Contents lists available at ScienceDirect

Computer Physics Communications

journal homepage: www.elsevier.com/locate/cpc

Computational Physics

Strong simulation of linear optical processes ☆

Nicolas Heurtel *, Shane Mansfield, Jean Senellart, Benoît Valiron

ARTICLE INFO

Article history:
Received 1 September 2022
Received in revised form 27 June 2023
Accepted 28 June 2023
Available online 4 July 2023

Keywords:
Quantum computing
Linear optics
Strong simulation

ABSTRACT

In this paper, we provide an algorithm and general framework for the simulation of photons passing through linear optical interferometers. Given n photons at the input of an m-mode interferometer, our algorithm computes the probabilities of all possible output states with time complexity $O\left(n\binom{n+m-1}{m-1}\right)$, linear in the number of output states $\binom{n+m-1}{m-1}$. It outperforms the permanent-based method by an exponential factor, and for the restricted problem of computing the probability for one given output it improves the time complexity over the state-of-the-art for the permanent of matrices with multiple rows or columns, with a tradeoff in the memory usage. Our algorithm also has additional versatility by virtue of its use of memorisation – the storing of intermediate results – which is advantageous in situations where several input states may be of interest. Additionally it allows for hybrid simulations, in which outputs are sampled from output states whose probability exceeds a given threshold, or from a restricted set of states. We consider a concrete, optimised implementation, and we benchmark the efficiency of our approach compared to existing tools.

1. Introduction

In quantum computation one encodes information into the states of quantum systems – photons, atoms, ions, etc. – which can then processed by evolving and manipulating those systems according to the laws of quantum mechanics. It is by now well-known that the paradigm opens vast possibilities for exploiting non-classical behaviours available to quantum systems in order to process information in radically new ways that can lead to a variety of *quantum advantages* including computational speedups [1], enhanced security [2], more efficient communication [3], and the potential for reduced energy consumption [4,5], when compared to classical information processing.

The development quantum technologies aiming to leverage such advantages have been advancing at pace over the past number of years. A variety of different hardwares, each using different physical supports for the quantum information, are being pursued. Among these, photonic hardware has a privileged role in the sense that regardless of hardware choice it will eventually be necessary to network quantum processors, and as the only viable support for communicating quantum information it is inevitable that some quantum information must eventually be treated photonically. Photons have a number of other desirable features too, including an absence of decoherence in transparent media – i.e. a capacity to reliably maintain their quantum states –, reduced cryogenic requirements compared to other hardware approaches, and good prospects for scalability due to compatibility with the existing semiconductor industry [6].

Photonic quantum technologies consisting of single-photon sources, which are coupled to linear optical interferometers – which may be parametrisable and take the form of integrated circuits –, which are coupled in turn to photon detectors, offer a promising route to implementing quantum computation. They enable both non-universal models of quantum computation [7], which have led to laboratory demonstrations that claim to show quantum computational advantages with today's technology [8,9], as well as models for achieving universal [10], and fault-tolerant quantum computation [11].

To accompany the technological developments in photonic quantum computing it is also important to have access to tools for design, testing, and experimenting with algorithms, protocols, and schemes. In this respect, the classical simulation of photonic quantum computing platforms has become an increasingly important problem. Of course, one of the main interests of quantum computation is that it quickly becomes unfeasible for classical processors to simulate. Yet there are clear benefits to achieving optimal classical simulation within the theoretical limits. This can aid in designing and perfecting interferometers that generate specific logic gates, entangled states, and other building-block components of quantum computers. It can provide both development and verification tools for algorithm and software development. Furthermore

☆ The review of this paper was arranged by Prof. W. Jong.
* Corresponding author.
E-mail address: nicolas.heurtel@quandela.com (N. Heurtel).

https://doi.org/10.1016/j.cpc.2023.108848
0010-4655/© 2023 Elsevier B.V. All rights reserved.

it can help to define the performance boundaries separating the quantum from the classical computational paradigm.

Two special cases are of particular interest: strong simulation, where a classical program computes the complete quantum state obtained at the outputs of a photonic circuit; and weak simulation, or sampling, where the classical program emulates the probabilistic behaviour that would be observed at the outputs. The task of predicting, with classical algorithms and computers, the output states for photons passing through interferometers relates to the $\#P$-hard problem of calculating the permanents of complex matrices [12] associated with the interferometer [13].

In order to perform strong simulation, one solution is therefore to reuse algorithms originally designed for computing permanents. The state-of-the-art classical algorithms for computation of the permanent of a complex matrix are those due to Ryser [14] and Glynn [15], while the state-of-the-art classical algorithms for boson sampling (essentially the weak simulation problem) are those due to Clifford and Clifford [16,17]. As the strong simulation of linear optical circuits involves the permanent computation of matrices with repeated rows or columns, algorithms with a better complexity [18] than Ryser's or Glynn's can be used.

On the other hand and independently from permanents, the folklore of linear optics is aware of informal, back-of-the-envelope techniques for running fixed-photon number simulations. The goal of this paper is to formalize and analyze these folklore techniques and provide a comparison with the algorithms for the permanents of complex matrices. In particular, we propose a framework for strong simulation, amenable to weak simulation, which additionally allow for hybrid forms of simulation between these two particular cases. Accross the paper, we shall be using the acronym SLOS, standing for *Strong Linear Optical Simulator*. We derive an algorithm for computing the permanent of a complex matrix with repeated rows that improves the state-of-the-art [18] time complexity, with a tradeoff in memory. We detail optimised implementations of our algorithms that can be found in the *QuandeLibC* library[1] (which is also integrated in the open-source software platform *Perceval* [19]). We benchmark the performance of our algorithms, comparing them to implementations of the Glynn and Ryser algorithms in *the Walrus* library [20], as well as implementations in *QuandeLibC* of these algorithms, which appear to be more efficient, and of the Clifford and Clifford algorithms also in *QuandeLibC*.

Contributions A preliminary version of this work has been presented at IEEE Quantum Computing and Engineering [21]. The main contributions of this paper are three Strong Linear Optical Simulators (`SLOS`), and can be summarised as follows:

- An algorithm, labelled `SLOS_full`, for computing the full output amplitude distribution of a linear optical circuit for a given input. Although this method would be naturally described in physics textbooks, to the authors' knowledge this is the first complexity study and explicit implementation of that method. The time complexity is $O\left(n\binom{n+m-1}{m-1}\right)$, and so is linear in the number of output states $\binom{n+m-1}{m-1}$. The full distribution can be obtained in an optimal space of $O(\binom{n+m-1}{m-1})$.
- A generalised strong simulation algorithm, labelled `SLOS_gen`, for computing the amplitudes of any set of outputs from any set of inputs. For one input, `SLOS_gen` has the same time complexity as `SLOS_full` for the full output distribution, and improves the state-of-the-art [18] for the specific case of a single output, giving a new upper bound for computing a permanent with repeated rows or columns.
- A hybrid algorithm that can combine both weak and strong simulation for many inputs and outputs, labelled `SLOS_hyb`. `SLOS_hyb` can sample from a set of outputs whose probability exceeds a given threshold, or even sample among a restricted set of states. `SLOS_hyb` can both perform both weak sampling and strong simulation (as `SLOS_gen`).
- Detailed optimised implementations of both `SLOS_full` and `SLOS_gen`. The implementations are open-source and available in the *QuandeLibC* library.
- Practical performance benchmarking of the algorithm in a generic example of a quantum machine learning algorithm [22], where it is seen to give a considerable practical edge over the permanent-based approach to simulation.

Plan The paper is structured as follows: in Section 2 we provide some background on linear optical simulation, and in particular the specific problems we focused on are set up in Section 2.4, with illustrations of typical use cases. In Section 3 we introduce the algorithms and their complexity analysis, summarised in Table 1. The practical implementation and optimisation are presented in Section 4, while benchmarks with the permanent-based model are in Section 5. We finally conclude and discuss in Section 6.

2. Simulating linear optical circuits

In Section 2.1, we set up some formalism and notational conventions to be used throughout the paper. After briefly explaining the hardness of LO-circuit simulation in Section 2.2, we present and define the weak and strong simulation problems in Section 2.3. In Section 2.4, we present the two strong simulation problems that we propose to answer in Section 3.

2.1. Formalism of linear optical circuits and notation

Throughout this paper, we will be considering n indistinguishable photons over m modes. Typically these are spatial modes, but they could in principle also correspond to other discrete degrees of freedom such as polarisation, frequency, or time-bins [23,24]. States of the system will be Fock states or their superpositions and we write $|s_1, s_2, \dots, s_m\rangle$ to denote the Fock state with s_i photons in mode i. Sometimes it will be interesting to consider states containing less than n photons, so we introduce the notation $|s_1, s_2, \dots, s_m\rangle^k$ to describe a state with $\sum_{i=1}^m s_i = k$ photons, sometimes shortening this to $|s\rangle_m^k$. The vacuum state, with no photon in m modes, will be denoted as $|0\rangle_m$. We also introduce $\mathcal{F}_m^k$ as the set of the Fock states of k photons into m modes, so that $\mathcal{F}_m^k = \left\{ |s_1, s_2, \dots, s_m\rangle^k \middle| s_i \in \mathbb{N} \right\}$. It is known that $\#\mathcal{F}_m^k = \binom{k+m-1}{m-1}$, as it is exactly the number of ways to put k indistinguishable balls into m distinguishable bins [25]. For readability and as m will be a fixed parameter, that number will be denoted as M_k.

It is standard in the second quantisation formalism to associate each mode i with a creation operator $\hat{a}_i^\dagger : \mathcal{F}_m^k \to \mathcal{F}_m^{k+1}$ acting as follows:

$$\hat{a}_i^\dagger \, |s_1, \dots, s_i, \dots, s_m\rangle^k = \sqrt{s_i + 1}\, |s_1, \dots, s_i + 1, \dots, s_m\rangle^{k+1}$$

This paper focuses only on linear optical operations, for which the transformations on the creation operators are described by a unitary matrix $U = (u_{i,j})$ of size $m \times m$ such that $\hat{a}_p^\dagger \mapsto \sum_{i=1}^m u_{i,p}\, \hat{a}_i^\dagger$. As shown in [26], every such unitary can be implemented by a linear optical circuit of m spatial modes, with only phase shifters and beamsplitters [23] as linear optical components. We will call such a circuit an LO-circuit.

Each unitary matrix $U : m \times m$ acting on the vector of creation operators can be associated with a unitary operator $\mathrm{U}_{\mathcal{F}}$ on

[1] On Github at https://github.com/Quandela/QuandeLibC.

$\mathcal{H}_m = \bigoplus_{k=0}^{+\infty} \mathcal{H}_m^k$ where $\mathcal{H}_m^k$ is the Hilbert space generated by the elements of $\mathcal{F}_m^k$ [7,13]. $\mathrm{U}_{\mathcal{F}} \left|s\right\rangle_m^k$ will represent the state obtained when the state $\left|s\right\rangle_m^k$ is the input of an LO-circuit implementing U and can be obtained with:

$$\mathrm{U}_{\mathcal{F}} \left|s\right\rangle_m^n = \prod_{p=1}^{m} \frac{1}{\sqrt{s_p!}} \left(\sum_{i=1}^{m} u_{i,p}\, \hat{a}_i^\dagger \right)^{s_p} \left|0\right\rangle_m \tag{1}$$

Following our notation, ${}^n_m\langle t|\mathrm{U}_{\mathcal{F}} \left|s\right\rangle_m^n$ is the amplitude assigned to the state $\left|t\right\rangle_m^n$ within the overall output state $\mathrm{U}_{\mathcal{F}} \ \left|s\right\rangle_m^n$.

2.2. Hardness of linear optical simulations

Given a matrix $M : n \times n$, the permanent of M is defined as follows:

$$\mathrm{Perm}(M) = \sum_{\sigma \in S_n} \prod_{i=1}^{n} m_{i,\sigma(i)}.$$

Efficiently computing the permanent is crucial in evaluating linear optical transformations, as we can show that:

$${}^n_m\langle t|\mathrm{U}_{\mathcal{F}} \left|s\right\rangle_m^n = \frac{\mathrm{Perm}\left(U_{|s\rangle,|t\rangle}\right)}{\sqrt{s_1!\ldots s_m! t_1!\ldots t_m!}}, \tag{2}$$

where $U_{|s\rangle,|t\rangle}$ is obtained from the unitary U by repeating s_i times its i^{th} column and t_j times its j^{th} row [27,28]. We return to this in Section 2.4.

It is known that computing a permanent of a general complex matrix [12] or even subclasses of real orthogonal matrices [29] is a $\#P$-hard problem. The fastest known algorithms [14,15] for computing a permanent of a general matrix of size n, with some pre-computation allowed [30], are in $O(n2^n)$.

For computing the permanent in Equation (2), the redundancy of the rows and columns in $U_{|s\rangle,|t\rangle}$ can be taken advantage of, giving a faster algorithm [18]. More generally, for an output state $|t_1, \ldots, t_m\rangle$, the permanent can be computed in:

$$O\left(n \frac{\prod_{i=1}^{m}(t_i+1)}{\min\limits_{t_l \neq 0}(t_l+1)} \right) \tag{3}$$

operations. Note that this bound is equal to $O\left(n2^{n-1}\right)$ in the worst case, when there is at most one photon per mode, and equal to $O\left(n\right)$ in the best case, when only one mode is occupied.

2.3. Weak and strong simulation

When considering a model for a biased coin, two strategies can be followed. Either one can try to literally emulate the probabilistic behaviour of the coin and have a protocol answering "head" or "tail" with the appropriate probabilities, or to more fully characterise the behaviour and by listing the (two) precise probabilities for "head" and "tail". The former approach is called *weak simulation* while the latter is *strong simulation*.

Weak simulation Weak simulation of LO-circuits is the classical sampling from their output distribution, also known as the Boson Sampling problem [7]. Given an input $\left|s\right\rangle_m^n$ and a LO-circuit implementing a unitary U, we would like to sample an output $\left|t\right\rangle_m^n$ from the distribution $\mathcal{D}_U(s) = \left\{ \left|\langle t|\mathrm{U}_{\mathcal{F}} \left|s\right\rangle\right|^2, \left|t\right\rangle \in \mathcal{F}_m^n \right\}$.

Under some assumptions, weak simulation has been shown to be classically hard [7], as it would imply $P^{\#P} = BPP^{NP}$ leading to a collapse of the polynomial hierarchy of complexity classes to the third level. Therefore, Boson Sampling is a good candidate for quantum advantage, as a linear optical computer with single-photon inputs can naturally sample from $\mathcal{D}_U(s)$. It was shown that weak simulation could be done in $O(n2^n + mn^2)$ [16], which was further improved to $O(n1.69^n)$ on average when $m = n$ [17] thanks to a more efficient way to compute permanents with repeated rows.

Even though weak simulation is not the main focus of the paper, it can be recovered as a by-product of the general algorithm `SLOS_hyb` presented in Section 3.3.

Strong simulation Strong simulation of LO-circuits is the classical computation of the output amplitudes (or the probabilities): Given an input $\left|s\right\rangle_m^n$ and an LO-circuit implementing a unitary U, we would like to compute the amplitudes $\langle t|\mathrm{U}_{\mathcal{F}} \left|s\right\rangle$, or the probabilities $\left|\langle t|\mathrm{U}_{\mathcal{F}} \left|s\right\rangle\right|^2$ with $\left|t\right\rangle \in \mathcal{F}_m^n$.

As explained in Section 2.2, we can directly compute them by computing the permanents of $U_{|s\rangle,|t\rangle}$. Therefore, the complexity of computing one amplitude or probability is exactly the complexity of computing one permanent.

In this paper, we propose a procedure directly computing the amplitudes of several outputs or inputs, which is more efficient than computing them separately and independently, cf. Sections 3.1 and 3.2, only needing $O\left(\sum_{i=1}^{m} t_i \prod_{j\neq i}(t_j+1)\right)$ operations for one output, as shown in Section 3.2.3 and summarised in Table 1. We therefore can efficiently solve two kinds of strong simulation problems, which we set out in Section 2.4.

2.4. `SLOS` problems: two strong linear optical simulation problems

This section will introduce the two classes LO simulation problems summarised in Fig. 1. For each class, we provide concrete examples and typical use-cases. Each problem has a proposed solution described in Section 3 and summarised in Fig. 2.

Problem 1: full amplitude list simulation Given an input $\left|s\right\rangle_m^n$ and an LO-circuit of m modes implementing a unitary matrix $U : m \times m$, what is the output state $\mathrm{U}_{\mathcal{F}} \ \left|s\right\rangle_m^n$? Equivalently, what is the full amplitude list $\left\{\langle t|\mathrm{U}_{\mathcal{F}} \left|s\right\rangle, \left|t\right\rangle \in \mathcal{F}_m^n\right\}$? This situation is schematised in Fig. 1a.

Example For instance, we can consider an LO-circuit of three modes implementing a unitary U, with an input of two photons $\left|1,1,0\right\rangle$. In that problem, we would like to compute every output state of $\mathcal{F}_3^2$, meaning we would like to know the output:

$$\begin{aligned}\mathrm{U}_{\mathcal{F}} \left|1,1,0\right\rangle &= \alpha_1 \left|2,0,0\right\rangle + \alpha_2 \left|0,2,0\right\rangle + \alpha_3 \left|0,0,2\right\rangle + \alpha_4 \left|1,1,0\right\rangle \\ &\quad + \alpha_5 \left|1,0,1\right\rangle + \alpha_6 \left|0,1,1\right\rangle.\end{aligned}$$

Notice that we can compute each amplitude α_i of an output state $\left|t\right\rangle$ by computing the permanent of $U_{|1,1,0\rangle,|t\rangle}$, as defined in Section 2.2. To compute α_3, we would need to compute $\mathrm{Perm}(U_{|1,1,0\rangle,|0,0,2\rangle})$. By taking the first and second column of U, $\begin{pmatrix} u_{1,1} & u_{1,2} \\ u_{2,1} & u_{2,2} \\ u_{3,1} & u_{3,2} \end{pmatrix}$ and repeating two times the third row to form a $U_{|1,1,0\rangle,|0,0,2\rangle}$, we have:

$$\begin{aligned}\alpha_3 &= \frac{\mathrm{Perm}(U_{|1,1,0\rangle,|0,0,2\rangle})}{\sqrt{2!}} = \frac{\mathrm{Perm}\begin{pmatrix} u_{3,1} & u_{3,2} \\ u_{3,1} & u_{3,2} \end{pmatrix}}{\sqrt{2}} \\ &= \sqrt{2} \times u_{3,1} \times u_{3,2}\end{aligned}$$

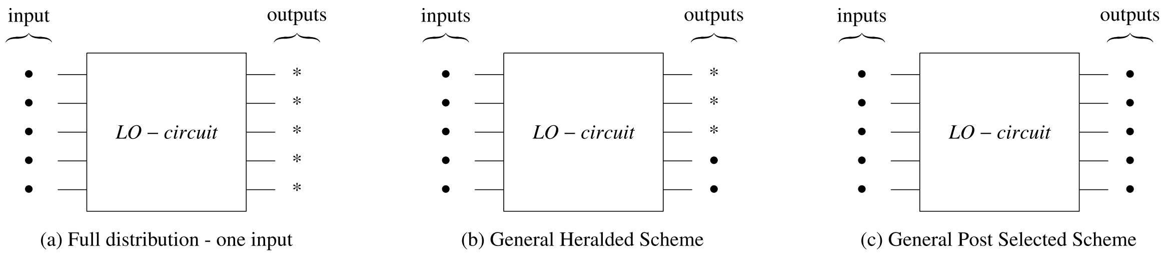
input
outputs
LO – circuit
(a) Full distribution - one input
inputs
outputs
LO – circuit
(b) General Heralded Scheme
inputs
outputs
LO – circuit
(c) General Post Selected Scheme

Fig. 1. Application Schemes of Problem 1 and Problem 2. $\bullet$ is a chosen number of photons, while $*$ is every possible number.

Typical use-cases As the full amplitude list is a complete description of an LO-circuit and the input can be directly the state right after the photon sources, applications for this problem can be both practical and theoretical. One can think of:

- Simulate or look for circuits preparing a specific output distribution like entangled high-dimensional states [31] or Bell Measurements [32].
- Check the correctness and noise of experimental circuits with the theoretical distribution. One can certify the correctness of Boson sampling by computing metrics (such as total variational distance, bunching probabilities,...) from the statistics obtained [33–36].
- Machine Learning algorithms as the approximation of differential equations of [22], application detailed in Section 5.1.

Problem 2: generic strong simulation Given a set of inputs $\mathcal{I}$, a set of outputs $\mathcal{O}$ and an LO-circuit of m modes implementing a unitary matrix $U : m \times m$, what are the output amplitudes $\left\{\langle t|\mathrm{U}_{\mathcal{F}}\,|s\rangle\,,\,|s\rangle \in \mathcal{I}, |t\rangle \in \mathcal{O}\right\}$?

Typical use-cases Often, only specific outcomes are of interest, so the full output distribution of a LO-circuit is not needed. Also, for every circuit encoding logic gates or functions in general, we need to know the effect of the LO-circuit on each different possible input, so several inputs are needed. Therefore, that problem is very general and tackles various applications, in which we can highlight two general classes of schemes.

- Post-selected scheme. Presented in Fig. 1c, it consists of only considering —or selecting— some inputs and some outputs according to some criteria. A canonical example is the implementation of the 2-qubit CNOT-gate of [37]. Working on 6 modes with 2 photons, the considered input and output states are $X = \{|0,1,0,1,0,0\rangle\,, |0,1,0,0,1,0\rangle\,, |0,0,1,1,0,0\rangle\,, |0,0,1,0,1,0\rangle\}$. In other words, the circuit is regarded as a map restricted on the 4-dimensional subspace $\mathbb{C}[X]$ generated by X, and not the full 21-dimensional $\mathcal{H}_6^2$ space. The term "postselected" refers to the fact that in general, the output state might have a component orthogonal to $\mathbb{C}[X]$ (for instance, maybe the state $|0,2,0,0,0,0\rangle$ has a non-zero amplitude). Such orthogonal components are considered bogus. They are ruled out at the end of the computation, when measuring the system.
- Heralded scheme. Presented in Fig. 1b, it consists in designing a LO-circuit that can fail, but for which the failure can be decided upon the measurement result of some of the output modes. For instance, in [38] a scheme is proposed to implement the 2-qubit CZ-gate. The 4 input modes can accommodate 2, 3 or 4 photons, and the input state is generated with $\{|1,1,1,1\rangle\,, |1,0,1,1\rangle\,, |0,1,1,1\rangle\,, |0,0,1,1\rangle\}$. For the output, we only consider the cases where the two last modes contain a photon: $|*,*,1,1\rangle$. This scheme is "heralded" in the sense that the computation only happens on the two first modes: the last two modes are just witnesses that the computation went well. Unlike the post-selected scheme, measuring these two last modes does not destruct possible entanglement on the two first modes.

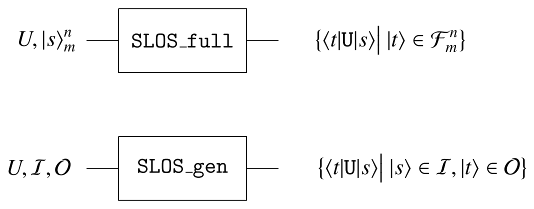
$U, |s\rangle_m^n$
SLOS_full
$\{\langle t|\mathrm{U}|s\rangle \big| \, |t\rangle \in \mathcal{F}_m^n\}$
$U, \mathcal{I}, \mathcal{O}$
SLOS_gen
$\{\langle t|\mathrm{U}|s\rangle \big| \, |s\rangle \in \mathcal{I}, |t\rangle \in \mathcal{O}\}$

Fig. 2. Input and Output relations of `SLOS_full` and `SLOS_gen`, solution algorithms respectively to the Problem 1 and Problem 2 of Section 2.4.

3. `SLOS` algorithms

We first present `SLOS_full` in Section 3.1, which computes the full distribution of a given input, while also proving its complexity in time of $O\left(nM_n\right)$, where $M_n = \#\mathcal{F}_m^n = \binom{n+m-1}{m-1}$ as discussed in Section 2.1. It is the setting of Problem 1 in Section 2.4. We will then present `SLOS_gen` in Section 3.2, when sets of inputs are outputs are allowed. It is the setting of Problem 2 in Section 2.4. We conclude in Section 3.3 with the presentation of `SLOS_hyb`, a simulation algorithm parameterised by a general cost function, and able to capture not only both weak and strong simulation, but also specific, crafted problems.

3.1. `SLOS_full`: computation of the full output distribution of one input

Given an input $|s\rangle_m^n = |s_1, s_2, \ldots, s_m\rangle^n$ and a unitary U, $\mathrm{U}_{\mathcal{F}}\,|s\rangle_m^n$ can be computed with Equation (1). As $\sum_{p=1}^m s_p = n$, the product only contains n non-trivial terms, that can be arbitrary labelled as $p_1, p_2, \ldots, p_n$, corresponding to the position of each photon. The key idea of `SLOS_full`, presented in Algorithm 1, is to decompose that product as follows:

$$\sum_{i_n=1}^{m} u_{i_n,p_n}\,\hat{a}_i^\dagger\left(\sum_{i_{n-1}=1}^{m} u_{i_{n-1},p_{n-1}}\,\hat{a}_i^\dagger\left(\ldots\left(\sum_{i_1=1}^{m} u_{i_1,p_1}\,\hat{a}_i^\dagger\,|0\rangle_m\right)\ldots\right)\right) \tag{4}$$

With this chosen order, we iteratively obtain the output of a state with $k+1$ photons from the output of a state with k photons. To understand more closely how `SLOS_full` works, let's first write the desired output state as:

$$\mathrm{U}_{\mathcal{F}}\,|s\rangle_m^n = \mathrm{U}_{\mathcal{F}}\left(\frac{\hat{a}_{p_n}^\dagger\,\hat{a}_{p_{n-1}}^\dagger \ldots \hat{a}_{p_1}^\dagger\,|0\rangle_m}{\prod_{p=1}^m \sqrt{s_p!}}\right)$$

Algorithm 1: `SLOS_full` with one input $|s\rangle_m^n$ computing the full distribution $\mathrm{U}_{\mathcal{F}}\ |s\rangle_m^n$. *Each coefficient $\langle t|\mathrm{U}_{\mathcal{F}}\ |s\rangle$ will be stored in an array $\mathrm{U}_{\mathcal{F}}$ and will be accessed by $\mathrm{U}_{\mathcal{F}}[|s\rangle][|t\rangle]$. The access of the coefficient $u_{i,p}$ will be labelled as $U[i][p]$. Every coefficient of $\mathrm{U}_{\mathcal{F}}$ is initialised at 0. For simplicity, the array $\mathrm{U}_{\mathcal{F}}$ is of size $O\left((\frac{n}{m}+1)M_n\right)$, even though it could be optimised to $O\left(M_{n-1}+M_n\right)$, by reallocating memory and erasing the intermediary states, as shown below in the comment.*

```
Global $\mathrm{U}_{\mathcal{F}}$
Function SLOS_full($|s\rangle_m^n$, $U$):
    $\mathrm{U}_{\mathcal{F}}[|0\rangle][|0\rangle] \leftarrow 1$ ;
    $\mathcal{I}_S = \left[(|0\rangle_m^0, p_1), \dots, (|s\rangle_m^{n-1}, p_n)\right]$ ;        // chosen arbitrarily, see Equation (4)
    for $k : 0 \to n-1$ do
        $|s\rangle, p \leftarrow \mathcal{I}_S[k]$ ;
        for $|t\rangle \in \mathcal{F}^k$ do
            for $i \in [m]$ do
                $\mathrm{U}_{\mathcal{F}}[|s_1,\dots,s_p+1,\dots,s_m\rangle][|t_1,\dots,t_i+1,\dots,t_m\rangle] \mathrel{+}= \sqrt{\frac{t_i+1}{s_p+1}} \times U[i][l] \times \mathrm{U}_{\mathcal{F}}[|s\rangle][|t\rangle]$ ;
            end
            // Possible memory optimization here: we don't need $\mathrm{U}_{\mathcal{F}}[|s\rangle][|t\rangle]$ anymore
        end
    end
```

and notice that for $k=0$ to $n-1$ we have:

$$\mathrm{U}_{\mathcal{F}}\left(\hat{a}^\dagger_{p_{k+1}}|s\rangle_m^k\right) = \sum_{i=1}^m u_{i,p_{k+1}}\,\hat{a}_i^\dagger\left(\mathrm{U}_{\mathcal{F}}\ |s\rangle_m^k\right) \tag{5}$$

Note that the normalisation factor can either be computed at each step k (as in Algorithm 1), or at the end for the final distribution $\mathrm{U}_{\mathcal{F}}\ |s\rangle_m^n$: in this case, the global normalisation factor is $1/\Pi_{p=1}^m\sqrt{s_p!}$ (as in Equation (1)).

Following Equation (4) and Equation (5), we obtain the full output distribution of $|s\rangle_m^n$, along with the full output distribution of intermediary states having from 0 to $n-1$ photons. We will denote these intermediate states as $|s\rangle_m^k$ and they will be stored in a list $\mathcal{I}_S$ of tuples: $[(|0\rangle_m^0, p_1), \dots, (|s\rangle_m^{n-1}, p_n)]$ where $|s\rangle_m^k$ is the Fock state $|s\rangle_m^{k-1}$ with one more photon in the mode p_k.

In practice, obtaining the full distribution $\mathrm{U}_{\mathcal{F}}\ |s\rangle_m^k = \sum_{|t\rangle\in\mathcal{F}_m^k}\langle t|\mathrm{U}_{\mathcal{F}}\ |s\rangle_m^k\,|t\rangle$ means computing and storing every coefficient $\langle t|\mathrm{U}_{\mathcal{F}}\ |s\rangle_m^n$ for $|t\rangle\in\mathcal{F}_m^k$. We store each of them in an array, where each coefficient of $\mathrm{U}_{\mathcal{F}}\ |s\rangle_m^k$ is used for the computation of $\mathrm{U}_{\mathcal{F}}\ |s\rangle_m^{k+1}$. Note that in the sum of Equation (5), $\hat{a}_i^\dagger\,\langle t|\,\mathrm{U}_{\mathcal{F}}\ |s\rangle_m^k\,|t\rangle = \sqrt{t_i+1}\,\langle t|\,\mathrm{U}_{\mathcal{F}}\ |s\rangle_m^k\,|t_1,\dots,t_i+1,\dots,t_m\rangle$, which adds a new term for $\langle t_1,\dots,t_i+1,\dots,t_m|\,\mathrm{U}_{\mathcal{F}}\ |s\rangle_m^{k+1}$ as in the formula of Algorithm 1.

Time complexity `SLOS_full` computes the full distribution of an input $|s\rangle_m^n$ in $O(nM_n)$. The complexity can be directly deduced from the three "for loops" of Algorithm 1, knowing that $\#\mathcal{F}_m^k = \binom{k+m-1}{m-1} = M_k$. The total number of operations is:

$$\sum_{k=0}^{n-1} mM_k = m\frac{n}{m}M_n = nM_n$$

The complexity is therefore linear in the number of states, so each state needs $O(n)$ operations in average.

Exponential gain from the permanent-based method We would now compare that complexity to the permanent-based method, which would compute every term independently using the permanent algorithm of [18]. Therefore, we need to sum over all the possible outputs states the term $n\frac{\prod_{i=1}^m(t_i+1)}{\min\limits_{t_l\neq 0}(t_l+1)}$. To simplify the expression of the sum, we will use a lower bound, considering $n\geq 1$ so that $\min\limits_{t_l\neq 0}(t_l+1)\leq n+1\leq 2n$. We therefore have the following lower bound:

$$\frac{1}{2}\sum_{|t\rangle\in\mathcal{F}_m^n}\prod_{i=1}^m(t_i+1) \leq \sum_{|t\rangle\in\mathcal{F}_m^n} n\frac{\prod_{i=1}^m(t_i+1)}{\min\limits_{t_l\neq 0}(t_l+1)}$$

As shown in the Lemma 2 of [17], the sum of the left-hand side is equal to $\binom{2m+n-1}{n}$, giving our lower bound of $\Omega\binom{2m+n-1}{n}$. To compare with the time complexity of `SLOS`, we can study the ratio $\frac{\binom{2m+n-1}{n}}{nM_n} = \frac{1}{n}\frac{\binom{2m+n-1}{n}}{\binom{n+m-1}{n}}$. By assuming $m=\theta n$ and for a fixed value θ, (cf Corrollary 2 of [17]), we can apply Stirling's formula to show the ratio is $\Omega(\frac{1}{n}\rho_\theta^n)$ with $\rho_\theta = \frac{(2\theta+1)^{2\theta+1}}{(4\theta)^\theta(\theta+1)^{\theta+1}}$, showing the exponential speedup of `SLOS`. Note that $\rho_1\approx 1.69$, $\rho_2\approx 1.80$, and that $\lim\limits_{\theta\to\infty}\rho_\theta = 2$.

Memory complexity At the step k, we need to use a memory of at most $O\left(M_k+M_{k+1}\right)$, as we use all the coefficients of $\mathrm{U}_{\mathcal{F}}\ |s\rangle_m^k$ to compute and store the new coefficients of $\mathrm{U}_{\mathcal{F}}\ |s\rangle_m^{k+1}$. At the end of the step k, we can erase all the coefficients of the step $k-1$.

Therefore, at the last step, we have stored at most $M_{n-1}+M_n\leq 2M_n = O\left(M_n\right)$ coefficients. Thus, if we allow reallocation, the complexity in memory is in $O\left(M_n\right)$. As we aim for the full distribution containing M_n amplitudes, we necessarily need a memory of $O\left(M_n\right)$. `SLOS_full` has therefore an optimal complexity in memory.

It is important to highlight for a simple allocating, we need $M_{n-1}+M_n$ space which can still be inconvenient for big values of n and m. We can reduce the overhead by only allocating the memory of $\mathrm{U}_{\mathcal{F}}[|s\rangle^{k+1}][|t_1,\dots,t_i+1,\dots,t_m\rangle]$ when needed, and by erasing coefficients as soon they have been used as in Algorithm 1. We can optimise even more by ordering $\mathcal{F}_m^k$ so that for consecutive $|t\rangle$, the coefficients $|t_1,\dots,t_i+1,\dots,t_m\rangle$ overlap and less memory is added at each iteration of a new $|t\rangle$. The memory optimisation can also only be done for the last steps as they are the most costly.

For faster overhead in time and easier implementation, we would rather store all the coefficients of the intermediary states without erasing them. This would require to store $\sum_{k=0}^n M_k = (\frac{n}{m}+1)M_n$ states, which is still feasible for reasonable values of m and n.

3.2. `SLOS_gen`: computation of several outputs for several inputs

The algorithm `SLOS_full` always computes the full output distribution, and does not offer any granularity by restricting the set of outputs states. However, often we may not need the full distribution for a given input. It is the case when we are looking for one coefficient or specific coefficients, as for post-selected or heralded scheme introduced in Section 2.4.

Algorithm 2: `SLOS_gen` with a set of q inputs $\mathcal{I}$ and a set of r outputs $\mathcal{O}$. *Each coefficient* $\langle t|U|s\rangle$ *will be stored in a dictionary or an array* $\mathrm{U}_{\mathcal{F}}$ *and will be accessed by* $\mathrm{U}_{\mathcal{F}}[|s\rangle][|t\rangle]$. *The access of the coefficient* $u_{i,p}$ *will be labelled as* $U[i][p]$. *Every coefficient of* $\mathrm{U}_{\mathcal{F}}$ *is initialised at* 0. *For simplicity, the array* $\mathrm{U}_{\mathcal{F}}$ *is of size* $O\left(q(\frac{n}{m}+1)M_n\right)$, even though it could be optimised by reallocating memory and erasing the intermediary states.

```
Global U_F
Function SLOS_Rec(k, $\mathcal{F}^{k+1}_{\mathcal{S}_\mathcal{M}}$, U):
    if $k > 0$ then
        SLOS_Rec($k-1, \mathcal{F}^{k}_{\mathcal{S}_\mathcal{M}}, U$) ;                // Build with Equation (7)
    end
    for $(|s\rangle, p) \in \mathcal{I}_S[k]$ do
        for $|t\rangle \in \mathcal{F}^{k+1}_{\mathcal{S}_\mathcal{M}}$ do
            for $i \in [m]$ when $t_i \neq 0$ do
                $\mathrm{U}_{\mathcal{F}}[|s_1,\dots,s_p+1,\dots,s_m\rangle][|t\rangle] \mathrel{+}= \sqrt{\frac{t_i}{s_p}} \times U[i][p] \times \mathrm{U}_{\mathcal{F}}[|s\rangle][|t_1,\dots,t_i-1,\dots,t_m\rangle]$
            end
        end
    end
Function SLOS_gen($\mathcal{I}, \mathcal{O}, U$):
    $\mathrm{U}_{\mathcal{F}}[|0\rangle][|0\rangle] \leftarrow 1$ ;
    $\mathcal{I}_S = \left[\{(|0\rangle^0_m, p^i_1)\}, \dots, \left\{\left(|s^i\rangle^{n-1}_m, p^i_n\right)\right\}\right]$ ;   // As in Algorithm 3
    SLOS_Rec($n-1, \mathcal{O}, U$) ;
```

3.2.1. Restriction of the subcomputation space: mask

In order to efficiently restrict the output distribution, we introduce the notion of a *mask*, a state which will filter out unnecessary intermediary states. We define the relation $\leq$ as: $|t\rangle_m \leq |\mathcal{M}\rangle_m \Leftrightarrow \bigwedge_{i=1}^m (t_i \leq \mathcal{M}_i)$, and we define $\mathcal{F}^k_{\leq\mathcal{M}} = \left\{|t\rangle \,\middle|\, |t\rangle \in \mathcal{F}^k_m, |t\rangle \leq |\mathcal{M}\rangle_m\right\}$. For a set of masks $\mathcal{S}_\mathcal{M}$, we will note as $\mathcal{F}^k_{\mathcal{S}_\mathcal{M}}$ all the states we compute at the step k.

At the step k of the computation, we compute each coefficient as follows:

$$\langle t|\mathrm{U}_{\mathcal{F}}|s\rangle^k = \frac{1}{\sqrt{s_{p_k}}} \sum_{i, t_i \neq 0} \sqrt{t_i} u_{i,p_k} \langle t_1,\dots,t_i-1,\dots,t_m|\mathrm{U}_{\mathcal{F}}|s\rangle^{k-1} \tag{6}$$

As $|t\rangle^k \in \mathcal{F}^k_{\leq\mathcal{M}} \Rightarrow \{|t_1,\dots,t_i-1,\dots,t_m\rangle, t_i \neq 0\} \subset \mathcal{F}^{k-1}_{\leq\mathcal{M}}$, for any mask $\mathcal{M}$, the set $\{\langle t|\mathrm{U}_{\mathcal{F}}|s\rangle^k, |t\rangle \in \mathcal{F}^k_{\leq\mathcal{M}}\}$ can be computed with $\{\langle t|\mathrm{U}_{\mathcal{F}}|s\rangle^{k-1}, |t\rangle \in \mathcal{F}^{k-1}_{\leq\mathcal{M}}\}$. For each k, it is sufficient to compute the states $\mathcal{F}^k_{\mathcal{S}_\mathcal{M}} = \left\{\mathcal{F}^k_{\leq\mathcal{M}} \middle| \mathcal{M} \in \mathcal{S}_\mathcal{M}\right\}$. The number of intermediary states is therefore $\#\mathcal{F}^k_{\mathcal{S}_\mathcal{M}}$ instead of M_k for the full distribution in Section 3.1. Given a set of outputs $\mathcal{O} = \{|o\rangle^n_m, |o\rangle^n_m \in \mathcal{F}^n_m\}$ we would like to compute, we can take $\mathcal{F}^k_{\mathcal{S}_\mathcal{M}} = \cup_{o \in \mathcal{O}} \mathcal{F}^k_{\leq o}$, so that at the last step we have $\mathcal{F}^n_{\mathcal{S}_\mathcal{M}} = \mathcal{O}$.

Unlike the iterative presentation of `SLOS_full`, it is here more natural to have a recursive procedure, as we can directly build the intermediary states as follows:

$$\mathcal{F}^{k-1}_{\mathcal{S}_\mathcal{M}} = \left\{|t_1,\dots,t_i-1,\dots,t_m\rangle, t_i \neq 0, |t\rangle^k_m \in \mathcal{F}^k_{\mathcal{S}_\mathcal{M}}\right\} \tag{7}$$

This makes it possible to build $\mathcal{F}^{k-1}_{\mathcal{S}_\mathcal{M}}$ from $\mathcal{F}^k_{\mathcal{S}_\mathcal{M}}$ on the fly without precompiling anything.

3.2.2. The `SLOS_gen` algorithm

Shown in Algorithm 2, `SLOS_gen` starts with $\mathcal{F}^n_{\mathcal{S}_\mathcal{M}} = \mathcal{O}$. The recursive computation of intermediary states is directly derived from Equation (6): `SLOS_rec`($\mathcal{F}^k_{\mathcal{S}_\mathcal{M}}$) computes $\{\langle t|\mathrm{U}_{\mathcal{F}}|s\rangle^k, |t\rangle \in \mathcal{F}^k_{\mathcal{S}_\mathcal{M}}\}$ from $\{\langle t|\mathrm{U}_{\mathcal{F}}|s\rangle^{k-1}, |t\rangle \in \mathcal{F}^{k-1}_{\mathcal{S}_\mathcal{M}}\}$ given by `SLOS_rec`($\mathcal{F}^{k-1}_{\mathcal{S}_\mathcal{M}}$).

However, unlike the case of `SLOS_full`, we are also computing the outputs of *several* input states at the same time. It is in fact possible to take advantage of the similarities in the inputs, by changing the order of computation of Equation (1).

Several inputs Let us consider two inputs $|s\rangle^n_m$ and $|s'\rangle^n_m$, having j photons in common modes. Without loss of generality, we can write them as $|d_1+c_1,\dots,d_m+c_m\rangle$ and $|d'_1+c_1,\dots,d'_m+c_m\rangle$, with $d_i, d'_i, c_i \geq 0$ and $\sum_{i=1}^m c_i = j$. We can therefore first compute $\mathrm{U}_{\mathcal{F}}|c_1,\dots,c_m\rangle_m = \prod_{p=1}^m (\sum_{i=1}^n u_{i,p}\hat{a}_i^\dagger)^{c_p}|0\rangle_m$, common term of both $\mathrm{U}_{\mathcal{F}}|s\rangle^n_m$ and $\mathrm{U}_{\mathcal{F}}|s'\rangle^n_m$. We can notice that $|c\rangle^j_m = |\min(s_1,s'_1),\dots,\min(s_m,s'_m)\rangle$ and so $j = \sum_{i=1}^m \min(s_i,s'_i)$.

More generally, with a set of inputs, we can always factorize the common terms, and use it in the computation to reduce the number of intermediary states. It is especially efficient when the common term is the same for all the inputs, as it is the case when we add ancilla photons for all different inputs, as in $|*,*,\dots,*,1,1,1\rangle$.

When there are many inputs, we want to build $\mathcal{I}_S$ that implies the minimum number of operations. This time, $\mathcal{I}_S$ will be a list of sets, where $\mathcal{I}_S[k]$ is the set of tuples $\left(|s^i\rangle^k_m, p^i_k\right)$ to compute at the step k. Let's note $\mathcal{I}_2$ as the set of all distinct pairs of $\mathcal{I}^2$. If we have $p \geq 2$ inputs, then we would choose one of the biggest common term among all pairs of $\mathcal{I}_2$. Therefore, we would take one element of:

$$\underset{(|s\rangle,|s'\rangle)\in\mathcal{I}_2}{\operatorname{argmax}} \sum_{i=1}^m \min(s_i,s'_i) \tag{8}$$

We then remove the chosen pair $(|s\rangle, |s'\rangle)$ from $\mathcal{I}$, and add $|\min(s_1,t_1),\dots,\min(s_m,t_m)\rangle_m$ both to $\mathcal{I}$ and to $\mathcal{I}_S$, with arbitrary paths from that state to $|s\rangle$ and $|s'\rangle$. The procedure will continue until $p=1$, where we will just add the last input in I_S, with an arbitrary path, as in Equation (4). The pseudocode is the Algorithm 3.

Example To illustrate the correspondence between $\mathcal{I}$ and $\mathcal{I}_S$, we give the steps for the computing the coefficients associated to $\{|1,0,0,1\rangle, |1,1,1,2\rangle, |1,1,2,1\rangle\}$.

1. This is the initial input to the procedure, and $\mathcal{I}_S$ is empty: $\mathcal{I} = \{|1,0,0,1\rangle, |1,1,1,2\rangle, |1,1,2,1\rangle\}$, $\mathcal{I}_S = [\{\},\{\},\{\},\{\},\{\}]$
2. We look for the best common factor between the three inputs (the biggest common term of Equation (8)): it is $|1,1,1,1\rangle$, factor of $|1,1,1,2\rangle$ and $|1,1,2,1\rangle$. So now $\mathcal{I} = \{|1,0,0,1\rangle, |1,1,1,1\rangle\}$, and $\mathcal{I}_S$ is populated with the operations needed to recover the two elements being factored: $\mathcal{I}_S = [\{\},\{\},\{\},\{\},\{(|1,1,1,1\rangle,4),(|1,1,1,1\rangle,3)\}]$

Algorithm 3: Construction of $\mathcal{I}_S$. The function **path**$(|t\rangle \to |s\rangle)$ returns an arbitrary path from $|t\rangle$ to $|s\rangle$ as in Equation (4).

```
$\mathcal{I}_S = [\{\}$ for $i \in [m]]$
while $\#\mathcal{I} \geq 2$ do
    argmax,max $\leftarrow$ None,0
    for $(|s\rangle, |s'\rangle) \in \{(|s\rangle, |s'\rangle) \in \mathcal{I}^2, |s\rangle \neq |s'\rangle\}$ do
        $k = \sum_{i=1}^{m} \min(s_i, s'_i)$
        if $k > max$ then
            argmax,max$\leftarrow (|s\rangle, |s'\rangle)$, k
        end
    end
    if $max=0$ then
        // No factorisation found: we add the paths of all remaining elements in $\mathcal{I}_S$ and empty $\mathcal{I}$
        for $|s\rangle \in \mathcal{I}$ do
            for $(|t\rangle_m^i, l) \in$ path$(|0\rangle_m \to |s\rangle)$ do
                $\mathcal{I}_S$[i].add$((|t\rangle_m^i, l))$
            end
        end
        $\mathcal{I}$.clear()
    end
    else
        // Factorisation for a pair found: we add the paths of the elements of the pair in $\mathcal{I}_S$, remove them from $\mathcal{I}$, and
           add the factor in $\mathcal{I}$
        $(|s\rangle, |s'\rangle) \leftarrow$ argmax
        $|\min_{s,s'}\rangle \leftarrow |\min(s_1, s'_1), \ldots, \min(s_m, s'_m)\rangle$
        for $|s^\star\rangle \in (|s\rangle, |s'\rangle)$ do
            for $(|t\rangle_m^i, p) \in$ path$(|\min_{s,s'}\rangle \to |s^\star\rangle)$ do
                $\mathcal{I}_S$[i].add$((|t\rangle_m^i, p))$
            end
            $\mathcal{I}$.remove$(|s^\star\rangle)$
        end
        $\mathcal{I}$.add$(|\min_{s,s'}\rangle)$
    end
end
if $\#\mathcal{I} == 1$ then
    // One element remaining: we add its path in $\mathcal{I}_S$
    $|s\rangle \leftarrow \mathcal{I}$.pop()
    for $(|t\rangle_m^i, p) \in$ path$(|0\rangle_m \to |s\rangle)$ do
        $\mathcal{I}_S$[i].add$((|t\rangle_m^i, l))$
    end
end
```

Table 1
Time and memory complexity analysis of `SLOS_gen` and [18] for a generic output $|t\rangle = |t_1, \ldots, t_m\rangle \in \mathcal{F}_m^n$, where $M_n = \#\mathcal{F}_m^n = \binom{m+n-1}{n}$. We highlighted the `SLOS` values in bold when they were similar or better than the permanent-based ones. For the memory complexity of the one output case, we consider the worst case, as detailed in 3.2.3. For the memory complexity of the full distribution case, we consider the problem of storing every value, therefore needing at least $O(M_n)$ of memory. If we consider the enumerating problem instead, the permanent-based method would have a memory complexity of $O(n)$, far better than the $O(M_n)$ of `SLOS`. It is an open problem to how much the memory efficiency can be increased.

	Time Complexity		Memory Complexity	
#Outputs	One	All	One	All
Permanent-Based [18]	$O\left(n\frac{\prod_{i=1}^{m}(t_i+1)}{\min_{t_l\neq 0}(t_l+1)}\right)$	$\Omega\binom{2m+n-1}{n}$	$O(n)$	$O(M_n)$
`SLOS_gen`	$\boldsymbol{O\left(\sum_{i=1}^{m} t_i \prod_{j\neq i}(t_j+1)\right)}$	$\boldsymbol{O(nM_n)}$	$O\binom{n}{n/2}$	$\boldsymbol{O(M_n)}$

3. The next step is similar, with the difference that the common factor is a term to be computed (so we keep it). $\mathcal{I} = \{|1,0,0,1\rangle\}$, and $\mathcal{I}_S = [\{\}, \{\}, \{(|1,0,0,1\rangle, 2)\}, \{(|1,1,0,1\rangle, 3)\}, \{(|1,1,1,1\rangle, 4), (|1,1,1,1\rangle, 3)\}]$
4. Finally, we can empty $\mathcal{I}$: $\mathcal{I} = \{\}$, and $\mathcal{I}_S = [\{(|0\rangle_4, 1)\}, \{(|1,0,0,0\rangle, 4)\}, \{(|1,0,0,1\rangle, 2)\}, \{(|1,1,0,1\rangle, 3)\}, \{(|1,1,1,1\rangle, 4), (|1,1,1,1\rangle, 3)\}]$

Complexity of `SLOS_gen` The theoretical complexity of the algorithm `SLOS_gen` is difficult to assess because it heavily depends on the redundancy in input and output states. In Section 3.2.3 we analyse the limit case where only one input and one output are considered, and how it relates to the permanent-based method of [18]. The results are summarised in the Table 1. In Section 4, we present a concrete implementation of `SLOS_gen` in the general case, and we discuss concrete benchmarks.

3.2.3. Limit case with one output/one input

The algorithm `SLOS_gen` can be specialized to the case where one only consider one input and one output state. This is the typical case that can be directly handled by [18]: it consists in computing Equation (2), so one can rely on the complexity results for computing the permanent of a matrix with potential repeated row or columns.

In this section, we discuss the complexity of the procedure in this simple one-input, one-output case, and compare it with [18].

Conjugate trick While it is usually more physical to have one photon at most per mode in the input $|s\rangle_m^n$, if $|s\rangle_m^n$ has more repetition than the output $|o\rangle_m^n$, it is faster and equivalent to compute $\langle s|\mathrm{U}_{\mathcal{F}}^\dagger |o\rangle^\dagger$ than $\langle o|\mathrm{U}_{\mathcal{F}} |s\rangle$. That trick of taking the conjugate transpose allows to transform the repetitions of the columns into the repetition of the rows if necessary, so the number of computation is reduced as much as possible. However, that trick is not general, as it is for a specific instance of one output, and needs more repetitions in the input, that is not very likely in practice with single photon sources. In the following, we will consider a generic input $|s\rangle$ and only consider the repetitions in the output $|t\rangle$.

Worst case To efficiently compute the output $\langle o|\mathrm{U}_{\mathcal{F}} |s\rangle_m^n$, one has to take the mask $|\mathcal{M}\rangle_m = |o\rangle_m^n$. The worst case is when the mask is of the form $|1,1,\dots,1,0,\dots,0\rangle^n$, as it maximises the size of $\mathcal{F}^k_{\leq\mathcal{M}}$ for each k. In that case, $\#\mathcal{F}^k_{\leq\mathcal{M}} = \binom{n}{k}$, and each term needs k operations, as it is the number of terms in the sum of Equation (6). Therefore, the number of needed operations is $\sum_{k=1}^n k\binom{n}{k} = n2^{n-1}$. The complexity therefore matches the time complexity for computing a general permanent, as $\langle 1,\dots,1,0,\dots,0|\mathrm{U}_{\mathcal{F}}|1,\dots,1,0,\dots,0\rangle$ is computed by the permanent of a matrix without any repetition of rows or columns.

General case For each state $|t\rangle_m \in \mathcal{F}_m$, let's define $\alpha_t = \#\{t_i \neq 0, 1 \leq i \leq m\}$. From Eq. (6), we can see that every $\langle t|\mathrm{U}_{\mathcal{F}} |s\rangle^k$ needs α_t computations, using the already computed coefficients. Given an output state $|t\rangle$, we will compute every $|t'\rangle \leq |t\rangle$ where $\leq$ is defined in 3.2.1. Therefore, the number of computations is $S(|t\rangle) = \sum\limits_{|t'\rangle\leq|t\rangle} \alpha_{t'}$. For the proof, we will use the notation $|t\rangle_k = |t_1,\dots,t_k\rangle$, restriction of $|t\rangle_m = |t\rangle$. To give a generic formula of S, we proceed by induction by defining $S_k(|t\rangle) = \sum\limits_{|t'\rangle\leq|t\rangle_k} \alpha_{t'}$, with $S_m(|t\rangle) = S(|t\rangle)$ and $S_1(|t\rangle) = t_1$. We can show that

$$\begin{aligned} S_m(|t\rangle) &= \sum_{|t'\rangle\leq|t\rangle, t'_m=0} \alpha_{t'} + \sum_{k=1}^{t_m} \sum_{|t'\rangle\leq|t\rangle, t'_m=k} \alpha_{t'} \\ &= S_{m-1}(|t\rangle) + \sum_{k=1}^{t_m} \sum_{|t'\rangle\leq|t\rangle_{m-1}} (\alpha_{t'}+1) \\ &= S_{m-1}(|t\rangle) + t_m S_{m-1}(|t\rangle) + t_m \sum_{|t'\rangle\leq|t\rangle_{m-1}} 1 \\ &= (t_m+1)S_{m-1}(|t\rangle) + t_m \prod_{i=1}^{m-1}(t_i+1) \end{aligned}$$

Leading to the following formula:

$$S(|t\rangle) = \sum_{i=1}^m t_i \prod_{j\neq i}(t_j+1)$$

We can show that it is always less than the bound found in [18]:

$$\begin{aligned} \sum_{i=1}^m t_i \prod_{j\neq i}(t_j+1) &= \sum_{t_i\neq 0} t_i \prod_{j\neq i}(t_j+1) \\ &\leq \sum_{t_i\neq 0} t_i \frac{1}{\min\limits_{t_l\neq 0}(t_l+1)} \prod_{j=1}^m (t_j+1) \\ &= \left(\frac{\prod_{j=1}^m (t_j+1)}{\min\limits_{t_l\neq 0}(t_l+1)}\right) \sum_{t_i\neq 0} t_i \\ &= n\left(\frac{\prod_{j=1}^m (t_j+1)}{\min\limits_{t_l\neq 0}(t_l+1)}\right) \end{aligned}$$

That bound is strict whenever there are two $t_i > 0, t_j > 0$ such that $t_i \neq t_j$.

Average case For each state $|t\rangle_m^k \in \mathcal{F}_m^k$, for each $u_{i,*}$ with $i \in [m]$, the term $u_{i,*}\langle t|\mathrm{U}_{\mathcal{F}} |s\rangle^k$ will be used as many times as $|t_1,\dots,t_i+1,\dots,t_m\rangle^{k+1} \leq |\mathcal{M}\rangle_m^n$ with $|\mathcal{M}\rangle_m^n \in \mathcal{F}_m^n$. We can show that the number of time is M_{n-k-1}, which is the number of putting $n-k-1$ photons into m modes, as $|\mathcal{M}\rangle$ is necessarily of the form $|t_1+q_1,\dots,t_i+1+q_i,\dots,t_m+q_m\rangle$ with $q_i \geq 0$ and $\sum_i q_i = n-k-1$.

As there are M_n different output states, we can show the average number of operation for computing one output is:

$$\begin{aligned} \frac{1}{M_n}\sum_{k=0}^{n-1} mM_kM_{n-k-1} &= \frac{m}{M_n}\binom{2m+n-2}{n-1} \\ &= \frac{n}{M_n}\binom{2m+n-1}{n}\frac{m}{2m+n-1} \end{aligned}$$

It is therefore the same average complexity of [17] (cf. the Corollary 3 for the precise formula), with a multiplicative ratio of $\frac{m}{2m+n-1} < 1$.

Memory complexity Even if `SLOS_gen` is better in time for one input and one output than [18], it is important to highlight it is far more costly in memory. Given an output state $|t\rangle^n$, at the step k, we need to store all the coefficients $\langle t|\mathrm{U}_{\mathcal{F}} |s\rangle^{k-1}$ and $\langle t|\mathrm{U}_{\mathcal{F}} |s\rangle^k$, with $|t\rangle^{k-1} \leq |t\rangle^k \leq |t\rangle^n$. As $\#\left\{|t\rangle^k \leq |t\rangle^n\right\}$ doesn't have a simple formula [39], we will only consider the worst case. The worst case is when $|t\rangle$ has at most one photon per mode, for instance when $|t\rangle = |1,1,\dots,1,0,\dots,0\rangle^n$. In that case, the number of space needed at each step is $O\left(\binom{n}{k}\right)$. As we only need to store the values of two steps, and not all the n steps, we can only consider the costliest which is when $k = n/2$. The worst case space complexity is therefore $O\left(\binom{n}{n/2}\right)$.

3.3. `SLOS_hyb`: general procedure for both weak and strong simulations

This section presents `SLOS_hyb` (shown in Algorithm 4), a hybrid of weak and strong simulation. It can be seen as an extension of `SLOS_gen`, with an iterative procedure and a recursive subroutine.

Let us denote $\mathcal{F}_S^k$ the set of intermediary states computed at the step k with Equation (6). In Section 3.1 we were interested in computing every state, so $\mathcal{F}_S^k = \mathcal{F}_m^k$, while in Section 3.2 we had masks to compute specific outputs, so $\mathcal{F}_S^k = \mathcal{F}^k_{S_{\mathcal{M}}}$. For the hybrid version in Algorithm 4, we use a `Select` function to select the next intermediary states we would like to compute, so $\mathcal{F}_S^{k+1}$ is defined as `Select`$(\mathcal{F}_S^k)$. In order to compute all coefficients in $\mathcal{F}_S^{k+1}$, we use `SLOS_Rec` with the particularity to only compute intermediary states that have not been computed so far, and we directly use the intermediary states needed for the coefficients of $\mathcal{F}_S^k$. We can code that information in a Boolean array (or a hash table) $\mathcal{B}$ such that $\mathcal{B}[|s\rangle][|t\rangle]$ is true if $\mathrm{U}_{\mathcal{F}}[|s\rangle][|t\rangle]$ has been computed, false otherwise.

For strong simulations, we are aiming at computing the probabilities of a given set of outputs. However, sometimes we would like to compute the states with the biggest probabilities, without knowing in advance the probability distribution of the set of outputs. More generally, we would like to compute outputs giving conditions on the distribution even if we don't know it yet. For instance, `SLOS_hyb` can solve the following problem.

Problem 3: hybrid simulation Given an input $|s\rangle_m^n$, a threshold η and a LO-circuit of m modes mapping a unitary $U: m\times m$, can we obtain a set $O = \left\{|t\rangle, |t\rangle \in \mathcal{F}_m^n\right\}$ such that $\sum_{|t\rangle\in O}\left|\langle t|\mathrm{U}_{\mathcal{F}} |s\rangle\right|^2 > \eta$?

Algorithm 4: `SLOS_hyb` with one input $|s\rangle_m^n$ and `Select` function. *Each coefficient $\langle t|U\,|s\rangle$ will be stored in a dictionary or an array $\mathrm{U}_{\mathcal{F}}$ and will be accessed by $\mathrm{U}_{\mathcal{F}}[|s\rangle][|t\rangle]$. The access of the coefficient $u_{i,p}$ will be labelled as $U[i][p]$. Every coefficient of $\mathrm{U}_{\mathcal{F}}$ is initialised at* 0. *For simplicity, the array $\mathrm{U}_{\mathcal{F}}$ is of size $O\left((\frac{n}{m}+1)M_n\right)$. The `Select` function enables to do Strong simulation without or with masks, weak simulation, or a mix of them. $\mathcal{B}$ is an array or dictionary initialised at False and stores the coefficients already computed, so we don't need to recompute $\mathrm{U}_{\mathcal{F}}[|s\rangle][|t\rangle]$ if $\mathcal{B}[|s\rangle][|t\rangle]$ is true. $\mathcal{F}^k_{\leq\mathcal{M}}\cap\overline{\mathcal{B}}_s$ is a shortcut for $\{|t\rangle\,|(|t\rangle\in\mathcal{F}^k_{\leq\mathcal{M}})\wedge(\mathcal{B}[|s\rangle][|t\rangle]==False)\}$.*

```
Global $\mathrm{U}_{\mathcal{F}}$, $\mathcal{B}$
Function SLOS_Rec($k, \mathcal{I}_S, \mathcal{F}^k_S, U$):
    $(|s\rangle, p) \leftarrow \mathcal{I}_S[k]$
    if $k > 1$ then
        SLOS_Rec($k-1, \mathcal{I}_S, \mathcal{F}^{k-1}_S \cap \overline{\mathcal{B}}_s, U$) ;          // Build with Equation (7)
    end
    for $|t\rangle \in \mathcal{F}^k_S$ do
        for $i \in [m]$ when $t_i \neq 0$ do
            $\mathrm{U}_{\mathcal{F}}[|s_1,\dots,s_p+1,\dots,s_m\rangle][|t\rangle] \mathrel{+}= \sqrt{\frac{t_i}{s_p}} \times U[i][p] \times \mathrm{U}_{\mathcal{F}}[|s\rangle][|t_1,\dots,t_i-1,\dots,t_m\rangle]$
        end
        $\mathcal{B}[|s\rangle][|t\rangle] \leftarrow$ True
    end
Function SLOS_hyb($|s\rangle^n_m$, $U$, Select):
    $\mathrm{U}_{\mathcal{F}}[|0\rangle][|0\rangle] \leftarrow 1$ ;
    $\mathcal{I}_S = \left[(|0\rangle^0_m, p_1), \dots, (|s\rangle^{n-1}_m, p_n)\right]$ ;          // chosen randomly from Equation (1)
    $\mathcal{F}^k_S = \{|0\rangle\}$
    for $k : 1 \to n$ do
        $\mathcal{F}^k_S \leftarrow$ Select($\mathcal{I}_S, \mathcal{F}^k_S, k$)
        SLOS_Rec($k, \mathcal{I}_S, \mathcal{F}^k_S, U$)
    end
    return $\mathcal{F}^k_S$
```

Even though this problem can be solved with 2.4 and just taking a subset of the full distribution, we can have a more efficient procedure by using the Algorithm 4 and the `Select` function of Algorithm 5. Note the "while loop" adds several states in $\mathcal{F}$.

Weak simulation The algorithm of [16] uses the probability chain rule to sequentially sample photon by photon. At each step, some probabilities are computed in order to sample the outputs $|t\rangle^k$ for $1\leq k\leq n$. At the end, we sample the desired sample $|t\rangle^n$ with n photons from the desired distribution. To lighten the notation, we will note $t^k=|t\rangle^k, s^k=|s\rangle^k$, and we will note $p(t^k|t^{k-1},s^k)$ the conditional probability to sample $|t\rangle^k$ conditioned on $|t\rangle^{k-1}\leq|t\rangle^k$, meaning we know the position of $k-1$ photons in the output, and conditioned on the input[2] being $|s\rangle^k$.
The main idea is to avoid directly sampling from $p(t^k)$, by noticing we can equivalently sample from $p(t^k|s^k)$, where the inputs $|s\rangle^k$ are randomly chosen. Then, to sample from $p(t^k|s^k)$, we use the chain rule to sample from $p(t^k|t^{k-1},s^k)$ instead, given we already sampled $|t\rangle^{k-1}$. The technical points can be summarised as follows[3]:

- We can show that $p(t^k)\propto\sum_{|s\rangle^k\leq|s\rangle^n}p(t^k|s^k)\propto\mathbb{E}_{|s\rangle^k\leq|s\rangle^n}\left\{p(t^k|s^k)\right\}$, with $p(t^k|s^k)=\left|\langle t|\,\mathrm{U}_{\mathcal{F}}\,|s\rangle^k\right|^2$, and with the expectation value summing uniformly over $|s\rangle^k\leq|s\rangle^n$. (cf Lemma 1 and 3 of [16][4]).
- We sample from $\mathbb{E}_{|s\rangle^k\leq|s\rangle^n}\left\{p(t^k|s^k)\right\}$ instead than sampling from $p(t^k)$. As the $|s\rangle^k$ are always uniformly distributed, we can first uniformly sample the order $\left\{|s\rangle^k, 1\leq k\leq n\right\}$, and then sample from the $p(t^k|s^k)$ distribution.
- To sample from $p(t^k|s^k)$, we use the chain rule. For a fixed order $\left\{|s\rangle^k, 1\leq k\leq n\right\}$ - randomly sampled at the step 2 - we have $p(t^n|s^n)=p(t^1|s^1)p(t^2|t^1,s^2)\dots p(t^n|t^{n-1},s^n)$. At the step k, given the previous sample $|t\rangle^{k-1}$, we sample the output $|t\rangle^k$ from $p(t^k|t^{k-1},s^k)$.
- $\left\{p(t^k|\,|t_1,\dots,t_m\rangle^{k-1},s^k), |t\rangle^k\geq|t\rangle^{k-1}\right\}=\left\{\left|\langle t_1,\dots,t_i+1,\dots,t_m|\mathrm{U}_{\mathcal{F}}\,|s\rangle^k\right|^2, 1\leq i\leq m\right\}$. We therefore compute those m permanents to compute the desired probabilities and sample $|t\rangle^k$.

We will now explicit how `SLOS_hyb` in Algorithm 4 can perform the weak simulation. To be clearer and slightly more efficient, the commands in the for loop are inversed: we first compute the amplitudes with `SLOS_Rec`, and then we would use the `Select` function to sample:

1. Given an input $|s\rangle^n$, the random order $s^1\leq s^2\leq\dots\leq s^n$ is done by the random choice of $\mathcal{I}_S$. We initialise $\mathcal{F}^k_S=\{|1,0,\dots,0\rangle_m,\dots,|0,\dots,0,1\rangle_m\}$.
2. The `Select` function at the step $1\leq k\leq n$ and given a set of states $\mathcal{F}^k_S$, will sample $|\alpha\rangle^k$ from $\left\{\left|\langle t|\,\mathrm{U}_{\mathcal{F}}\,|s\rangle\right|^2\middle|\,|t\rangle\in\mathcal{F}^k_S\right\}$. Then if $k<n$, it will return the next values to compute, i.e. the set $\{|\alpha_1,\dots,\alpha_i+1,\dots,\alpha_m\rangle, i\in[m]\}$. If $k=n$, then we are at the last step, so we just can return the last sample $\{|\alpha\rangle^n\}$.

4. Implementation

In this section, we discuss the concrete implementation of `SLOS_gen`, as found in Perceval [19].

To build-up intuition, Table 2 gives some orders of magnitude to the quantities involved and Table 3 gives a quick equivalence of these quantities in time and storage. Considering the exponential number of values we are dealing with, our challenge is to optimize as much as possible the memory structures and the details

[2] In the case of Boson Sampling we generally have $|s\rangle^n=|1,1,1,\dots,1,0,\dots,0\rangle_m$.

[3] We use $\propto$ to lighten the formula and disposing of factorials coefficients depending on k,n.

[4] In that paper, the photon to mode encoding as described in Section 5.3 is used, so the positions of the i^{th} photon is described by $r_i\in[m]$. Linking the notations, we have $p(r^k|r_1,\dots,r_{k-1})=p(t^k|t^{k-1})$. Their $\boldsymbol{\alpha}$ is a way to parameterise the permutation of the columns, representing the order of the input photons. Therefore considering a random permutation $\alpha_1,\dots,\alpha_n$ is equivalent to consider a random order $\left\{|s\rangle^k, 1\leq k\leq n\right\}$ of the inputs.

Algorithm 5: `Select` function which answers to the Problem 3.

```
Function Select($\mathcal{I}_S, \mathcal{F}_S^k, k$):
    S ← 0
    $(|s\rangle, \star) \leftarrow \mathcal{I}_S[k]$
    $\mathcal{F} \leftarrow \{\}$
    while $S \leq \eta$ do
        $|t\rangle = \mathcal{F}_S^k.pop()$
        $S \mathrel{+}= \left|\langle t|\,\mathrm{U}_{\mathcal{F}}\,|s\rangle\right|^2$
        $\mathcal{F}.add(|t\rangle)$
    end
    return $\left\{|t_1,\ldots,t_i+1,\ldots,t_m\rangle, |t\rangle \in \mathcal{F}, i \in [m]\right\}$
```

Table 2
Value of M_n for practical combinations of m and n.

n	M_n for $m = n$	M_n for $m = 2n$	M_n for $m = 3n$
12	1.35×10^6	8.34×10^8	5.23×10^{10}
14	$2,01 \times 10^7$	3.52×10^{10}	4.35×10^{12}
16	3.01×10^8	1.5×10^{12}	3.66×10^{14}
17	1.17×10^9	9.85×10^{12}	3.37×10^{15}
18	4.54×10^9	6.46×10^{13}	3.11×10^{16}
20	6.89×10^{10}	2.79×10^{15}	2.65×10^{18}

Table 3
Equivalency of 10^k with concrete time and storage references.

	time necessary for single instructions processed on a 1 GHz computer	equivalent storage size (any unit)	number of bytes necessary to store a pointer
10^6	1 milliseconds	0.95 Mega	3
10^8	0.1 seconds	95 Mega	4
10^{10}	10 seconds	9.3 Giga	5
10^{12}	17 minutes	903 Giga	5
10^{14}	27 hours	90 Tera	6
10^{16}	115d	8.8 Peta	7
10^{18}	31 years	888 Peta	8

of the implementation to gain practical factors allowing to extend the strong simulation for a few more photons.

In this section we assume that we have a fixed number of modes m. We will first describe the memory structure we have been introducing for the implementation of the algorithms. We then discuss the implementation challenges and the optimization tricks, and finally add a note on masking.

4.1. Memory structures

The two main memory structures will be labelled as `fsarray` (Fock State Array) and `fsimap` (Fock State Inverse Map). `fsarray` is mainly used to get indices used for building `fsimap`. Both structures depend only on m and n and are independent of the unitary matrix. During the actual simulation they will be used as read-only structures containing the "*calculation path*". It is thus possible to pre-compute these structures and serialize them to files. For the largest ones, it is also possible to use the files as a memory-map.

`fsarray` - indexes of the Fock states Each `fsarray`(k) is an array in charge of representing all the Fock states $|s\rangle_m^k$ and assigning a unique ID to each of them. We want this structure to be searchable, and to have the least memory footprint. For this we use sequences $S(|s\rangle^k) = (p_1, ..., p_k)$ where p_i is the position of the i^{th} photon in the Fock state $|s\rangle^k$. For instance: $S(|1, 0, 0, 2\rangle) = (1, 4, 4)$, which we denote ADD, mapping each number to an uppercase letter for the sake of readability.

This representation of Fock states, which we call *photon to mode* encoding, has the following nice properties:

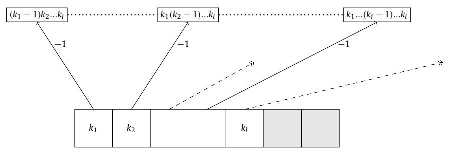

Fig. 3. Representation of a cell of a `fsimap`(n): here the n photons are occupying l mode, so we have $n - l$ pointers not used.

1. It is ordered using an intuitive lexicographical order on the $(p_1, ..., p_k)$: e.g.

$$S(|1,0,0,2\rangle) = \mathtt{ADD} > S(|1,1,1,0\rangle) = \mathtt{ABC}.$$

   Thus it is searchable.
2. The sequence $S(|s\rangle)$ is iterable: $(p_1, ..., p_k) \xrightarrow{+1} (p'_1, ..., p'_k)$, where we can simply increment as follows: (a) $p'_k = p_k + 1$ if $p_k < m$, otherwise (b) we calculate p_{k-1} the same way (possibly going to $p_{k-2}, \ldots, p_1$) and $p_k = p_{k-1}$, e.g. $(1, 4, 4)/\mathtt{ADD} \xrightarrow{+1} (2, 2, 2)/\mathtt{BBB}$.
3. The sequence $S(|s\rangle)$ can be represented in memory with a fixed $\lceil \log_2(m) \rceil$-bit buffer.
4. It is more compact than *mode to photon* encoding for $n < m$. This case is the most common use case with circuits with single photon sources, as each input mode would have at most one photon.

Properties 1 and 3 imply that, by binary search, it is possible to find a specific Fock state in a `fsarray` in $O(\log_2 M_k)$. Property 2 implies that it is even possible to not store the `fsarray` if we are interested in **all** the Fock states since we can directly iterate through them. Thus, we will only store `fsarray`(n), as the computation of the states `fsarray`$(k+1)$ will only need the iteration of `fsarray`(k) and the structure `fsimap` as detailed below.

`fsimap` - mapping between Fock states Our algorithms require that we store a map between the "child" Fock states of $k+1$ photons $|s\rangle_m^{k+1}$ to the "parent" Fock states of k photons $|s\rangle_m^k$ (Fig. 3). We call this structure `fsimap`$(k+1)$ and we need to build a list of n `fsimap`, `fsimap`(n), `fsimap`$(n-1)$,..., `fsimap`(1).

For a Fock state to find its parent, we just need to remove one photon. For instance for ADD we just remove a A or a D to reach AD or DD. So more generally for each M_{k+1} Fock state in $\mathcal{F}_m^{k+1}$ we need to store up to $k+1$ indexes to the parent. Most of the time we could store less, but we prefer keeping direct access to each entry in the `fsmap`, so we are keeping for each state exactly $k+1$ pointers.

The size of a layer is then: $S_{\mathtt{fsimap}(k)} = \mathtt{fsimap}(k)$ is $\lceil \log_{256}(M_{k-1}) \rceil \times k$ bytes.

So to summarize - starting from all the states in $\mathcal{F}_m^{k+1}$, we can find their index *idx* thanks to `fsa`$(k+1)$. With the index, we can find the location of the index to the parent states in `fsa`(k) by direct memory access in `fsimap`$(k+1)$ to the position: $idx \times (k+1) \times S_{\mathtt{fsimap}(k)}$.[5]

The construction of `fsimap` is straightforward using the 2 involved `fsarray`.

[5] Note that we can get a more compact version of the `fsimap`(k) for $k < n$ when the size of the index to each state is 4-bytes long. Indeed in that case, we can then get rid of the non-used cells for the parents pointers. We then lose the ability the direct access through the `fsa`(k) index, but this index is only used when referred to by child structure, so we can instead replace the index by the direct pointer to the position of the state in memory.

4.2. Implementation optimization

All the implementation is done in C++ optimized with SIMD vectorization. Some specific points:

1. Vectorization: Cross-platform SIMD vectorization is done through MIPP [40] library[6] supporting SSE, AVX, AVX-512 and ARM NEON (32-bit and 64-bit) instructions. The main operation that we are interested in is "horizontal vector sum of complex number multiplication". This operation is not a primitive and is decomposed into the 3 MIPP primitives: `interleave`, `cmul`, `hadd`.
2. The `SLOS_full` algorithm is fully iterative: for $k = 1$ to n the coefficients corresponding to $|s\rangle_m^{k+1}$ are computed in one single loop by iterating on the `fsimap`(n) structure. Each new coefficient is a sum of products of unitary matrix coefficient times coefficient of $|s\rangle_m^k$. There is no overhead in the implementation.
3. Recursion: The `SLOS_gen` algorithm is recursive. Starting from the index retrieved by binary search on `fsaarray` on layer n, the expected coefficient (the probability amplitude) is calculated by recursively retrieving the coefficients for all the parents at level $k-1$. To reduce overhead caused by recursion, a bit vector is used to check if the parent coefficient is not yet calculated before going recursive. Also, all the structures necessary for Vectorization are pre-allocated globally to reduce overhead created by dynamic allocation on the stack.
4. Memory access: one major challenge for `SLOS` algorithms resides in memory allocation. When pushed to the maximum (say, 17 photons for 34 modes) - memory structures will use several hundreds/thousands of gigabytes in memory. Each coefficient calculation needs parent coefficient scattered on this memory range limiting possibility of processor to benefit from hardware L_x memory caching. Compared to local permanent calculation algorithm, this memory access time creates overhead depending strongly on hardware configuration as seen in Section 5.
5. Multithreading:
   - Algorithm `SLOS_full` can be fully run on multiple threads without additional overhead: for each k, we simply divide the coefficients list by the number of threads.
   - For algorithm `SLOS_gen`, multi-threading is more limited: it is only possible to distribute top branches of the calculation.

   In both cases, impact of the multi-threading is actually limited by memory access: adding more threads on very large memory structure do not significantly increase the performance.

4.3. Masking

Let us note that the implementation proposed is fully compliant with the notion of "masking" introduced in section 3.2.1: technically masking is a transversal optimization allowing to reduce (potentially massively) the complete Fock state space and therefore reducing proportionally the time and storage.

The only location in the code impacted by `masking` is on the `fsaarray` construction. To find all the masked items, we are keeping the same global iterations on all possible Fock states, but are skipping the ones that are not compliant with the mask. It is likely that we could find a faster iteration method on masked Fock states, however, since this only impacts pre-computing, we don't need a special optimization.

[6] https://github.com/aff3ct/MIPP.

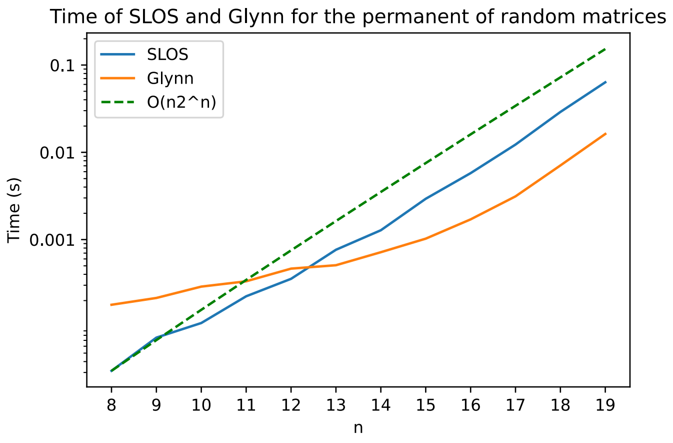

Fig. 4. Comparative performance of Glynn and `SLOS` for calculation of a single $n \times n$ permanent 128-bit complex numbers. The green curve is the $n2^n$ curve renormalised with the first point of the `SLOS` curve. Benchmark computes 100 permanents in a row and outputs the mean time. The benchmark is run on a Intel Core i7-10510U with 16GB of memory. (For interpretation of the colours in the figure(s), the reader is referred to the web version of this article.)

Once the pre-computed memory structures detailed below are built, there is no difference in implementation when simulating on a masked or non-masked system.

5. Performance

In this section, we discuss practical aspects of SLOS. We first show a typical use case demonstrating the need of strong simulations and effectiveness of our implementation in Section 5.1. We then discuss possible limitations. In Section 5.2, we compare performance of `SLOS` for a single output with at most one photon per mode compared to Glynn's algorithm, state of the art algorithm for computing the permanents of generic matrices. Finally, in Section 5.3 we analyze the space limitation of strong simulation using our algorithm.

5.1. A typical QML application requiring strong simulation

Using the framework Perceval [19], we have implemented[7] the simulation of [22], using a generic $m \times m$ interferometer and its full output probability distribution to train a model solving differential equations. For an evolving configuration of the circuit, the algorithm has to iterate on all output states $|t\rangle_m^n$ corresponding to input state $|1, 1, ..., 1\rangle_n$. Based on Broyden–Fletcher–Goldfarb–Shanno (BFGS) optimiser [41], the algorithm converges in 200-400 iterations, each of them needing thousands of full distribution computations. Table 4 shows the evolution of M_n for different values of n and the time necessary for 150 iterations of the algorithm comparatively with the direct calculation of M_n permanent with Glynn's algorithm and `SLOS_full` algorithm. Use of `SLOS_full` practically allows pushing of simulation from 6 photons (processed in 2 min with `SLOS_full` and 11 h without) to 10 photons.

5.2. Benchmarking `SLOS_gen` for one output

We compare in Fig. 4 the speed of our algorithm to compute a single permanent in the worst possible situation, i.e. when the output is $|1, ..., 1\rangle$, with a traditional permanent calculation algorithm. We selected the algorithm from Glynn [15] as [19] shows

[7] Details of the algorithm can be found at https://perceval.quandela.net/docs/notebooks/Differential%20equation%20solving.html.

Table 4
In this table, we compare time of QML algorithm to perform 150 optimization iterations with both Permanent-Based and SLOS algorithms. This application is the ideal use case of SLOS since we are interested into the full output states exact distribution.

	Number of photons n									
	2	3	4	5	6	7	8	9	10	11
M_n	3	10	35	126	462	1716	6435	24310	92378	352716
Permanent-Based	36s	3m	16m	1h30	11h	3d	*not possible*			
SLOS	14s	15s	29s	73s	2m	5m	22m	1h45m	10h	2d8h

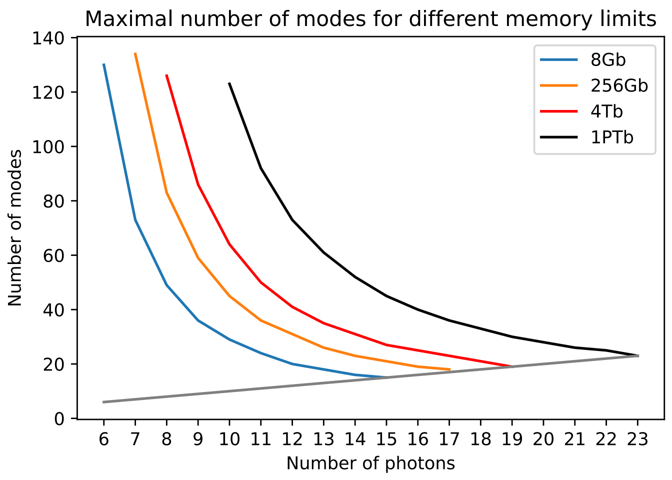

Fig. 5. This graph shows four regions for (m, n) combinations. In blue - $m \geq n$ with configurations that can run on a personal laptop (up to 8Gb memory). Curves in orange/red/black are the respective limits of 256Gb/4Tb/1Pb memory. For instance, strong simulation with SLOS of 24 photons on a 24 modes circuit would require $1500Tb$ of processing memory!

that for up to 19 modes, this algorithm is practically one of the most efficient.

The curves do not show the precompilation time needed for SLOS nor the allocation of the Fock states. We can see that the practical time for SLOS is better for small cases, and that, as predicted by the complexity analysis, the growth is very close to the $n2^n$ curve. This is even true for small instances.

The code for Glynn's algorithm developed in *Perceval* [19] makes heavy use of hardware optimizations, and shows a speedup compared to SLOS when $n > 12$. The speedup has a multiplicative factor between 2 and 4 depending on the machine. On Fig. 4, the factor is around 4 for large values of n. Indeed, as the data in Glynn's algorithm is local, one can rely on efficient libraries to sumprod the terms, improving the time of the computations. Due to the structure of the algorithm (because of the heavy use of precomputed data), SLOS cannot make use of these optimizations. One has however to note that without these hardware-specific optimizations, there is no particular speedup for Glynn's algorithm compared to SLOS.

5.3. Memory usage

Finally, since SLOS_gen is making intensive use of memory, we have computed in Fig. 5 the practical limitations of strong simulation. The graphics vertical axis is the number of modes m, and the horizontal axis the number of photons n. We are only interested in $m \geq n$, and we have drawn boundaries of three typical workstations. In blue, computer with up to 8Gb of memory - typically any modern laptop. In orange, the limit of 256 Gb memory - typically a large compute node. Any (m, n) point below the orange curve will fit in 256 Gb memory. The red curve represents memory need up-to 4 Terabytes memory representing a very large HPC node. The black curve represents a potential super-computer with up to 1 Petabyte of memory. As of today technology this boundary can be considered as an area unreachable for strong simulation, and so for a full description of linear optical processes.

6. Conclusion

In this paper, we presented a versatile framework for the simulation of linear optical circuits, with a trade-off between time and memory usage. An efficient implementation is provided and we discuss how it outperforms the permanent-based algorithms. It is an open problem to determine to what extent the memory usage of SLOS can be improved, or to what extent the time complexity of the permanent-based method can be improved with more memory. As a future work, we plan to incorporate noise models, and validate the simulations against physical hardware.

Declaration of competing interest

The authors declare no competing interest.

Data availability

Data will be made available on request.

Acknowledgement

The authors would like to thank Rawad Mezher and Timothée Goubault de Brugière for helpful discussions and references, and the two reviewers for their very valuable comments, that have greatly improved the quality of the paper. This work is supported by the PEPR integrated project EPiQ ANR-22-PETQ-0007 part of Plan France 2030 and by the CIFRE 2022/0081. It is also supported by the French National Research Agency (ANR) under the research projects SoftQPro ANR-17-CE25-0009-02, by the STIC-AmSud project Qapla' 21-STIC-10, and by the BPI France Concours Innovation PIA3 projects DOS0148634-00, DOS0148633-00.

References

[1] Peter W. Shor, in: Proceedings 35th Annual Symposium on Foundations of Computer Science, IEEE, 1994, pp. 124–134.
[2] Charles H. Bennet, in: Proc. of IEEE Int. Conf. on Comp. Sys. and Signal Proc., Dec. 1984, 1984.
[3] Gilles Brassard, Found. Phys. 33 (11) (2003) 1593–1616.
[4] Alexia Auffèves, PRX Quantum 3 (2) (2022) 020101.
[5] Daniel Jaschke, Simone Montangero, Quantum Sci. Technol. 8 (2) (2023) 025001, https://doi.org/10.1088/2058-9565/acae3e.
[6] Rudolph Terry, APL Photon. (ISSN 2378-0967) 2 (3) (2017), https://doi.org/10.1063/1.4976737.030901.
[7] Scott Aaronson, Alex Arkhipov, in: Proceedings of the Forty-Third Annual ACM Symposium on Theory of Computing, STOC '11, Association for Computing Machinery, ISBN 978-1-4503-0691-1, 2011, pp. 333–342.
[8] Han-Sen Zhong, Hui Wang, Yu-Hao Deng, Ming-Cheng Chen, Li-Chao Peng, Yi-Han Luo, Jian Qin, Dian Wu, Xing Ding, Yi Hu, Peng Hu, Xiao-Yan Yang, Wei-Jun Zhang, Hao Li, Yuxuan Li, Xiao Jiang, Lin Gan, Guangwen Yang, Lixing You, Zhen Wang, Li Li, Nai-Le Liu, Chao-Yang Lu, Jian-Wei Pan, Science 370 (6523) (2020) 1460–1463, https://doi.org/10.1126/science.abe8770.
[9] Yulin Wu, Wan-Su Bao, Sirui Cao, Fusheng Chen, Ming-Cheng Chen, Xiawei Chen, Tung-Hsun Chung, Hui Deng, Yajie Du, Daojin Fan, Ming Gong, Cheng

Guo, Chu Guo, Shaojun Guo, Lianchen Han, Linyin Hong, He-Liang Huang, Yong-Heng Huo, Liping Li, Na Li, Shaowei Li, Yuan Li, Futian Liang, Chun Lin, Jin Lin, Haoran Qian, Dan Qiao, Hao Rong, Hong Su, Lihua Sun, Liangyuan Wang, Shiyu Wang, Dachao Wu, Yu Xu, Kai Yan, Weifeng Yang, Yang Yang, Yangsen Ye, Jianghan Yin, Chong Ying, Jiale Yu, Chen Zha, Cha Zhang, Haibin Zhang, Kaili Zhang, Yiming Zhang, Han Zhao, Youwei Zhao, Liang Zhou, Qingling Zhu, Chao-Yang Lu, Cheng-Zhi Peng, Xiaobo Zhu, Jian-Wei Pan, Phys. Rev. Lett. 127 (18) (2021) 180501, https://doi.org/10.1103/PhysRevLett.127.180501.

[10] E. Knill, R. Laflamme, G.J. Milburn, Nature (ISSN 1476-4687) 409 (6816) (2001) 46–52, https://doi.org/10.1038/35051009.

[11] Sara Bartolucci, Patrick Birchall, Hector Bombin, Hugo Cable, Chris Dawson, Mercedes Gimeno-Segovia, Eric Johnston, Konrad Kieling, Naomi Nickerson, Mihir Pant, et al., Nat. Commun. 14 (1) (2023) 912.

[12] G. Valiant Leslie, Theor. Comput. Sci. (ISSN 0304-3975) 8 (2) (1979) 189–201, https://doi.org/10.1016/0304-3975(79)90044-6.

[13] Stefan Scheel, Permanents in linear optical networks, Available as arXiv:quant-ph/0406127, 2004.

[14] Herbert John Ryser, Combinatorial Mathematics, The Carus Mathematical Monographs, vol. 14, American Mathematical Society, ISBN 9781614440147, 1963.

[15] David G. Glynn, Eur. J. Comb. (ISSN 0195-6698) 31 (7) (2010) 1887–1891, https://doi.org/10.1016/j.ejc.2010.01.010.

[16] Peter Clifford, Raphaël Clifford, in: Proceedings of the 2018 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), Proceedings, Society for Industrial and Applied Mathematics, 2018, pp. 146–155.

[17] Peter Clifford, Raphaël Clifford, Faster classical boson sampling, Available as arXiv:2005.04214, 2020.

[18] Valery Shchesnovich, On the classical complexity of sampling from quantum interference of indistinguishable bosons, Available as arXiv:1904.02013, 2019.

[19] Nicolas Heurtel, Andreas Fyrillas, Grégoire de Gliniasty, Raphaël Le Bihan, Sébastien Malherbe, Marceau Pailhas, Boris Bourdoncle, Pierre-Emmanuel Emeriau, Rawad Mezher, Luka Music, Nadia Belabas, Benoît Valiron, Pascale Senellart, Shane Mansfield, Jean Senellart, Quantum 7 (2023) 931, https://doi.org/10.22331/q-2023-02-21-931.

[20] Brajesh Gupt, Josh Izaac, Nicolás Quesada, J. Open Sour. Softw. 4 (44) (2019) 1705.

[21] Nicolas Heurtel, Shane Mansfield, Jean Senellart, Benoît Valiron, in: Proceedings of the 2022 IEEE International Conference on Quantum Computing and Engineering (QCE), 2022, pp. 577–581.

[22] Beng Yee Gan, Daniel Leykam, Dimitris G. Angelakis, EPJ Quantum Technol. 9 (1) (2022), https://doi.org/10.1140/epjqt/s40507-022-00135-0.

[23] Pieter Kok, W.J. Munro, Kae Nemoto, T.C. Ralph, Jonathan P. Dowling, G.J. Milburn, Rev. Mod. Phys. 79 (2007) 135–174, https://doi.org/10.1103/RevModPhys.79.135.

[24] Pieter Kok, Brendon W. Lovett, Introduction to Optical Quantum Information Processing, Cambridge University Press, ISBN 9781139193658, 2010.

[25] William Feller, An Introduction to Probability Theory and Its Applications, vol. 1, Wiley, ISBN 0471257087, 1968.

[26] Michael Reck, Anton Zeilinger, Herbert J. Bernstein, Philip Bertani, Phys. Rev. Lett. 73 (1994) 58–61, https://doi.org/10.1103/PhysRevLett.73.58.

[27] Eduardo R. Caianiello, Nuovo Cimento (Italy) divided into Nuovo Cimento A, Nuovo Cimento B 10 (1953), https://doi.org/10.1007/BF02781659.

[28] Scott Aaronson, Proc. R. Soc. A, Math. Phys. Eng. Sci. 467 (2136) (2011) 3393–3405, https://doi.org/10.1098/rspa.2011.0232.

[29] Daniel Grier, Luke Schaeffer, in: Rocco A. Servedio (Ed.), 33rd Computational Complexity Conference, CCC 2018, June 22-24, 2018, San Diego, CA, USA, in: LIPIcs, vol. 102, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018, pp. 19:1–19:29.

[30] Albert Nijenhuis, Herbert S. Will, Combinatorial Algorithms for Computers and Calculators, Academic Press, 1978.

[31] F.V. Gubarev, I.V. Dyakonov, M.Yu. Saygin, G.I. Struchalin, S.S. Straupe, S.P. Kulik, Phys. Rev. A 102 (2020) 012604, https://doi.org/10.1103/PhysRevA.102.012604.

[32] Andrea Olivo, Frédéric Grosshans, Phys. Rev. A 98 (4) (2018), https://doi.org/10.1103/physreva.98.042323.

[33] Benjamin Villalonga, Murphy Yuezhen Niu, Li Li, Hartmut Neven, John C. Platt, Vadim N. Smelyanskiy, Sergio Boixo, Efficient approximation of experimental Gaussian boson sampling, Available as arXiv:2109.11525, 2021.

[34] Mattia Walschaers, Jack Kuipers, Juan-Diego Urbina, Klaus Mayer, Malte Christopher Tichy, Klaus Richter, Andreas Buchleitner, New J. Phys. 18 (3) (2016) 032001, https://doi.org/10.1088/1367-2630/18/3/032001.

[35] Valery Shchesnovich, Quantum 5 (2021) 423, https://doi.org/10.22331/q-2021-03-29-423.

[36] Valery Shchesnovich, Boson sampling cannot be faithfully simulated by only the lower-order multi-boson interferences, Available as arXiv:2204.07792, 2022.

[37] T.C. Ralph, N.K. Langford, T.B. Bell, A.G. White, Phys. Rev. A 65 (2002) 062324, https://doi.org/10.1103/PhysRevA.65.062324.

[38] E. Knill, Phys. Rev. A 66 (2002) 052306, https://doi.org/10.1103/PhysRevA.66.052306.

[39] Mike Earnest, Extended stars-and-bars problem (where the upper limit of the variable is bounded), Mathematics Stack Exchange, https://math.stackexchange.com/q/3182858.

[40] Adrien Cassagne, Olivier Aumage, Denis Barthou, Camille Leroux, Christophe Jégo, in: Proceedings of the 2018 4th Workshop on Programming Models for SIMD/Vector Processing, 2018, pp. 1–8.

[41] Roger Fletcher, Practical Methods of Optimization, John Wiley & Sons, 2013.