x}) d\mathbf{x} \approx Q(f) = \sum_{\mathbf{x}_k \in P_n} \frac{1}{n} f(\mathbf{x}_k), \quad (1)$$

where the n evaluation points are a rank-1 lattice

$$P_n = \left\{ \frac{k \cdot \mathbf{z}}{n} : 0 \leq k < n \right\}, \quad (2)$$

and by $k \cdot \mathbf{z}$ we mean (componentwise) multiplication modulo n . The integer vector \mathbf{z} is called the generating vector of the lattice and its components are

taken from the set $\mathcal{Z}_n = \{0 < z < n : \gcd(z, n) = 1\}$, i.e. $\mathbf{z} \in \mathcal{Z}_n^s$. In this paper we assume that n is prime and thus this set simplifies to $\mathcal{Z}_n = \{1, \dots, n-1\}$. The generating vector is chosen such as to minimize a certain error measure for the cubature rule given in (1). Here we minimize the worst-case error of functions in the unit ball of a shift-invariant tensor-product reproducing kernel Hilbert space $\mathcal{H}(K)$ with a given reproducing kernel K :

$$e(\mathbf{z}) = \sup \{|I(f) - Q(f)| : f \in \mathcal{H}(K), \|f\| \leq 1\}. \quad (3)$$

A particularly nice algorithm to construct a good generating vector \mathbf{z} is the component-by-component algorithm, see e.g. [10, 13, 19, 20]. In this algorithm the components of the generating vector \mathbf{z} are found component by component. This was first deemed not likely to be a successful strategy by Niederreiter [14, page 987] but later shown possible by Sloan and Reztsov [20]. This construction method makes the generated lattice rule naturally extensible in the number of dimensions. The typical construction cost of this algorithm is $O(sn^2)$, but can be significantly reduced to $O(sn \log(n))$, as described in previous work for n prime [16]. This fast algorithm makes it possible to construct lattice rules with a large number of points (say $n = 10^8$), which was practically intractable with the original algorithm. As an added advantage of the fast algorithm, we note that, due to the reduced number of arithmetic operations, its numerical qualities are also better.

In Section 2 we will present the component-by-component algorithm and rewrite it as a repeated matrix-vector multiplication with a fixed matrix Ω_n . Then in Section 3 we will show that this matrix-vector multiplication can be done in time $O(n \log(n))$ instead of the classical $O(n^2)$. Since the complete algorithm consists of s times this matrix-vector multiplication (when constructing a generating vector for s dimensions), the total time is $O(sn \log(n))$ as promised. In Section 4 we will show some of the applications in which this algorithm can be used. We will describe the worst-case Korobov space setting, the average-case Sobolev space setting, the weighted lattice criterion $R_{n,\gamma}$ and even the construction of polynomial lattice rules. At MC²QMC2004 the component-by-component algorithm was introduced for these last two, and in particular for a combination of those in the talk by Leobacher. This paper is written from a more practical point of view and therefore we present a possible implementation of the fast algorithm in Matlab in Section 5 and finally make some general remarks in Section 6.

2 The Component-by-Component Algorithm

The component-by-component algorithm will construct the generating vector \mathbf{z} for increasing dimension s up to s_{\max} . Hereby in each iteration s , the s -th component z_s is chosen and all previously chosen components remain fixed. We choose $z_s \in \mathcal{Z}_n$ which minimizes the worst-case error (3) in our shift-invariant

tensor-product reproducing kernel Hilbert space \mathcal{H} over all functions in the unit ball. The advantage of using a shift-invariant tensor-product reproducing kernel Hilbert space is that the expression for this worst-case error can be written explicitly as

$$e_{s,\gamma,\beta}(z_1, \dots, z_s) = \left(-\prod_{j=1}^s \beta_j + \frac{1}{n} \sum_{k=0}^{n-1} \prod_{j=1}^s \left(\beta_j + \gamma_j \omega\left(\frac{k \cdot z_j}{n}\right) \right) \right)^{1/2}. \quad (4)$$

See e.g. [9] for the derivation of such formulae in different settings and specifically in the case of a shift-invariant kernel.

Some words about notation are in order here. The kernel of an s -dimensional tensor-product space is the product of 1-dimensional kernels

$$K_{s,\gamma,\beta}(\mathbf{x}) = \prod_{j=1}^s (\beta_j + \gamma_j \omega(x_j)), \quad (5)$$

where ω is the variable kernel part and the γ_j are positive (product-type) weights which denote the importance of the different dimensions. Since we consider only shift-invariant kernels the variable kernel function has only one argument. The γ_j are taken as a decaying sequence of positive weights,

$$\gamma_1 \geq \gamma_2 \geq \dots \geq \gamma_s \geq 0,$$

which means that the successive coordinates are less and less important. The function ω was dubbed the *variable* part of the kernel in the previous work [16], since it can be seen to correspond to the non-constant frequency part of the Fourier series of the 1-dimensional kernel (up to scaling):

$$K_{1,\gamma,\beta}(x) = \sum_{h \in \mathbb{Z}} \hat{k}_h \exp(2\pi i h x) = \hat{k}_0 + \sum_{h \in \mathbb{Z} \setminus \{0\}} \hat{k}_h \exp(2\pi i h x).$$

Sometimes the *constant* part of the kernel is weighted by weights β_j . However a simple calculation shows that the worst-case error (4) for a function space with weights γ_j and β_j can be written in terms of the worst-case error in a function space with weights $\hat{\gamma}_j = \gamma_j/\beta_j$ and $\hat{\beta}_j = 1$. We then obtain the worst-case error as a scaled version of the worst-case error in this simpler function space as

$$e_{s,\gamma,\beta}^2 = \prod_{j=1}^s \beta_j e_{s,\hat{\gamma},\hat{\beta}}^2 = \prod_{j=1}^s \beta_j \left(-1 + \frac{1}{n} \sum_{k=0}^{n-1} \prod_{j=1}^s \left(1 + \frac{\gamma_j}{\beta_j} \omega\left(\frac{k \cdot z_j}{n}\right) \right) \right).$$

From this we see that (up to scaling) only the ratios γ_j/β_j matter. We want to emphasize however, that the function ω can be *any* function in our derivations, and so the choice of the reproducing kernel is completely free.

Given the expression for the worst-case error (4) and the description of the component-by-component construction, we can present the original algorithm as Algorithm 1. In the algorithm we consider the worst-case error in dimension s as a function of all the possible choices of $z_s \in \mathcal{Z}_n$. We then pick the minimum and keep this choice fixed for the following iterations. Under certain conditions on the weights γ_j , it can be shown that the constructed lattice rule achieves the optimal rate of convergence, see [13, 10, 6], and thus this construction method is justified.

Algorithm 1 (Original) CBC for shift-invariant tensor-product RKHS

```

for  $s = 1$  to  $s_{\max}$  do
  for all  $z_s \in \mathcal{Z}_n$  do
    
$$e_s^2(z_s) = - \prod_{j=1}^s \beta_j + \frac{1}{n} \sum_{k=0}^{n-1} \prod_{j=1}^s \left( \beta_j + \gamma_j \omega \left( \frac{k \cdot z_j}{n} \right) \right)$$

  end for
  
$$z_s = \operatorname{argmin}_{z \in \mathcal{Z}_n} e_s^2(z)$$

end for

```

A quick inspection of Algorithm 1 shows that a direct implementation would have a construction cost of $O(s^2 n^2)$. But since this is an iterative process in s , the product over s terms can be split in two parts: a product over the first $s - 1$ terms and just the last term:

$$\prod_{j=1}^s \left(\beta_j + \gamma_j \omega \left(\frac{k \cdot z_j}{n} \right) \right) = \prod_{j=1}^{s-1} \left(\beta_j + \gamma_j \omega \left(\frac{k \cdot z_j}{n} \right) \right) \left(\beta_s + \gamma_s \omega \left(\frac{k \cdot z_s}{n} \right) \right). \quad (6)$$

In iteration s the product up to dimension $s - 1$ is fixed, since we do not change the z_j , $j < s$, anymore. Consequently we can store the product in an n -vector denoted \mathbf{p}_{s-1} . Equation (6) then becomes

$$\prod_{j=1}^s \left(\beta_j + \gamma_j \omega \left(\frac{k \cdot z_j}{n} \right) \right) = p_{s-1}(k) \left(\beta_s + \gamma_s \omega \left(\frac{k \cdot z_s}{n} \right) \right),$$

which needs to be calculated for each k and each possible choice for z_s . This immediately allows us to rewrite the expression for the worst-case error as

$$e_s^2(z) = - \prod_{j=1}^s \beta_j + \frac{1}{n} \sum_{k=0}^{n-1} p_{s-1}(k) \left(\beta_s + \gamma_s \omega \left(\frac{k \cdot z}{n} \right) \right), \quad \forall z \in \mathcal{Z}_n. \quad (7)$$

It is not hard to see that (7) is a matrix-vector multiplication. Define the matrix

$$[\Omega_n]_{z,k} = \omega \left(\frac{k \cdot z}{n} \right),$$

which has at most n different elements (depending on the function ω), and define the worst-case error vector

$$\mathbf{e}_s^2 = -\bar{\beta}_s + \frac{1}{n}(\beta_s + \gamma_s \Omega_n) \mathbf{p}_{s-1} = -\bar{\beta}_s + \frac{\beta_s}{n} \sum_{k=0}^{n-1} p_{s-1}(k) + \frac{\gamma_s}{n} \Omega_n \mathbf{p}_{s-1},$$

where a scalar plus a matrix means adding this scalar to all elements of the matrix and $\bar{\beta}_s = \prod_{j=1}^s \beta_j$, the cumulative product up to dimension s . Then we can restate the component-by-component algorithm in its matrix-vector form as Algorithm 2. The only extra ingredient is the updating of the product vector after we have fixed a z_s . A simple calculation shows that this is done by multiplying each element of \mathbf{p}_{s-1} by the corresponding elements of the row z_s of Ω_n , denoted in a Matlab-like notation by $\Omega_n(z_s, :)$, after multiplication by the weight γ_s and addition of the constant β_s , which minimized $e_s^2(z)$.

Algorithm 2 CBC in matrix-vector form

```

 $\mathbf{p}_0 = \mathbf{1}$ 
for  $s = 1$  to  $s_{\max}$  do
   $\mathbf{e}_s^2 = -\bar{\beta}_s + \frac{1}{n}(\beta_s + \gamma_s \Omega_n) \mathbf{p}_{s-1}$ 
   $z_s = \operatorname{argmin}_{z \in \mathcal{Z}_n} e_s^2(z)$ 
   $\mathbf{p}_s = \operatorname{diag}(\beta_s + \gamma_s \Omega_n(z_s, :)) \mathbf{p}_{s-1}$ 
end for

```

The matrix-vector form in Algorithm 2, shows the real beauty of the component-by-component algorithm. A short description of the algorithm could now be given in terms of the variable kernel matrix Ω_n as:

1. Initialize an n -vector \mathbf{p} by all ones. Create a representation of the matrix Ω_n , i.e. evaluate the function ω in the points $\{0, 1, \dots, n-1\}/n$.
2. For each dimension s do the following:
 - a) Calculate the matrix-vector product of Ω_n and \mathbf{p} and assign this to a vector $\tilde{\mathbf{e}}^2$. (Note that it is not necessary to multiply with γ_s , add the constant β_s , divide by n and subtract $\bar{\beta}_s$. We only need to find the index z associated with the minimal value.)
 - b) Pick as choice for z_s the index z where the vector $\tilde{\mathbf{e}}^2$ has its minimum.
 - c) Update the vector \mathbf{p} as the elementwise product of row z_s of $(\beta_s + \gamma_s \Omega_n)$ with the current elements of \mathbf{p} . (Note that only here the γ_s , as well as the constant kernel part β_s , is of importance.)
3. Return the generating vector of the lattice rule.

The cost of step 1 is $O(nc_\omega)$, where c_ω is the cost of evaluating the function ω . We will consider c_ω constant. The cost of step 2a is the cost of a matrix-vector product, this is $O(n^2)$. Finding the minimum in step 2b is $O(n)$ and updating \mathbf{p} in step 2c is also $O(n)$. The total is thus $O(sn^2)$.

From a computational point of view it is now very clear that we could reduce the total construction cost of $O(sn^2)$ if we could find a matrix-vector

multiplication algorithm for our given matrix Ω_n with cost less than $O(n^2)$. Such a technique was introduced in [16] and is shortly restated in the next section.

3 Fast Matrix-Vector Multiplication

The technique introduced in [16] is reformulated in this section in a way that allows a generalization of the fast algorithm to the case that n is not a prime, see [17]. An exception where the fast algorithm is also directly applicable for $n = b^m$, with b prime, is presented in Section 4.4 on polynomial lattice rules.

Recall that the set $\mathcal{Z}_n = \{1, \dots, n-1\}$ here. This set together with multiplication modulo n forms the cyclic group \mathbb{Z}_n^* . Since the matrix Ω_n is defined as the function ω elementwise over elements of the form $(k \cdot z)/n$, we can identify part of this matrix with the Cayley table of this cyclic group.

Before we explain the fast matrix-vector multiplication with the matrix Ω_n we introduce the following notation: a prime on a symbol will denote that the 0-index is left out. Thus with \mathbf{p}' we mean the vector \mathbf{p} without its 0-th element, i.e. assuming that \mathbf{p} is defined as $p(k)$ with $k \in \mathbb{Z}_n$, this means that \mathbf{p}' is defined as $p(k)$ with $k \in \mathbb{Z}_n \setminus \{0\} = \mathbb{Z}_n^*$. The same holds for the matrix Ω_n , where Ω'_n is defined as

$$\Omega'_n = \left[\omega \left(\frac{k \cdot z}{n} \right) \right]_{\substack{k \in \mathbb{Z}_n^* \\ z \in \mathbb{Z}_n^*}}. \quad (8)$$

Thus Ω'_n has size $(n-1) \times (n-1)$, where the original matrix Ω_n has size $(n-1) \times n$. In other words we dropped the first column of Ω_n .

Using a trick originally introduced by Rader for deriving a fast Fourier transform when the number of points is prime [18], we can now permute the rows of Ω'_n by the positive powers of a generator g for \mathbb{Z}_n^* and the columns by the negative powers of this same generator. This will give us constant powers of g along the diagonals and bring Ω'_n in circulant form:

$$\begin{array}{c|cccccc} \cdot & 1 & g^{-1} & g^{-2} & \dots & g \\ \hline 1 & 1 & g^{-1} & g^{-2} & \dots & g \\ g & g & 1 & g^{-1} & \dots & g^2 \\ g^2 & g^2 & g & 1 & \dots & g^3 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ g^{-1} & g^{-1} & g^{-2} & g^{-3} & \dots & 1 \end{array}$$

It is well known that a circulant matrix has a similarity transform which has the Fourier matrix as its eigenvectors, see e.g. [3].

Theorem 1 (Diagonalization of a circulant matrix). *A circulant matrix C_m of order m has a similarity transform $C_m = F_m^{-1}DF_m$, where F_m is the Fourier matrix of order m and the eigenvalues are defined by the discrete Fourier transform of the first column \mathbf{c} of C_m as $D = \text{diag}(F_m\mathbf{c})$.*

Corollary 1 (Fast matrix-vector multiplication with a circulant matrix). *A matrix-vector multiplication of a vector \mathbf{x} with a circulant matrix C_m can be done in $O(m \log(m))$ by using the similarity transform of C_m :*

$$\begin{aligned} C_m\mathbf{x} &= F_m^{-1} \text{diag}(F_m\mathbf{c})F_m\mathbf{x} \\ &= \text{IFFT}(\text{diag}(\text{FFT}(\mathbf{c})) \text{FFT}(\mathbf{x})), \end{aligned}$$

where each discrete Fourier transform (as well as its inverse) is done using an FFT (or IFFT).

Note that for each m an m -point FFT (as well as an IFFT) is possible in time $O(m \log(m))$. A possible implementation is [8], which is also used by Matlab.

Plugging in this knowledge into Algorithm 2 results in a fast algorithm for component-by-component construction with cost $O(sn \log(n))$. We do not need to consider a matrix-vector multiplication with Ω_n , since the first column of Ω_n has constant value $\omega(0)$ and thus will not influence the minimum.

The circulant matrix after the Rader permutation on Ω'_n will be denoted by Ψ_n . It is completely defined by its first column $\boldsymbol{\psi}'$ which, given a generator g for \mathbb{Z}_n^* , can be calculated as

$$\boldsymbol{\psi}' = \left[\omega(g^{\ell-1}/n) \right]_{1 \leq \ell \leq n-1}. \quad (9)$$

The permutation defined by the positive powers of g is also needed to map the index w_s of the minimum in the row permuted error vector, call this E_s^2 , back to the unpermuted space as

$$z_s = g^{w_s-1}.$$

We can be oblivious of the permutation on the columns of Ω'_n , since we are not really interested in the value of the \mathbf{p} vector, and since on initialization this vector only contains the constant value 1, so the permutation would have no effect. For clarity we will however call the product vector in the permuted space \mathbf{q} .

This brings us to the following practical formulation of Algorithm 2:

1. Initialize an n -vector \mathbf{q} by all ones. Find a generator g for \mathbb{Z}_n^* and initialize an $(n-1)$ -vector $\boldsymbol{\psi}'$ as given in (9). Preferably: precalculate the FFT of $\boldsymbol{\psi}'$.
2. For each dimension s do the following:
 - a) Calculate the matrix-vector product of Ψ_n and \mathbf{q}' using FFT's and assign this to a vector $\tilde{\mathbf{E}}^2$.

- b) Pick w_s as the index in $\tilde{\mathbf{E}}^2$ where the vector has its minimum. Pick z_s as g^{w_s-1} .
 - c) Update the vector \mathbf{q}' as the elementwise product of row w_s of $(\beta_s + \gamma_s \Psi_n)$ with the current elements of \mathbf{q}' . The row corresponding to w_s can be formed as a circular up shift over w_s of ψ' , and this update rule is thus easily implementable (e.g. as a for-loop from w_s down to 1 and a second for-loop from $n-1$ down to w_s+1).
3. Return the generating vector of the lattice rule.

It can be seen that this algorithm has construction cost $O(sn \log(n))$. The only changes of cost are in step 1 and step 2a. The cost of step 1 is $O(n \log(n))$ if precalculation of the FFT is done. The cost of step 2a is now $O(n \log(n))$. An example implementation of this algorithm is given in Section 5.

4 Applications

In this section we will use weights β_j for the constant part of the kernel to easily accommodate other kernels, but we still only use γ_j weights to define the actual function space setting.

4.1 Worst-Case Korobov Space Setting

The Korobov function space is the space of $[0, 1)^s$ -periodic functions for which a converging Fourier series exists with Fourier coefficients decaying with a certain rate $\alpha > 1$. The 1-dimensional kernel is given by

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

The ω function in this case is

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

The infinite sum can be written in terms of a Bernoulli polynomial if α is even [1, page 805],

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

In practice α is taken to be 2, which is the smallest possible value. Larger values of α correspond to function spaces with smoother functions. So the typical choice is

$$\omega(x) = 2\pi^2(x^2 - x + 1/6), \quad \text{with } \alpha = 2.$$

4.2 Average-Case Sobolev Space Setting

We consider the Sobolev space in s dimensions which is a tensor-product of 1-dimensional Sobolev spaces of absolutely continuous functions over $[0, 1]$ whose first derivatives are in $L_2([0, 1])$. The 1-dimensional reproducing kernel is given by

$$K_{1,\gamma}(x, y) = 1 + \gamma \sigma_a(x, y),$$

where

$$\sigma_a(x, y) = \begin{cases} \min(|x - a|, |y - a|), & \text{if } (x - a)(y - a) > 0, \\ 0, & \text{if } (x - a)(y - a) \leq 0. \end{cases}$$

The parameter a is the anchor of this kernel, and is typically chosen as 0, 1 or $1/2$. Note that this kernel is not shift-invariant and so is a function in two variables x and y .

The study of this function space in the worst-case setting involves so-called shifted lattice rules. For a shifted lattice rule with shift $\Delta \in [0, 1]^s$, the point set takes the form

$$P_n = \left\{ \left(\frac{k \cdot \mathbf{z}}{n} + \Delta \right) \bmod 1 : 0 \leq k < n \right\}.$$

Since this kernel defines a function ω in two variables, it is not immediately applicable for the fast algorithm. Following [19, 9], we can define a shift-invariant kernel for this function space

$$\begin{aligned} K_{1,\gamma}^*(x) &= 1 + \gamma (x^2 - x + a^2 - a + 1/2) \\ &= 1 + \gamma (B_2(x) + a^2 - a + 1/3). \end{aligned}$$

When using this shift-invariant kernel K^* , we are actually calculating the worst-case error in a Korobov space with $\alpha = 2$ and adjusted weights

$$\widehat{\beta}_j = 1 + \gamma_j (a_j^2 - a_j + 1/3), \quad \widehat{\gamma}_j = \gamma_j / (2\pi^2).$$

The error now corresponds to the average-case error in the original Sobolev space by using random shifts Δ for the point set. Indeed, the shift-invariant kernel is obtained by taking the mean over all possible shifts, see [19] for details.

Taking a typical choice for the anchor of the space and adjusting the weights we can use the fast algorithm to construct randomly shifted lattice rules in a weighted Sobolev space by taking

$$\omega(x) = x^2 - x + 1/6, \quad \text{with } a = 1 \text{ and } \beta_j = 1 + \gamma_j/3.$$

4.3 The Lattice Rule Criterion R_n and the Connection with the Weighted Star Discrepancy

Joe considered a component-by-component construction method for rank-1 lattice rules which achieve the optimal rate of convergence $O(n^{-1+\delta})$, under certain conditions on the weights, for the weighted star discrepancy at MC²QMC2004, see [11]. Also see [10] for the unweighted version.

There is a link with the classical Koksma-Hlawka inequality by a bound on the star discrepancy D_n^* involving the quantity R_n (see [15])

$$D_n^*(\mathbf{z}) \leq \frac{s}{n} + \frac{R_n(\mathbf{z})}{2}.$$

So it suffices to only consider R_n to obtain a bound on D_n^* . Here, as in [11], we will consider the weighted star discrepancy, and thus the weighted quantity $R_{n,\gamma}$. This quantity $R_{n,\gamma}$ in s dimensions can be associated with the worst-case error of a product of the 1-dimensional kernels (consult [11] for details)

$$K_{1,\gamma}(x) = (1 + \gamma) + \gamma \sum_{\substack{-n/2 < h \leq n/2 \\ h \neq 0}} \frac{\exp(2\pi i h x)}{|h|}.$$

This kernel transforms easily into the needed form, and so we have that

$$\omega(x) = \sum_{\substack{-n/2 < h \leq n/2 \\ h \neq 0}} \frac{\exp(2\pi i h x)}{|h|^{\beta_j}}, \quad \text{with } \beta_j = 1 + \gamma_j.$$

To reduce the evaluation cost of the ω function, the asymptotic techniques in [12] should be used.

4.4 Polynomial Lattice Rules and the Digital Walsh Kernel

Another component-by-component construction method presented also at MC²QMC2004 by Dick, Leobacher and Pillichshammer, is that of polynomial lattice rules based on the digital Walsh kernel, see [6, 7]. These polynomial lattice rules are in fact (t, m, s) -nets and were introduced by Niederreiter, see [15, Section 4.4] for a more thorough description. Here the number of points is $n = b^m$ which is not prime, but we will show that the algorithm works due to the algebraic structure of polynomial multiplication over a field modulo an irreducible polynomial.

For polynomial lattice rules in base b the generating vector \mathbf{z} consists of polynomials over a finite field, $z_j \in (\mathbb{F}_b[x]/p)^*$, with p an irreducible polynomial over \mathbb{F}_b with $\deg(p) = m$. This means that there are $b^m - 1$ possible choices for the components of the generating vector of the lattice rule, which look like $a_0 + a_1x + \dots + a_{m-1}x^{m-1}$, with $a_\ell \in \mathbb{F}_b$ and not all a_ℓ zero.

Generalizing the construction of the point set (2) we consider the b -adic expansion $k = k_0 + k_1b + \dots + k_{m-1}b^{m-1}$ and associate a polynomial $k(x) = k_0 + k_1x + \dots + k_{m-1}x^{m-1}$ with each such k , $0 \leq k < b^m$. The multiplication is now defined modulo p and thus $q(x) = k(x) \cdot z_j(x) \in \mathbb{F}_b[x]/p$. To map $q(x)/p(x)$ into $[0, 1)$, we consider the formal Laurent series expansion of $q(x)/p(x)$ up to x^{-m} and then evaluate at $x = b$. This is denoted by the function v_m . We thus have that component j of point k is generated as

$$x_{k,j} = v_m \left(\frac{k(x) \cdot z_j(x)}{p(x)} \right) = v_m \left(\sum_{\ell=1}^{\infty} u_{\ell} x^{-\ell} \right) = \sum_{\ell=1}^m u_{\ell} b^{-\ell}, \quad u_{\ell} \in \mathbb{F}_b.$$

See [6, 15] for further details.

The theory for component-by-component construction of these rules is similar to that of the classical lattice rules. Here the components z_j of the generating vector are chosen from the set $(\mathbb{F}_b[x]/p)^*$, where the multiplier k is from the set $\mathbb{F}_b[x]/p$. As in the scalar case we can define a matrix Ω'_n as

$$\Omega'_n = \left[\omega \left(v_m \left(\frac{k(x) \cdot z(x)}{p(x)} \right) \right) \right]_{\substack{k(x) \in (\mathbb{F}_b[x]/p(x))^* \\ z(x) \in (\mathbb{F}_b[x]/p(x))^*}},$$

compare with (8). The product of $k(x)$ and $z(x)$ is done in the field $\mathbb{F}_b[x]/p$ and is thus modulo the irreducible polynomial $p(x)$ – this is the equivalence of the product modulo the prime n in the scalar case. Since the multiplicative group of every finite field is cyclic, we can find a primitive polynomial g which generates all of $\mathbb{F}_b[x]/p$ and thus we can transform the matrix Ω'_n into circulant form. This is all we need for the fast algorithm.

In [6] a digital Walsh kernel is used to define the function space. The 1-dimensional kernel defined for a given base b and smoothness α is given as

$$K_{1,\gamma}(x) = 1 + \gamma \phi_{\text{wal},b,\alpha}(x).$$

Here $\phi_{\text{wal},b,\alpha}$ is defined as

$$\phi_{\text{wal},b,\alpha} = \begin{cases} \mu_b(\alpha), & \text{if } x = 0, \\ \mu_b(\alpha) - b^{\lfloor \log_b(x) \rfloor (\alpha-1)} (\mu_b(\alpha) - 1), & \text{otherwise,} \end{cases}$$

and $\mu_b(\alpha) = b^{\alpha}(b-1)/(b^{\alpha}-b)$, see [6] for more details.

For polynomial lattice rules the logical choice is to take base $b = 2$ and then define a suitable power m to have a practical number of points $n = 2^m$. Since $b^{-\infty} = 0$ and $\log_b(0) = -\infty$, we thus have for a typical choice that $\omega(x) = \phi_{\text{wal},b,\alpha}$ is given as

$$\omega(x) = 2 - 2^{\lfloor \log_2(x) \rfloor}, \quad \text{with } b = 2, \alpha = 2.$$

5 Example Implementation in Matlab

In this section we give a possible implementation in Matlab of the fast algorithm for the scalar lattice rules. In this example implementation we will assume that the variable kernel function ω is symmetric around $1/2$, so that $\omega(x) = \omega(1 - x)$. Using this assumption we only need half of the \mathbf{q}' and $\boldsymbol{\psi}'$ vectors, and so the needed FFT's are only half the size. See [16] for a more thorough explanation.

```
function [z, e2] = fastrank1(n, s_max, omega, gamma, beta)

if ~isprime(n), error('n must be prime'); end;
z = zeros(s_max, 1);
e2 = zeros(s_max, 1);

m = (n-1)/2;          % assume the  $\omega$  function symmetric around 1/2
q = ones(m, 1);       % permuted product vector  $\mathbf{q}'$  (without zero index)
q0 = 1;               % zero index of permuted product vector:  $q(0)$ 
E2 = zeros(m, 1);     % the vector  $\tilde{E}^2$  in the text
cumbeta = cumprod(beta);

g = generator(n);     % generator  $g$  for  $\{1, 2, \dots, n-1\}$ 
perm = zeros(m, 1);   % permutation formed by positive powers of  $g$ 
perm(1) = 1; for j=1:m-1, perm(j+1) = mod(perm(j)*g, n); end;
perm = min(n - perm, perm); % map everything back to  $[1, n/2)$ 
psi = feval(omega, perm/n); % the vector  $\boldsymbol{\psi}'$ 
psi0 = feval(omega, 0); % zero index:  $\psi(0)$ 
fft_psi = fft(psi);

for s = 1:s_max
    % step 2a: circulant matrix-vector multiplication
    E2 = ifft(fft_psi .* fft(q));
    E2 = real(E2); % Matlab uses complex fft's: remove this noise
    % step 2b: choose  $w_s$  and  $z_s$  which give minimal value
    [min_E2, w] = min(E2); % pick index of minimal value
    if s == 1, w = 1; noise = abs(E2(1) - min_E2), end;
    z(s) = perm(w);
    % extra: we want to know the exact value of the worst-case error
    e2(s) = -cumbeta(s) + ( beta(s) * (q0 + 2*sum(q)) + ...
        gamma(s) * (psi0*q0 + 2*min_E2) )/n;
    % step 2c: update  $\mathbf{q}$ 
    q = (beta(s) + gamma(s) * psi([w:-1:1 m:-1:w+1])) .* q;
    q0 = (beta(s) + gamma(s) * psi0) * q0;
    fprintf('s=%4d, z=%4d, w=%4d, e2=%.5e, e=%.5e\n', ...
        s, z(s), w, e2(s), sqrt(e2(s)));
end;
```

The implementation we give is not the best possible. The FFT's could in fact be taken as real to complex transforms, instead of the complex to

complex transforms provided by Matlab. Likewise the IFFT could be taken as a complex to real transform. This functionality is not provided in Matlab, but could be exploited in a C implementation resulting in less memory usage and a faster construction.

Also note that in theory the vector \tilde{E}^2 should be constant in the first iteration. This means that we can always pick $z_1 = w_1 = 1$. In practice there will be numerical noise and so we force this decision. This immediately gives us some information about the noise level for the FFT, but nothing is done with this information in the example implementation. We did not list any code for the function `generator` which is used for finding a generator for \mathbb{Z}_n^* , but such an algorithm can be found in [2, page 25] and is relatively easy to implement.

We give an example usage of the routine for constructing a lattice rule for the average-case Sobolev space setting:

```
omega = inline('x.^2 - x + 1/6');
n = 4001; s = 100; gamma = 0.9.^[1:s];
[z, e2] = fastrank1(n, s, omega, gamma, 1+gamma/3);
```

For polynomial lattice rules the initialization part of the algorithm becomes only a little bit harder. We have to find a generator for $\mathbb{F}_b[x]/p$ and calculate all its positive powers, then find the Laurent series expansion for g^k/p and apply the functions v_m and ω . However, the core of the algorithm, which is the matrix-vector multiplication, remains the same.

6 Critical Remarks

We have shown that one simple implementation framework works for any shift-invariant kernel and that the algorithm is practically implementable by giving an example implementation in Matlab. This algorithm has construction cost $O(sn \log(n))$.

Although it is possible to do fast construction for a composite number of points, the results in [5, 4, 16] show that rules constructed with a prime number of points have better worst-case errors (i.e. the constant is better, while the rate is the same). We also would like to note that the prime number algorithm is a lot easier to implement and to understand. The reader is referred to [17] for details about the general algorithm. In the case of the polynomial lattice rules we arrive at an algorithm for a number of points of the form $n = b^m$, with b prime. Because of the parallels between prime numbers and irreducible polynomials, the presented algorithm is immediately applicable, although b^m is not prime.

In [16] it was shown that with a relatively standard PC configuration, lattice rules with a number of points almost up to 10^8 can be calculated by this algorithm. The question however is: how many points do we need? Or is it enough that we can generate the lattice rule on the fly for a moderate amount of points?

Depending on the number of points n , but also depending on the function space, the algorithm can exhibit numerical problems. It must be noted however that these numerical problems will manifest themselves a lot sooner when using the classical $O(sn^2)$ algorithm due to the significantly larger number of operations. An open question is therefore: can we improve the numerical qualities of the component-by-component algorithm?

Acknowledgements

This research is part of a project financially supported by the Onderzoeksfonds K.U.Leuven / Research Fund K.U.Leuven.

References

1. M. Abramowitz and I. A. Stegun, editors. *Handbook of Mathematical Functions with Formulas, Graphs and Mathematical Tables*, volume 55 of *National Bureau of Standards Applied Mathematics Series*. U.S. Government Printing Office, Washington, D.C., 1964.
2. H. Cohen. *A Course in Computational Algebraic Number Theory*. Graduate Texts in Mathematics. Springer-Verlag, 3rd edition, 1996.
3. P. J. Davis. *Circulant Matrices*. Wiley, 1979.
4. J. Dick and F. Kuo. Constructing good lattice rules with millions of points. In H. Niederreiter, editor, *Monte-Carlo and quasi-Monte Carlo Methods - 2002*, pages 181–197. Springer-Verlag, Jan. 2004.
5. J. Dick and F. Kuo. Reducing the construction cost of the component-by-component construction of good lattice rules. *Math. Comp.*, 73:1967–1988, 2004.
6. J. Dick, F. Kuo, F. Pillichshammer, and I. H. Sloan. Construction algorithms for polynomial lattice rules for multivariate integration. *Math. Comp.*, 2005. To appear.
7. J. Dick, G. Leobacher, and F. Pillichshammer. Construction algorithms for digital nets with small weighted star discrepancy. Report AMR04/13, School of Mathematics, University of New South Wales, Sydney, Apr. 2004.
8. M. Frigo and S. G. Johnson. FFTW: An adaptive software architecture for the FFT. In *Proc. 1998 IEEE Intl. Conf. Acoustics Speech and Signal Processing, Vol. 3*, pages 1381–1384. IEEE, 1998.
9. F. J. Hickernell. Lattice rules: How well do they measure up? In P. Hellekalek and G. Larcher, editors, *Random and Quasi-Random Point Sets*, pages 109–166. Springer-Verlag, 1998.
10. S. Joe. Component by component construction of rank-1 lattice rules having $O(n^{-1}(\ln(n))^d)$ star discrepancy. In H. Niederreiter, editor, *Monte-Carlo and quasi-Monte Carlo Methods - 2002*, pages 293–298. Springer-Verlag, Jan. 2004.
11. S. Joe. Construction of good rank-1 lattice rules based on the weighted star discrepancy. In H. Niederreiter and D. Talay, editors, *Monte Carlo and Quasi-Monte Carlo Methods 2004*. Springer-Verlag. This volume.
12. S. Joe and I. H. Sloan. On computing the lattice rule criterion R . *Math. Comp.*, 59(200):557–568, Oct. 1992.

13. F. Kuo. Component-by-component constructions achieve the optimal rate of convergence for multivariate integration in weighted Korobov and Sobolev spaces. *J. Complexity*, 19:301–320, June 2003.
14. H. Niederreiter. Quasi-Monte Carlo methods and pseudo-random numbers. *Bull. Amer. Math. Soc.*, 84(6):957–1041, Nov. 1978.
15. H. Niederreiter. *Random Number Generation and Quasi-Monte Carlo Methods*. Number 63 in Regional Conference Series in Applied Mathematics. SIAM, 1992.
16. D. Nuyens and R. Cools. Fast algorithms for component-by-component construction of rank-1 lattice rules in shift-invariant reproducing kernel Hilbert spaces. *Math. Comp.*, 2005. Accepted.
17. D. Nuyens and R. Cools. Fast component-by-component construction for (non-)primes. Submitted. Presented at Dagstuhl Seminar 04401: ‘Algorithms and Complexity for Continuous Problems’, 2004.
18. C. M. Rader. Discrete Fourier transforms when the number of data samples is prime. *Proc. IEEE*, 5:1107–1108, 1968.
19. I. H. Sloan, F. Kuo, and S. Joe. Constructing randomly shifted lattice rules in weighted Sobolev spaces. *SIAM J. Numer. Anal.*, 40(5):1650–1665, 2002.
20. I. H. Sloan and A. V. Reztsov. Component-by-component construction of good lattice rules. *Math. Comp.*, 71(237):263–273, 2002.