Uncertainty Quantification and Quasi-Monte Carlo

Sommersemester 2025

Vesa Kaarnioja
vesa.kaarnioja@fu-berlin.de

FU Berlin, FB Mathematik und Informatik

Seventh lecture, May 26, 2025

Let $f \in H_{s,\boldsymbol{\gamma}}$, where we assume that the positive weights $\boldsymbol{\gamma} := (\gamma_{\mathfrak{u}})_{\mathfrak{u} \subseteq \{1:s\}}$ have the following *product-and-order dependent (POD)* form

$$\gamma_{\mathfrak{u}} := \Gamma_{|\mathfrak{u}|} \prod_{j \in \mathfrak{u}} \gamma_j, \quad \mathfrak{u} \subseteq \{1:s\},$$

where $(\Gamma_k)_{k=0}^s$ and $(\gamma_j)_{j=1}^s$ are positive numbers such that $\Gamma_0 = 1$ and the empty product is interpreted as 1.

A randomly shifted rank-1 lattice rule with generating vector $\boldsymbol{z} \in \mathbb{N}^s$ satisfies the error bound

$$\sqrt{\mathbb{E}_{\boldsymbol{\Delta}} |I_s f - Q_{n,s}^{\boldsymbol{\Delta}} f|^2} \le e_{n,s}^{\mathrm{sh}}(\boldsymbol{z}) \|f\|_{s,\boldsymbol{\gamma}},$$

where the squared shift-averaged worst-case error in the weighted unanchored Sobolev space is given by the formula

$$[e_{n,s}^{\mathrm{sh}}(\boldsymbol{z})]^2 = \frac{1}{n} \sum_{k=0}^{n-1} \sum_{\varnothing \ne \mathfrak{u} \subseteq \{1:s\}} \gamma_{\mathfrak{u}} \prod_{j \in \mathfrak{u}} B_2\left(\left\{\frac{k z_j}{n}\right\}\right),$$

with $B_2(x) = x^2 - x + \frac{1}{6}$ denoting the Bernoulli polynomial of degree 2.

The components of the generating vector $\boldsymbol{z}$ can be restricted to the set

$$\mathbb{U}_n := \{z \in \mathbb{Z} \mid 1 \le z \le n \text{ and } \gcd(z, n) = 1\},$$

whose cardinality is given by the Euler totient function $\varphi(n) := |\mathbb{U}_n|$.

Component-by-component (CBC) construction.
Given n, s, and weights $(\gamma_{\mathfrak{u}})_{\mathfrak{u} \subseteq \{1:s\}}$, do

1. Set $z_1 = 1$.
2. With z_1 fixed, choose $z_2 \in \mathbb{U}_n$ to minimize $[e^{\mathrm{sh}}_{n,2}(z_1, z_2)]^2$.
3. With z_1 and z_2 fixed, choose $z_3 \in \mathbb{U}_n$ to minimize $[e^{\mathrm{sh}}_{n,3}(z_1, z_2, z_3)]^2$.

   $\vdots$

From the previous lecture, we know that the generating vector obtained using the CBC algorithm satisfies a certain *a priori* cubature error bound.

This week's lecture: *How to implement the CBC algorithm efficiently for POD weights and prime n?*

Remark: The so-called POD weights arise in the context of elliptic PDEs with random coefficients (next week's lecture), hence our interest in weights having this abstract form.

Our strategy will be as follows:

- First, we will describe a computationally inefficient version of the CBC algorithm. This will serve as a basis for a more efficient implementation.
- We will address the computational bottlenecks inherent in the naïve implementation of the CBC algorithm in order to construct an implementation of the so-called *fast CBC algorithm*.

For the fast CBC algorithm, we will require some sophisticated mathematical machinery (specifically, an algorithm for computing a primitive root modulo n and carrying out circulant matrix-vector multiplication using the fast Fourier transform), which will be discussed later on.

Naïve CBC construction

We write the error criterion as

$$[e_{n,d}^{\mathrm{sh}}(z_1,\dots,z_d)]^2 = \frac{1}{n}\sum_{k=0}^{n-1}\sum_{\varnothing\neq\mathfrak{u}\subseteq\{1:d\}}\gamma_{\mathfrak{u}}\prod_{j\in\mathfrak{u}}B_2\left(\left\{\frac{kz_j}{n}\right\}\right)$$

$$= \frac{1}{n}\sum_{k=0}^{n-1}\sum_{\ell=1}^{d}\underbrace{\sum_{\substack{|\mathfrak{u}|=\ell\\ \mathfrak{u}\subseteq\{1:d\}}}\gamma_{\mathfrak{u}}\prod_{j\in\mathfrak{u}}B_2\left(\left\{\frac{kz_j}{n}\right\}\right)}_{=:p_{d,\ell}(k)}.$$

By plugging in the POD weights $\gamma_{\mathfrak{u}} := \Gamma_{|\mathfrak{u}|}\prod_{j\in\mathfrak{u}}\gamma_j$, note that we have the following recursion (we split the sum over $\mathfrak{u}$ in two parts depending on whether $d\in\mathfrak{u}$):

$$p_{d,\ell}(k) = \sum_{\substack{|\mathfrak{u}|=\ell\\ \mathfrak{u}\subseteq\{1:d\}}}\Gamma_\ell\left(\prod_{j\in\mathfrak{u}}\gamma_j B_2\left(\left\{\frac{kz_j}{n}\right\}\right)\right)$$

$$= \sum_{\substack{|\mathfrak{u}|=\ell\\ \mathfrak{u}\subseteq\{1:d-1\}}}\Gamma_\ell\left(\prod_{j\in\mathfrak{u}}\gamma_j B_2\left(\left\{\frac{kz_j}{n}\right\}\right)\right)$$

$$+ \sum_{\substack{|\mathfrak{u}|=\ell-1\\ \mathfrak{u}\subseteq\{1:d-1\}}}\Gamma_\ell\gamma_d B_2\left(\left\{\frac{kz_d}{n}\right\}\right)\left(\prod_{j\in\mathfrak{u}}\gamma_j B_2\left(\left\{\frac{kz_j}{n}\right\}\right)\right)$$

$$= p_{d-1,\ell}(k) + \frac{\Gamma_\ell}{\Gamma_{\ell-1}}\gamma_d B_2\left(\left\{\frac{kz_d}{n}\right\}\right)p_{d-1,\ell-1}(k).$$

Plugging the recurrence

$$p_{d,\ell}(k) = p_{d-1,\ell}(k) + \frac{\Gamma_\ell}{\Gamma_{\ell-1}} \gamma_d B_2\left(\left\{\frac{k z_d}{n}\right\}\right) p_{d-1,\ell-1}(k)$$

into the expression for the squared shift-averaged WCE yields

$$\begin{aligned}
[e^{\mathrm{sh}}_{n,d}(z_1,\ldots,z_d)]^2 &= \frac{1}{n}\sum_{k=0}^{n-1}\sum_{\ell=1}^{d} p_{d,\ell}(k) \\
&= \frac{1}{n}\sum_{k=0}^{n-1}\sum_{\ell=1}^{d} p_{d-1,\ell}(k) + \frac{1}{n}\sum_{k=0}^{n-1}\sum_{\ell=1}^{d} \frac{\Gamma_\ell}{\Gamma_{\ell-1}} \gamma_d B_2\left(\left\{\frac{k z_d}{n}\right\}\right) p_{d-1,\ell-1}(k) \\
&= [e^{\mathrm{sh}}_{n,d-1}(z_1,\ldots,z_{d-1})]^2 + \frac{1}{n}\sum_{k=0}^{n-1} B_2\left(\left\{\frac{k z_d}{n}\right\}\right) \sum_{\ell=1}^{d} \frac{\Gamma_\ell}{\Gamma_{\ell-1}} \gamma_d p_{d-1,\ell-1}(k).
\end{aligned}$$

Recall that in the d^{th} step of the CBC algorithm, the components $z_1,\ldots,z_{d-1}$ are fixed and it is therefore sufficient to find $z_d \in \mathbb{U}_n$ which minimizes the expression $\sum_{k=0}^{n-1} B_2\left(\left\{\frac{k z_d}{n}\right\}\right) \sum_{\ell=1}^{d} \frac{\Gamma_\ell}{\Gamma_{\ell-1}} \gamma_d p_{d-1,\ell-1}(k)$.

Let us introduce the matrix $\Omega_n := \left[B_2\left(\left\{\frac{kz}{n}\right\}\right)\right]_{\substack{z\in\mathbb{U}_n \\ k\in\{0,\ldots,n-1\}}}$ and define a set of n-vectors recursively via

$$\boldsymbol{p}_{d,\ell} = \boldsymbol{p}_{d-1,\ell} + \gamma_d \frac{\Gamma_\ell}{\Gamma_{\ell-1}} \Omega_n(z_d, :) .* \boldsymbol{p}_{d-1,\ell-1}$$

starting from the initial values

$$\begin{aligned} \boldsymbol{p}_{d,0} &= \mathbf{1}_n \quad \text{for all } d \geq 1, \\ \boldsymbol{p}_{d,\ell} &= \mathbf{0}_n \quad \text{for all } d \geq 1 \text{ and } \ell > d, \end{aligned}$$

with $.*$ denoting the componentwise product between two vectors.

Then the value of $\sum_{k=0}^{n-1} B_2\left(\left\{\frac{kz_d}{n}\right\}\right) \sum_{\ell=1}^{d} \frac{\Gamma_\ell}{\Gamma_{\ell-1}} \gamma_d p_{d-1,\ell-1}(k)$ in the d^{th} step of the CBC algorithm can be obtained for all $z_d \in \mathbb{U}_n$ via

$$\Omega_n \boldsymbol{x}, \quad \text{where } \boldsymbol{x} = \sum_{\ell=1}^{d} \frac{\Gamma_\ell}{\Gamma_{\ell-1}} \gamma_d \boldsymbol{p}_{d-1,\ell-1}.$$

CBC algorithm – naïve version

1. Define the matrix $\Omega_n := \left[B_2\left(\left\{\frac{kz}{n}\right\}\right)\right]_{\substack{z\in\mathbb{U}_n \\ k\in\{0,\ldots,n-1\}}}$ and initialize the n-vectors

$$\boldsymbol{p}_{d,0} = \mathbf{1}_n \quad \text{for all } d \geq 1,$$
$$\boldsymbol{p}_{d,\ell} = \mathbf{0}_n \quad \text{for all } d \geq 1 \text{ and } \ell > d.$$

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

2. Pick the value $z_d \in \{1, \ldots, n-1\}$ corresponding to the smallest entry in the matrix-vector product

$$\Omega_n \boldsymbol{x}, \quad \text{where } \boldsymbol{x} = \sum_{\ell=1}^{d} \frac{\Gamma_\ell}{\Gamma_{\ell-1}} \gamma_d \boldsymbol{p}_{d-1,\ell-1}. \tag{1}$$

3. Update $\boldsymbol{p}_{d,\ell} = \boldsymbol{p}_{d-1,\ell} + \gamma_d \frac{\Gamma_\ell}{\Gamma_{\ell-1}} \Omega_n(z_d, :) .* \boldsymbol{p}_{d-1,\ell-1}$.

end for

Remarks: We only need the ratio $a_\ell := \frac{\Gamma_\ell}{\Gamma_{\ell-1}}$ for the implementation, e.g., for $\Gamma_\ell = \ell!$ this is $a_\ell = \ell$. The computational bottleneck is the dense matrix-vector product $\Omega_n \boldsymbol{x}$ in (1), which has complexity $\mathcal{O}(n^2)$. The *fast CBC algorithm* reduces this product down to $\mathcal{O}(n \log n)$ complexity.

Fast CBC algorithm

What makes fast CBC fast?

The matrix-vector product $\Omega_n \boldsymbol{x}$ has time complexity $\mathcal{O}(n^2)$, which is too slow if n is, say, of the order of a million or more. (Not to mention the problem of storing a dense matrix of such size!)

However, the matrix Ω_n has a lot of structure. It turns out that we can implement the matrix-vector product $\Omega_n \boldsymbol{x}$ in $\mathcal{O}(n \log n)$ time using some sophisticated mathematical tools.

In a nutshell, we let $n \geq 3$ be *prime* and do the following:

- Using some natural symmetries of Ω_n, we can ignore the first column (since it corresponds to shifting the objective functional in the CBC minimization step by a constant value) and it will be sufficient to consider only the top-left block $\Omega'_n := \Omega_n(1 : \mathtt{m}, 2 : \mathtt{m} + 1)$, where $m := (n - 1)/2$.
- For *prime* n, we can find a *generator* g (primitive root modulo n) and use this to permute Ω'_n into a circulant matrix.
- A circulant matrix implements a circular convolution, so a matrix-vector product (in the permuted indexing) can be implemented in $\mathcal{O}(n \log n)$ time using the fast Fourier transform (FFT).

Before getting to the implementational details of fast CBC, we will need to

- discuss an algorithm to find a primitive root modulo n;
- discuss how to compute a circulant matrix-vector product using FFT.

Primitive root modulo n

Definition

Let $g, n \in \mathbb{N}$. The number g is called a *primitive root modulo n* if for any integer $a \in \mathbb{N}$ such that $\gcd(a, n) = 1$, there exists an integer k (called the *index*) such that

$$g^k \equiv a \pmod{n}.$$

Such a number g is the *generator* of the multiplicative group of integers modulo n, i.e., $(\mathbb{Z}/n\mathbb{Z})^\times$.

Theorem (Gauss 1801)

A primitive root modulo n exists if and only if

- n *is* 1, 2, 4, *or*
- $n = p^k$, *where* $p \geq 3$ *is a prime and* $k \in \mathbb{N}$, *or*
- $n = 2p^k$, *where* $p \geq 3$ *is a prime and* $k \in \mathbb{N}$.

Note especially that a primitive root modulo n exists whenever n is prime.

Recall that the Euler totient function is defined by $\varphi(n) := |\{k \in \mathbb{N} \mid 1 \leq k \leq n,\ \gcd(k, n) = 1\}|$. We have the following.

Proposition

The number g is a primitive root modulo n if and only if the smallest positive integer k for which $g^k \equiv 1 \pmod{n}$ is precisely $k = \varphi(n)$.

Lagrange's theorem: the smallest k satisfying $g^k \equiv 1 \pmod{n}$ divides $\varphi(n)$. Therefore, it is enough to check for all proper divisors $d|\varphi(n)$ that $g^d \not\equiv 1 \pmod{n}$.

However, we can do even better!

Find the prime number factorization $\varphi(n) = p_1^{a_1} \cdots p_\ell^{a_\ell}$. It turns out that it is enough to check that $g^d \not\equiv 1 \pmod{n}$ for all $d \in \left\{ \frac{\varphi(n)}{p_1}, \ldots, \frac{\varphi(n)}{p_\ell} \right\}$. To see this, let d be any proper divisor of $\varphi(n)$. Then there exists j such that $d | \frac{\varphi(n)}{p_j}$, meaning that $dk = \frac{\varphi(n)}{p_j}$ for some $k \in \mathbb{N}$. However, if $g^d \equiv 1 \pmod{n}$, we would get

$$g^{\frac{\varphi(n)}{p_j}} \equiv g^{dk} \equiv (g^d)^k \equiv 1^k \equiv 1 \pmod{n}.$$

That is, if g was not a primitive root, then one could find a number of the form $\frac{\varphi(n)}{p_j}$ for which $g^{\frac{\varphi(n)}{p_j}} \equiv 1 \pmod{n}$.

$\therefore$ It is enough to check that $g^{\frac{\phi(n)}{p_j}} \not\equiv 1 \pmod{n}$ for all $j \in \{1, \ldots, \ell\}$.

Algorithm for finding a primitive root modulo n

1. Find the prime number factorization $\varphi(n) = p_1^{a_1} \cdots p_\ell^{a_\ell}$.

   Iterate through all numbers $g = 1, 2, \ldots, n-1$ and, for each number, check whether it is a primitive root by doing the following:

   2. Calculate $\mathrm{mod}(g^{\frac{\varphi(n)}{p_j}}, n)$ for all $j \in \{1, \ldots, \ell\}$.
   3. If all the calculated values are different from 1, then g is a primitive root.

Remark: In Python, the quantities in step 2 can be computed, e.g., via `pow(g,sympy.totient(n)/p_j,n)`

Discrete and fast Fourier transform

The *discrete Fourier transform* of (complex) vector $\boldsymbol{x} := (x_j)_{j=1}^n$ is defined as the vector $\boldsymbol{y} := (y_j)_{j=1}^n$ with

$$y_j = \sum_{k=1}^{n} x_k \mathrm{e}^{-2\pi \mathrm{i}(j-1)(k-1)/n}, \quad j \in \{1, \ldots, n\},$$

and the *inverse discrete Fourier transform* is given by

$$x_j = \frac{1}{n} \sum_{k=1}^{n} y_k \mathrm{e}^{2\pi \mathrm{i}(j-1)(k-1)/n}, \quad j \in \{1, \ldots, n\}.$$

The *fast Fourier transform (FFT)* can be used to carry out these operations in $\mathcal{O}(n \log n)$ time. In Python, one has $\boldsymbol{y} = \texttt{numpy.fft.fft}(\boldsymbol{x})$ and $\boldsymbol{x} = \texttt{numpy.fft.ifft}(\boldsymbol{y})$.

Circular convolution

Let $\boldsymbol{x} := (x_i)_{i=1}^n$ and $\boldsymbol{y} := (y_i)_{i=1}^n$ be (complex) vectors. Then the sequence $\boldsymbol{z} := (z_i)_{i=1}^n$ defined by

$$z_i = \sum_{k=1}^{n} x_k y_{\mathrm{mod}(i-k,n)+1}, \quad i \in \{1, \ldots, n\},$$

is called the *circular convolution* of $\boldsymbol{x}$ and $\boldsymbol{y}$ and we denote it by $\boldsymbol{z} := \boldsymbol{x} \star \boldsymbol{y}$.

Similarly to the continuous convolution, we have the following identity using discrete/fast Fourier transform:

$$\texttt{fft}(\boldsymbol{x} \star \boldsymbol{y}) = \texttt{fft}(\boldsymbol{x}).*\texttt{fft}(\boldsymbol{y}),$$

where $\boldsymbol{x}.*\boldsymbol{y} := (x_i y_i)_{i=1}^n$ is the pointwise product of two vectors.

Circular convolution and circulant matrices

A matrix $A \in \mathbb{R}^{n \times n}$ is called *circulant* if it has the form

$$A = \begin{bmatrix} a_0 & a_{n-1} & \cdots & a_2 & a_1 \\ a_1 & a_0 & a_{n-1} & & a_2 \\ \vdots & a_1 & a_0 & \ddots & \vdots \\ a_{n-2} & & \ddots & \ddots & a_{n-1} \\ a_{n-1} & a_{n-2} & \cdots & a_1 & a_0 \end{bmatrix}.$$

- Each row is equal to the row above shifted to the right by one (wrapping around the edge in a periodic way).
- The first column/row contains all information about the matrix.
- A circulant matrix implements a circular convolution:

$$A\boldsymbol{x} = \boldsymbol{a} \star \boldsymbol{x}, \tag{2}$$

  where $\boldsymbol{a} := [a_0, a_1, \ldots, a_{n-1}]^{\mathrm{T}}$ is the first column of matrix A.
- The identity (2) implies that a circulant matrix-vector product can be implemented in $\mathcal{O}(n \log n)$ time as $A\boldsymbol{x} = \texttt{ifft}(\texttt{fft}(\boldsymbol{a}).*\texttt{fft}(\boldsymbol{x}))$.

Putting it all together

The matrix-vector product $\Omega_n \boldsymbol{x}$ in the CBC loop costs $\mathcal{O}(n^2)$ operations. However, it was shown by Kuo, Nuyens, and Cools (2006) that the blocks of Ω_n can be permuted into circulant form $\rightarrow$ the matrix-vector product can be implemented in $\mathcal{O}(n \log n)$ operations using FFT.

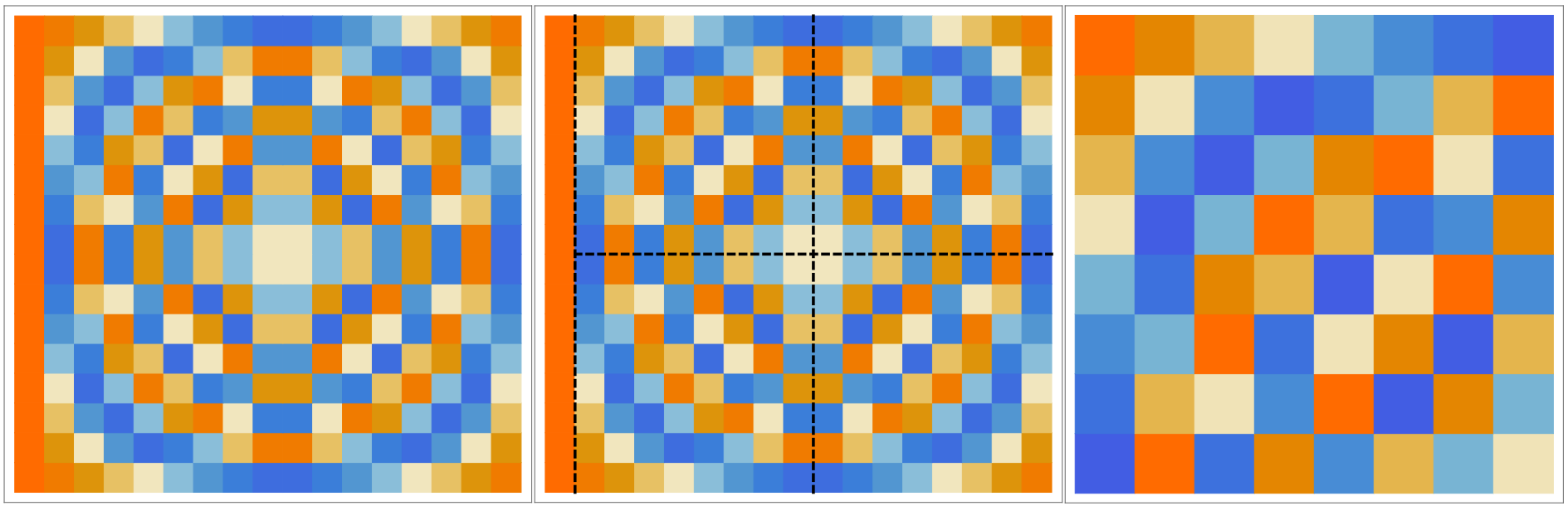

Figure: Example with Ω_{17}. Note that the first column is a constant and can be left out (the components of $\Omega_n \boldsymbol{x}$ are shifted by a constant $\rightarrow$ the smallest component stays invariant). Noting the obvious symmetries in the remaining four blocks, we can focus on the top left block.

When n is prime, it is possible to use the so-called Rader transformation to permute the block matrices into circulant form. The permutation matrices can be easily obtained by computing the generator, i.e., primitive root modulo n.

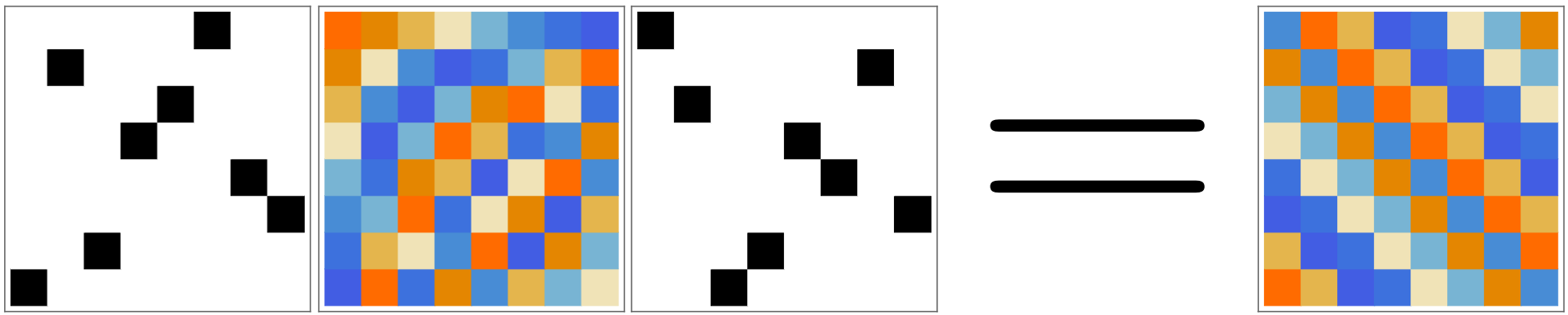

Figure: The original block matrix is multiplied from both sides by Rader permutation matrices (the black elements indicate the value 1 and white elements indicate the value 0) to obtain a circulant matrix.

Example with $n = 1009$

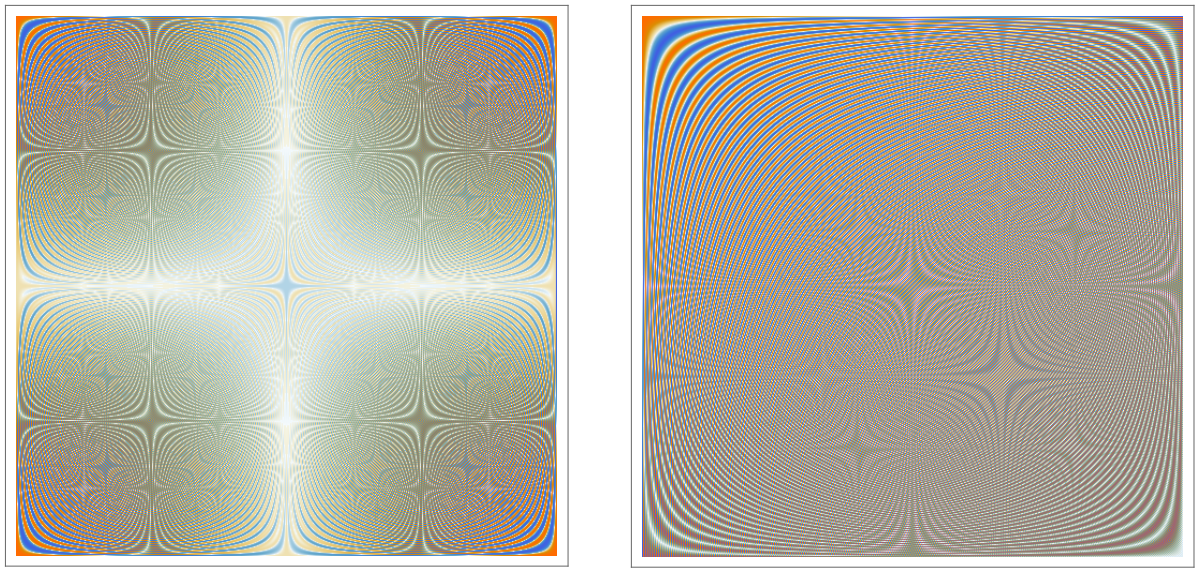

Figure: LHS: Original Ω_{1009}. RHS: top left block of Ω_{1009} (sans first column).

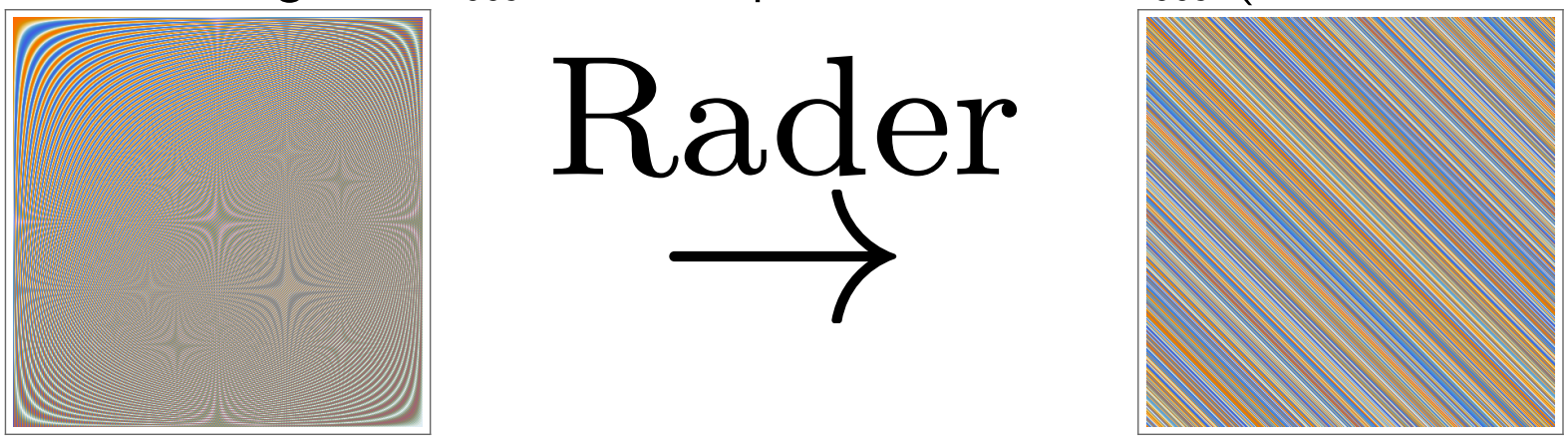

Figure: Rader transformation turns the top left block matrix circulant.

Python implementation given in the file `fastcbc.py` available on the course webpage!

- The overall cost of the CBC algorithm with POD weights is $\mathcal{O}(s\,n \log n + s^2 n)$.
- For simplicity, we considered only the case where n is prime. An extension for composite n was discussed by Nuyens and Cools (J. Complexity 2006). The idea for composite n is that the complete matrix Ω_n can be partitioned in blocks which have a circulant or block-circulant structure. The special case of n being a power of 2 has been discussed by Cools, Kuo, and Nuyens (SIAM J. Sci. Comput. 2006).
- There also exist freely available software implementing the fast CBC construction, cf., e.g., https://people.cs.kuleuven.be/~dirk.nuyens/qmc4pde/, https://people.cs.kuleuven.be/~dirk.nuyens/fast-cbc/, https://qmcpy.org/, ...