.

Linear-Optical Feynman Paths (LOFP): code for strong simulation of Fock-state boson sampling

W. F. Balthazar[1, 2, *] and E. F. Galvão[1, 3, †]

[1] *International Iberian Nanotechnology Laboratory (INL), Av. Mestre José Veiga, 4715-330 Braga, Portugal.*

[2] *Instituto Federal do Rio de Janeiro, 27213-100, Volta Redonda, Rio de Janeiro, Brazil*

[3] *Instituto de Física, Universidade Federal Fluminense, Av. Gal. Milton Tavares de Souza s/n, Gragoatá, 24210-346 Niterói, Rio de Janeiro, Brazil*

(Dated: February 27, 2023)

Linear-Optical Feynman Paths (LOFP) is a C++ package for the strong simulation, that is, the exact calculation of probability amplitudes, of Fock-state boson sampling experiments on planar multimode interferometers. LOFP uses a Feynman path sum approach, together with tensor contraction. LOFP also includes routines with implementations of general-purpose permanent calculation algorithms for benchmarking purposes and examples. The code has better performance than general algorithms for permanent calculation in some cases, in particular for constant-depth circuits, and high density of photons per mode.

I. INTRODUCTION

This document provides brief documentation on code to simulate Fock-state boson sampling [1, 2] using Feynman's path sum formalism ([3]). The main idea is to sum the probability amplitudes over all possible paths to find the amplitude associated with a fixed choice of input and output states, and multi-mode interferometer parameters. Our code simulates a local-connectivity, planar design for multimode interferometers, in particular, depth-shortened versions of the universal design by Clements et al. [4]. The code is written in C++ and is available at the Github link `https://github.com/wagnerbalthazar/Linear_Optics_Feynman_Path` under an open-source Creative Commons Attribution 4.0 Internacional License. A paper with a complete description of the simulation algorithm will be available soon [5].

II. AVAILABLE ROUTINES

In this section, we have presented a brief description of the routines available in the package. The main routine implements an algorithm based on Feynman path sums for strong simulation of Fock-state boson sampling in local-connectivity planar interferometer designs. The additional routines available include code to evaluate amplitudes by calculating the permanents of matrices, using Ryser and Glynn's formulas. We also provide code with examples used in the benchmark section to illustrate how one may use the code.

* wagner.balthazar@inl.int

† ernesto.galvao@inl.int

A. Main routine

- Probability_Feynman – This package contains a source file with all functions to calculate the probability of a Fock-state boson sampling experiment for chosen input/output states, and a given interferometer made out of a finite number of locally-connected beam-splitter layers.

B. Additional routines

- Probability_Ryser – This source file calculates the probability of observing an input mode $|s\rangle$ go through a multi-mode planar interferometer built out of some number of locally-connected beam-splitter layers, represented by the unitary U, and observing the output state $|t\rangle$. The matrix U is defined by setting up the beam-splitters parameters with the function "*bs_parameters*" and using the function "*mat_unitary_circuit*". After, we need to find the submatrix U_{st} of U, which is defined by repeating t_i times the ith columns of U to obtain U_s, and repeating s_j times the jth row of U_s to obtain U_{st}. After, the code calls the function *perm_ryser*, based on Ryser's formula, to calculate the permanent of the matrix U_{st}. Finally, the *probability_Ryser* function is used to obtain the boson sampling probabilities [2].
- Probability_Glynn – This source file does the same process described for Ryser's code. The difference is the permanent is calculated using Glynn's formula [6] Therefore, *perm_glynn* and *probability_Glynn* functions must be called.
- Example_1 – The code Example_1_correctness.cpp is used to verify the correctness of the code by com-

paring Feynman, Glynn, and Ryser. This is completely described in section V.A.

- Example_2 – The code Example_2_increase_modes_photons.cpp is used to verify, for random interferometers with different depths, the code runtime as the number of modes increases, always with 1 photon per mode. This is completely described in section V.B.

- Example_3 – The code Example_3_increase_depth.cpp is used to study the runtime and memory as we increase the number of layers, keeping the number of modes constant and equal to 8. This is completely described in section V.C.

- Example_4 – The code in the file Example_4_Higher_input.cpp is used to study the runtime and memory increasing the photon density, i.e. the number of photons per mode. This is completely described in section V.D.

- Example_5 – The code Example_5_read_csv_bs_parameters.cpp illustrates how one may input arbitrary beam-splitter parameters from a local file. This example uploads information on a six-mode, depth-5 interferometer, stored in two CSV files (one for transmissivity parameters θ, the other for phase shifter parameters ϕ - see Fig. 2).

III. BASIC CONCEPTS

We simulate Fock-state boson sampling in multimode interferometers. The interferometers are built by locally connecting layers of beam-splitters (BS's) as in Fig. 1. If the number of layers is the same number of modes, this reproduces a universal design for linear optics, proposed by Clements et al. [4].

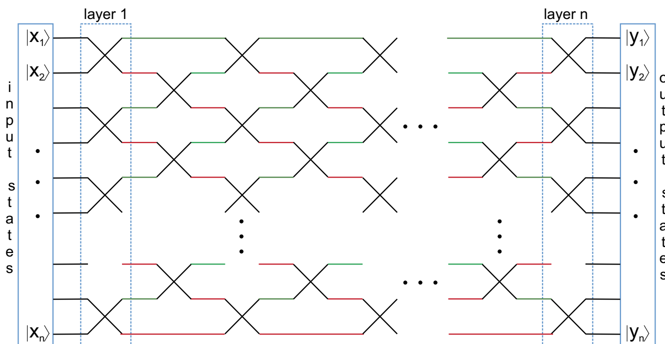

FIG. 1. Planar interferometer design consisting in n layers of locally-connected beam-splitters.

The input (output) states are defined by $|x_m\rangle = |x_1, ..., x_n\rangle$ ($|y_m\rangle = |y_1, ..., y_n\rangle$), where m is the number of modes and integer $x(y)$ is the photon occupation number in the respective mode. Each layer of the circuit consists of a column of beam-splitters, highlighted with a blue rectangle in Fig 1. The number of layers defines the depth of the circuit. For our simulation, we can pick any number of even modes $m \geq 2$, and any depth ≥ 2.

Any multi-mode photonic interferometer can be decomposed in fundamental building blocks, which are beam-splitters (in Fig. 2, these are black, X-shaped waveguide structures) with transmissivity t and reflectivity r, with amplitudes defined by parameter θ, and by a ϕ parameter that corresponds to the phase shift of one output mode with respect to the other. The amplitudes are given by $t = \cos\theta$ and $r = e^{i\phi}\sin\theta$. Fig. 2 shows the beam splitter and its corresponding phase shifter, even though the phase shifters are omitted in Fig. 1.

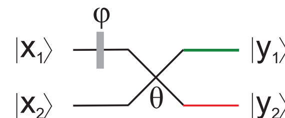

FIG. 2. Beam-splitter parameters θ (defining transmissivity) and ϕ (phase shift).

This beam splitter implements a general SU(2) unitary transformation on creation operators, represented by the following 2 x 2 matrix:

$$BS(\theta, \phi) = \begin{bmatrix} \cos(\theta) & -e^{-i\phi}\sin(\theta) \\ e^{i\phi}\sin(\theta) & \cos(\theta) \end{bmatrix},$$

where $\theta \in [0, \pi/2]$ and $\phi \in [0, \pi]$.

Fig. 1 also shows a coloring of the waveguides connecting pairs of beam-splitters. This coloring is important for our goals. In Feynman's approach to the description of general quantum dynamics, we need to sum over all possible transition amplitudes to obtain the total event amplitude. For fixed values of the input and output photon occupation numbers (x_1, x_2, y_1, y_2), we can calculate the transition probability for this single beam splitter in a runtime that scales linearly with the total number of photons (as described in [5]).

The next task is to find all possible Feynman paths for fixed inputs and outputs of an interferometer with the structure shown in Fig. 1. In our context, a Feynman path consists of a choice for occupation numbers for all green and red waveguides in the figure; there are all waveguides interconnecting different BS in the linear-optical network. This can be done in two steps. First, we find bounds on the maximum photon occupation numbers in each green waveguide. This can be done by looking at the past and future light-cones of the waveguide in question, as shown in Fig. 3. The green waveguide highlighted here is constrained to have occupation numbers compatible with these two light cones, which connect it to its past (the fixed input) and its future (the fixed output). A little thought shows that the photon number occupation in red waveguides is constrained by the fixed input

and output, and by fixed values for the green waveguide occupations numbers, due to photon number conservation at each beam-splitter. So looping over all paths, or waveguide photon occupation numbers, corresponds to looping over all possible occupation numbers for the green waveguides only, and setting the corresponding occupation numbers for red waveguides.

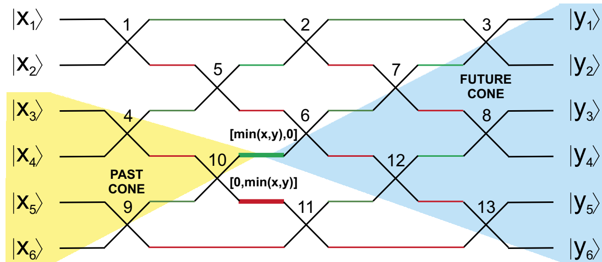

FIG. 3. Considering the past and future light cones of each green-colored waveguide, to constrain its possible photon occupation numbers. The beam-splitters are numbered in the order used for the input file of beam-splitter parameters.

We then see that the amplitude for each path is the product of the amplitudes associated with all BS in the network. The sum of the calculated amplitudes of each configuration is the probability amplitude we want to calculate, for fixed input and output, and for a particular choice of interferometer parameters. It can be expressed as follows:

$$\langle y| U |x\rangle = \sum_{x,y} \prod_{i=1}^{N} \langle x_{1i}, x_{2i}| U_{BS(\theta,\phi)_i} |y_{1i}, y_{2i}\rangle , \qquad (1)$$

where N is the total number of beam-splitters, $\langle x_{1i}, y_{1i}, y_{2i}|$ and $|y_{1i}, y_{2i}\rangle$ corresponds to the input and output of each beam splitter, and $U_{BS(\theta,\phi)_i}$ is the unitary specifying the beam splitter action. Each BS amplitude can be computed in a time that scales linearly with the number of photons going through it [5]; the overall complexity comes from the sums over all possible photon occupation numbers that need to be considered. While in principle the memory scaling is polynomial with the number of photons, modes, and depth, in our implementation we use a tensor contraction method that incurs in memory scaling exponentially with depth, with the benefit of a faster runtime, as will be illustrated in the next section.

IV. RUNNING THE CODE

A. Probability_Feynman

To run the source code called *prob_feynman.cpp*, you need to define the initial conditions in the main function: the input state, the output state, the number of modes, and the depth of the circuit following the description below:

1. the input state – a vector of non-negative integers specifying the input Fock occupation number at each input mode. For example, in an interferometer with 8 modes and 1 photon per mode, the input is written as follows:

```
const vector<int> input = { 1,1,1,1,1,1,1,1 };
```

2. the output state – vector specifying the Fock-state occupation numbers of the output. For example, in an interferometer with 8 modes, one possible output of 8 photons could be written as follows:

```
const vector<int> output = { 2,0,0,3,0,1,1,1 };
```

3. the (even) number of modes, which is defined by the length of the input or output vector, as follows:

```
const int modes = input.size();
```

4. the depth of the circuit, that is, the number of BS layers ≥ 2, as follows:

```
int depth = 5.
```

After these definitions, it is necessary to specify all parameters describing the beam-splitters that comprise the interferometer. This can be done in three different ways:

5. Choosing all beam-splitters to be balanced, 50/50 beam-splitters – in this case, the option **false** must be chosen as a parameter of the function bs_parameters.

```
vector<vector<vector<complex<double>>>> beam_splits = bs_parameters(false , modes, depth);
```

6. Picking uniformly random parameters for all beam-splitters – in this case, the option **true** must be chosen as a parameter of the function bs_parameters.

```
vector<vector<vector<complex<double>>>> beam_splits = bs_parameters(true, modes, depth);
```

7. Defining the parameters of each beam-splitter by hand – in this case, the function parameters **t_1** and **t_2** must be typed, and the function bs_parameters_2 must be chosen. Considering depth = 5 and 8 modes, we have 18 beam-splitters to be configured, so we need two vectors with 18 elements each. Note that **t_1** corresponds to the θ parameter of the beam-splitter, and **t_2** corresponds to the ϕ parameter of each beam splitter, see Eq. III and Fig. 2.

```
vector<double> t1 = {0.4595912, 0.85014466, 0.05414373, 0.56944304,
0.34649183, 0.26516494, 0.75055521, 1.50788223,
1.29585152, 0.26597035, 0.5081206, 1.50363248,
```

0.27852718, 0.13658876, 0.89655676, 1.49307613, 0.45337888, 0.03398879}

vector<double> t2 = {0.93827935, 2.46627691, 1.59271753, 0.10410264, 0.9995343, 2.48067184, 0.62522083, 3.09866259, 1.90669285, 2.8284845, 2.00854026, 0.55957266, 0.38713007, 0.90334558, 1.99934321, 3.00567887, 0.00134568, 1.13459999};

vector<vector<vector<complex<double>>>> beam_splits = bs_parameters_2(**t_1**, **t_2**, modes, depth);

8. Alternatively, the parameters t_1 and t_2 of all beam-splitters can be uploaded from two CSV files provided by the user. We illustrate the procedure with the code in file Example_5_read_csv_bs_parameters.cpp, which simulates a depth-5 interferometer with six modes. We also offered two CSV files (θ and ϕ parameters) to illustrate how to upload parameters from user-defined files.

The phases θ and ϕ must be specified in the order described in Fig. 3, where we have the example of a multi-mode interferometer with depth 5. The beam-splitters on the top are numbered in increasing order from left to right. After those, we continue the numbering by the second beam splitter in the first layer, then the next unnumbered one to the right of it, following this order from left to right. The same process continues with the third beam-splitter in the first layer, and so on until we have numbered all beam-splitters.

After defining the inputs, the probability can be calculated using the LOFP algorithm:

9. cout << "Probability Feynman = " << probability_Feynman(input, output, depth, beam_splits) << endl;

Fig. 4 shows the screen with the same example used above for the case with random beam-splitters in the main function. The only difference is the chrono library which we use to get the code runtime.

```
int main(int argc, char* argv[]) {

    // Initial conditions
    const vector<int> input = { 1,1,1,1,1,1,1,1 };
    const vector<int> output = { 2,0,0,3,0,1,1,1 };
    const int modes = input.size();
    int depth = 5;
    vector<vector<vector<complex<double>>>> beam_splits = bs_parameters(true, modes, depth);

    // Probability and time Feyman.
    auto start = chrono::high_resolution_clock::now();
    cout << "Probabilty Feynman = " << probability_Feynman(input, output, depth, beam_splits) << endl;
    auto end = chrono::high_resolution_clock::now();
    auto duration = std::chrono::duration_cast<std::chrono::microseconds>(end - start);
    double duration_seconds = static_cast<double>(duration.count()) / 1000000.0;
    std::cout << "Time taken Feynman: " << duration.count() / 1000000.0 << " seconds" << std::endl;

    return 0;
}
```

FIG. 4. Example of the main function with random beam-splitters.

B. Probability_Ryser

To run the code *probability_Ryser* presented in Section II.B, the first step is to set up the initial conditions of the multi-mode interferometer following the same steps described above for Probability_Feynman. This involves defining: the input and output states, the number of modes, the depth, and the parameters describing all beam-splitters.

After this, we can find the unitary matrix U describing the interferometer we have just specified as follows:

1. vector<vector<complex<double>>> unitary_circuit = mat_unitary_circuit(beam_splits, depth, modes).

The next step is to use the unitary matrix U and the input and output states already defined to find the matrix U_{st}. It is done by writing the function:

2. vector<vector<complex<double>>> unitary_st = Unitary_ST(unitary_circuit, input, output);

After, the code calls the function "*perm_ryser*", based on Ryser's formula, to calculate the permanent of the matrix U_{st}. Finally, the "*probability_Ryser*" as follows:

3. cout << "Probability Ryser = " << probability_Ryser(unitary_st, input, output, depth) << endl;

Fig. 5 shows the main function to run the code and find a probability.

```
int main(int argc, char* argv[]) {
    // Initial conditions
    const vector<int> input = { 1,1,1,1,1,1 };
    const vector<int> output = { 1,1,0,3,1,0};
    const int modes = input.size();
    int depth =7;

    vector<vector<vector<complex<double>>>> beam_splits = bs_parameters(true, modes, depth);
    vector<vector<complex<double>>> def = mat_unitary_circuit(beam_splits, depth, modes);
    vector<vector<complex<double>>> ghi = Unitary_ST(def, input, output);

    //Probability and time Ryser.
    auto start1 = chrono::high_resolution_clock::now();
    cout << "Probability Ryser = " << probability_Ryser(ghi, input, output, depth) << endl;
    auto end1 = chrono::high_resolution_clock::now();
    auto duration1 = chrono::duration_cast<chrono::microseconds>(end1 - start1);
    double duration1_seconds = static_cast<double>(duration1.count()) / 1000000.0;
    cout << "Time taken Ryser: " << duration1.count() / 1000000.0 << " seconds" << std::endl;


    return 0;
}
```

FIG. 5. Example of the main function to calculate the probability of boson sampling for one amplitude using Ryser's formula.

Additionally, this same code can be used to calculate the permanent of an arbitrary matrix with complex entries using Ryser's formula. In this case, we just need to input the *matrix* as follows:

4. complex <double> permanent = complex <double> perm_ryser(*matrix*);

C. Probability_Glynn

Here, we are doing the same step-by-step procedure described for Ryser, with one difference, using Glynn's formula [6] to evaluate the permanent. In this sense, only need to change the name of functions that contain Ryser to Glynn. So, we use the same functions used before and the new ones: perm_glynn($parameters$) and probability_Glynn($parameters$). Note that the $parameters$ are the same used in Ryser's case.

V. BENCHMARK - COMPARING LOFP WITH RYSER'S AND GLYNN'S FORMULAS

In this section, we have used LOFP and the well-known Ryser and Glynn's formula to benchmark LOFP's performance. The Ryser and Glynn codes were based on the description in [6].

All calculations were done using a laptop computer with the following specifications: Processor – Intel(R) Core(TM) i7-8565U, CPU @ 1.80GHzb – 1.99 GHz; Installed RAM: 16.0 GB (15.8 GB usable); System type: 64-bit operating system, x64-based processor.

A. Verifying correctness

To test the reliability of the Feynman code, we choose an input and compute the probability associated with all possible outputs. These probabilities must then add up to 1. This test was done using the Feynman algorithm, as well as Ryser's. We also calculated the total variation distance between pairs of distributions calculated using different methods:

$$\delta(p,q) = \frac{1}{2}\sum_i |p_i - q_i|, \tag{2}$$

where p_i (q_i) are the probabilities obtained with Feynman's method (Ryser). We consider a randomly picked six-mode interferometer, for different depths of 3 to 6. Fig. 6 shows the total variation distance obtained by finding the whole output distribution for two different inputs: $\text{input}_1 = \{1,1,1,1,1,1\}$; $\text{input}_2 = \{2,0,0,2,0,2\}$.

As we can see, the code is very accurate, obtaining the distributions with an error compatible with the round-off error of the numerical accuracy, of the order of 10^{-13}.

B. Increasing the number of photons and modes for different depths

In this simulation, we run the code increasing the number of modes from 4 to 22 in steps of two, always with 1 photon per mode. We do this for circuit depths $3, 4, 5, 6$.

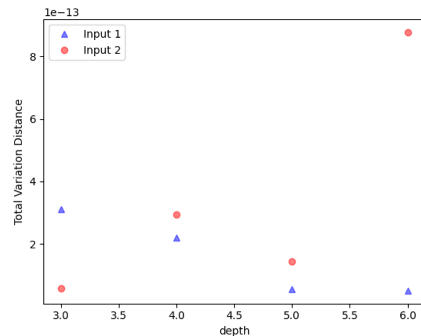

FIG. 6. Total variation distance between complete output distributions calculated with LOFP code and Ryser's algorithm, for two different inputs.

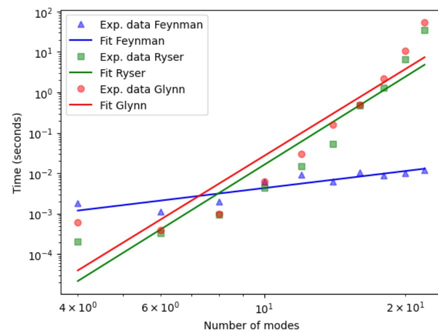

FIG. 7. Run-time for three-layer interferometers, one photon per mode at input and output, as the number of photons = number of modes increases. Comparison between calculation time using Feynman paths, Glynn's formula, and Ryser's algorithm.

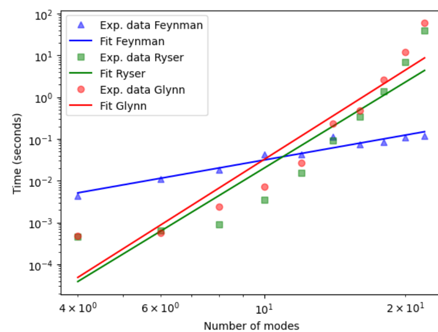

FIG. 8. Run-time for four-layer interferometers, one photon per mode at input and output, as the number of photons = number of modes increases. Comparison between calculation time using Feynman paths, Glynn's formula, and Ryser's algorithm.

Figs. 7 - 10 show the runtime as the number of modes increases.

As we can see, Ryser's algorithm and Glynn's formula are faster than Feynman's approach when we have a small number of modes (photons), but Feynman's path sum

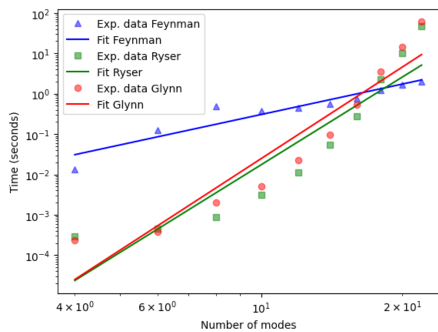

FIG. 9. Run-time for five-layer interferometers, one photon per mode at input and output, as the number of photons = number of modes increases. Comparison between calculation time using Feynman paths, Glynn's formula, and Ryser's algorithm.

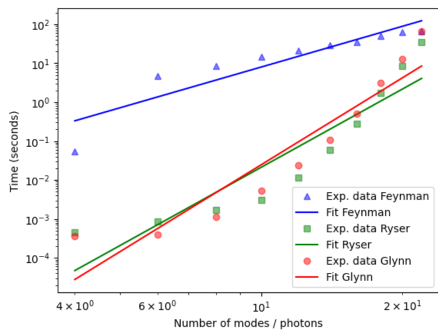

FIG. 10. Run-time for six-layer interferometers, one photon per mode at input and output, as the number of photons = number of modes increases. Comparison between calculation time using Feynman paths, Glynn's formula, and Ryser's algorithm.

shows an advantage as we increase the number of modes, at least for small depth circuits. Even in Fig. 10, we can see that from 22 photons Feynman code starts to beat Ryser's algorithm and Glynn's formula. This happens because both Ryser's and Glynn's formulas have runtime that increases exponentially with the number of photons, which is independent of the interferometer depth. With LOFP the number of photons can be much larger, but the simulation runtime grows exponentially with the depth.

Another advantage of Feynman's path sum is that the time increases linearly with the number of modes, due to a tensor contraction that is performed as part of the routine [5]. Fig. 11 shows the Feynman path sum runtime for different circuit depths.

C. Increasing the depth of the optical circuit

In this subsection, we simulate a multi-mode interferometer with 8 input modes and with 1 photon per mode. The depth varies from 3 to 7. Figs. 12 and 13 show in a log-log plot that time and memory grow exponentially

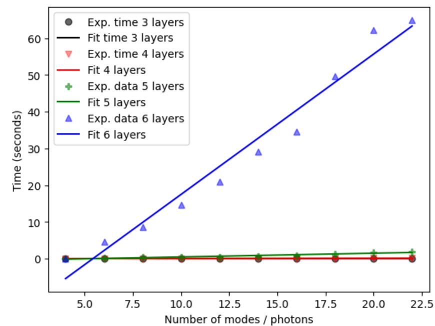

FIG. 11. Run-time for a single amplitude calculation on random interferometers with varying depth, and an increasing number of modes. We pick one mode per photon at the input and output.

with the depth, but the same does not occur with the number of modes/photons.

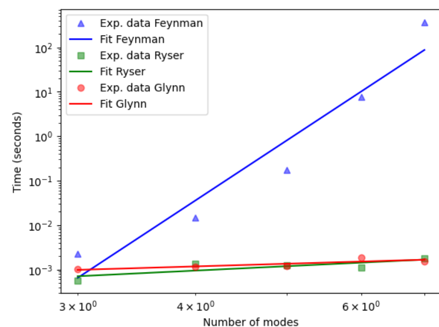

FIG. 12. Runtime scaling for Feynman path sum calculation of a single amplitude for an 8-mode interferometer, with 1 photon per mode at the input and output, as a function of the depth.

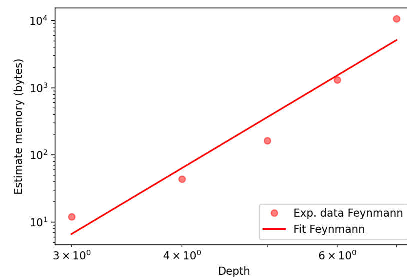

FIG. 13. Memory use scaling for Feynman path sum calculation of a single amplitude for an 8-mode interferometer, with 1 photon per mode at the input and output, as a function of the depth.

The Feynman path sum algorithm with tensor contraction uses memory that increases exponentially with the depth, but not with the number of modes. For the limited depth interferometers we simulate here, the total memory use of only the order of a few kilobytes.

D. Higher input photon occupation numbers, depth 3 and 4

In this subsection, we increase the number of input photons from 1 to 20 per mode, considering a circuit with 6 modes, and depths 3 and 4. The total number of photons is then between 6 and 120. Figs. 14 and 15 show plots of runtime and memory, as a function of the number of photons per mode.

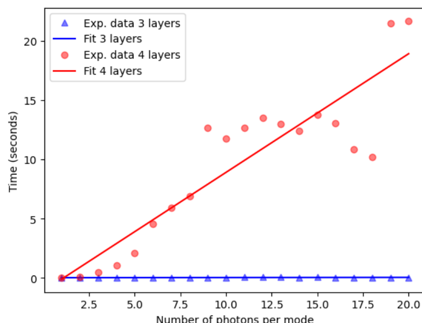

FIG. 14. Runtime as a function of the number of photons per mode, for a 6-mode interferometer and depths 3, 4.

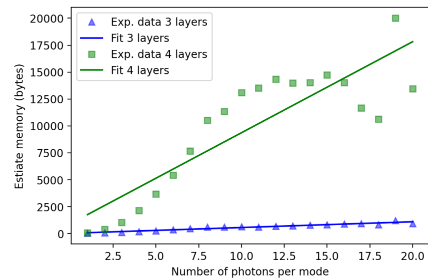

FIG. 15. Memory use for the Feynman path sum simulator, as a function of the number of photons per mode, for a 6-mode interferometer and depths 3, 4.

These last two plots show computational evidence that the runtime and memory use of the Feynman path sum algorithm increases linearly with the number of photons, for these limited-depth interferometers. The memory used is small, on the order of a few kilobytes. The runtime dependence on the number of photons is in stark contrast with both Ryser's algorithm and Glynn's formula, whose runtimes scale exponentially with the total number of photons.

VI. CONCLUSIONS

We introduce LOFP code that simulates Fock-state boson sampling experiments using Feynman's path sums and tensor contraction. We illustrated some situations where our code presents some advantages when compared with Ryser's algorithm and Glynn's formula. We can highlight the results for high photon density per mode and small depth, where the Feynman path sum code is faster than the alternatives. The results indicate that for those cases, runtime and memory use increase linearly with the number of modes/photons. A detailed description of the tensor-network contraction and other variations of the Feynman path sum concept for the simulation of linear optics will appear elsewhere [5].

ACKNOWLEDGMENTS

We would like to thank Filipa Peres for helpful discussions. We acknowledge the financial support of H2020-FETOPEN Grant PHOQUSING (GA no.: 899544).

[1] S. Aaronson and A. Arkhipov, arXiv:quant-ph/1011.3245v1 (2011).
[2] D. J. Brod, E. F. Galvão, A. Crespi, R. Osellame, N. Spagnolo, and F. Sciarrino, Advanced Photonics **1**, 034001 (2019).
[3] S. Aaronson and L. Chen, arXiv preprint arXiv:1612.05903 (2016).
[4] W. R. Clements, P. C. Humphreys, B. J. Metcalf, W. S. Kolthammer, and I. A. Walmsley, Optica **3**, 1460 (2016).
[5] Q. Palmer, J. Bulmer, A. Jones, and E. F. Galvão, manuscript in preparation (2023).
[6] D. G. Glynn, European Journal of Combinatorics **31**, 1887 (2010), ISSN 0195-6698, URL `https://www.sciencedirect.com/science/article/pii/S0195669810000211`.