Introduction and Basics

Status	Done

Part 1: Building a basic GBS circuit

Background

- Boson and fermions, are the two elementary classes of particles.
- The most prevalent bosonic system in our everyday lives is light. The distinguishing characteristic of bosons is that they follow “Bose-Einstein statistics” - the particles like to bunch together (contrast this to fermionic matter like electrons, which must follow the Pauli Exclusion Principle and keep apart).
- **Boson Sampling:** Aaronson and Arkhipov’s original proposal was to inject many single photons into distinct input ports of a large interferometer, then measure which output ports they appear at. The natural interference properties of bosons means that photons will appear at the output ports in very unique and specific ways.
- Physical implementations of boson sampling make use of a process known as spontaneous parametric down-conversion to generate the single-photon source inputs. However, this method is non-deterministic — as the number of modes in the apparatus increases, the average time required until every photon source emits a simultaneous photon increases exponentially.
- Hence we now use states of light that are experimentally less demanding (though still challenging!). These states of light are called Gaussian states, because they bear strong connections to the Gaussian (or Normal) distribution from statistics. In practice, we use a particular Gaussian state called a squeezed state for the inputs.
- In this case, single mode squeezed states — can produce problems in the same computational complexity class as boson sampling. Even more significantly, this negates the scalability problem with single photon sources, as single mode squeezed states can be deterministically generated simultaneously.

Squeezed Gaussian States

Our starting point is the vacuum state. Other states can be created by evolving the vacuum state according to:

$$|\psi\rangle = exp(-itH)|0\rangle$$

where H is a bosonic Hamiltonian and t is the evolution of time.

- State where the Hamiltonian is at most quadratic in $\hat{x}\ and\ \hat{p}$ are called **Gaussian.**
- For a single qumode, Gaussian states are parameterized by 2 continuous complex variables.
  1. Displacement parameter: $\alpha \in C$
  2. Squeezing parameter: $z \in C\ (z = re^{i\phi}\ with\ r \geq 0)$
- Given a Gaussian distribution, we can identify each state, through it's displacement (centre of Gaussian) and squeezing (variation and rotation) parameters.
- Squeezed states are a family of Gaussian states where the displacement is 0 and squeezing parameter is free.

Some useful points

- Unitary operations can always be associated with a generating Hamiltonian. Gaussian are those with degree one or two of the generating Hamiltonians.

$$U = exp(-itH)$$

- Number states **(Fock states)** are eigenstate of the number operator $\hat{n} = \hat{a}^\dagger \hat{a}$. They form a discrete countable basis for a single qumode. And all Gaussian states can be written in this basis.
- Note that a single-mode squeezed vacuum state is a superposition of only **even photon number** states. Therefore, if you measure the photon number in a single squeezed vacuum mode, you will only ever detect an even number of photons (0, 2, 4, ...).

A squeezed state is a minimum uncertainty state with unequal quadrature variances, and satisfies the following eigenstate equation:

$$\left(\hat{a}\cosh(r)+\hat{a}^{\dagger}e^{i\phi}\sinh(r)\right)|z\rangle=0$$

In the Fock basis, it has the decomposition

$$|z\rangle=\frac{1}{\sqrt{\cosh(r)}}\sum_{n=0}^{\infty}\frac{\sqrt{(2n)!}}{2^n n!}(-e^{i\phi}\tanh\,(r))^n|2n\rangle$$

whilst in the Gaussian formulation, $\bar{\mathbf{r}}=(0,0)$, $\mathbf{V}=\frac{\hbar}{2}R(\phi/2)\begin{bmatrix}e^{-2r}&0\\0&e^{2r}\end{bmatrix}R(\phi/2)^T$.

We can use the squeezed vacuum state to approximate the zero position and zero momentum eigenstates;

$$|0\rangle_x\approx S(r)\,|0\rangle\,,\quad |0\rangle_p\approx S(-r)\,|0\rangle$$

where $z=r$ is sufficiently large.

- If you have just one qumode (a single quantum mode) prepared in a squeezed vacuum state, and you increase your simulation cutoff high enough, you will see a non-zero probability of detecting 8 photons (or any other even number of photons) from that single mode. That is the probability generally decreases as the photon number increases, but it's *never zero* for any even number (theoretically, it extends infinitely).

Hafnian

- This the source of the hardness in GBS.
- Finding the permanent if #P-Hard and by extension so is Boson Sampling.
- From bellow it is clear that for similar reasons GBS falls under the same difficulty class.

Using phase space methods, Hamilton et al. [2] showed that the probability of measuring a Fock state containing only 0 or 1 photons per mode is given by

$$\left|\langle n_1, n_2, \ldots, n_N|\psi'\rangle\right|^2 = \frac{\left|\mathrm{Haf}[(U(\bigoplus_i \tanh(r_i))U^T)]_{st}\right|^2}{\prod_{i=1}^N \cosh(r_i)}.$$

i.e., the sampled single photon probability distribution is proportional to the **hafnian** of a submatrix of $U(\bigoplus_i \tanh(r_i))U^T$, dependent upon the output covariance matrix.

Note

The hafnian of a matrix is defined by

$$\mathrm{Haf}(A) = \frac{1}{n!2^n}\sum_{\sigma\in S_{2N}}\prod_{i=1}^{N} A_{\sigma(2i-1)\sigma(2i)}$$

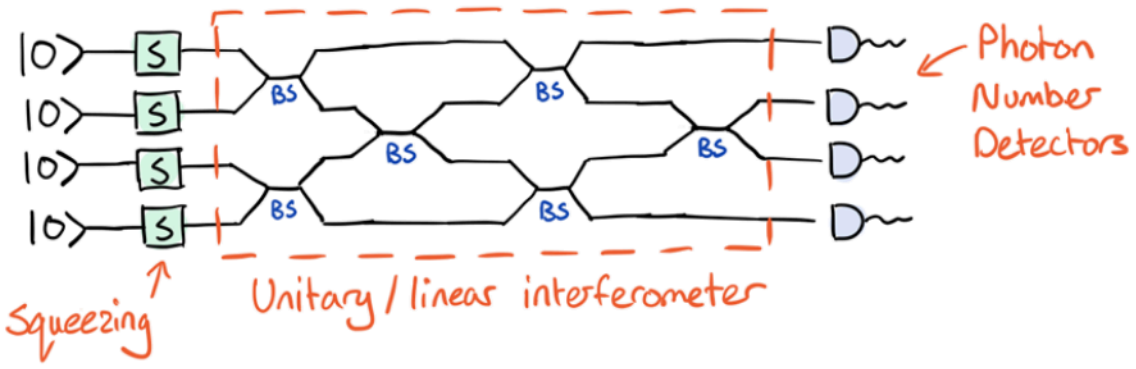
where S_{2N} is the set of all permutations of $2N$ elements. In graph theory, the hafnian calculates the number of perfect matchings in an **arbitrary graph** with adjacency matrix A.

Compare this to the permanent, which calculates the number of perfect matchings on a *bipartite* graph - the hafnian turns out to be a generalization of the permanent, with the relationship

$$\mathrm{Per(A)} = \mathrm{Haf}\left(\begin{bmatrix} 0 & A \\ A^T & 0 \end{bmatrix}\right)$$

As any algorithm that could calculate (or even approximate) the hafnian could also calculate the permanent - a #P-hard problem - it follows that calculating or approximating the hafnian must also be a classically hard problem.

Basic Circuit Outline



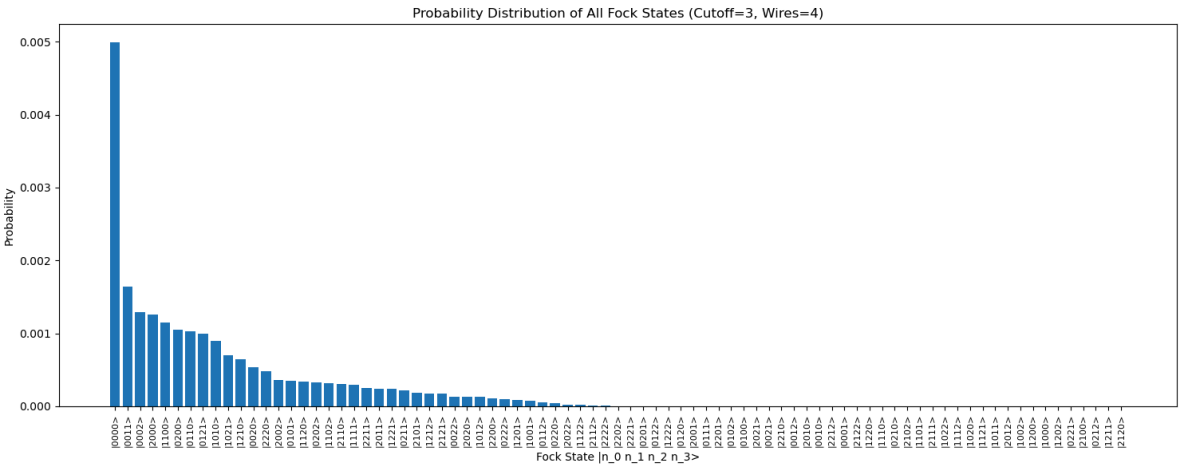

```
import numpy as np
np.random.seed(42)

import strawberryfields as sf
from strawberryfields.ops import *
from scipy.stats import unitary_group
```

```
n_wires = 4
cutoff = 3 #The cutoff is crucial here for calculating Fock probabilities
#Otherwise we will end up with infinite possibilities as discussed before.

#Initialize the circuit
boson_sampling_circuit = sf.Program(n_wires)

#Define the linear interferometer
#Note that this function always gives a unitary that can be implemented in linea
r option and hence be Gaussian.
U_interferometer = unitary_group.rvs(4)
print(U_interferometer)

#Output of the above code looks like this
'''[[ 0.06775892+0.16300837j -0.2195115 +0.88035426j  0.06882153+0.127909
45j
   0.35159543-0.03019121j]
 [-0.27973364+0.30193374j  0.1577798 +0.28412608j  0.00744449-0.4669026
j
  -0.48589977+0.52039502j]
 [ 0.11323677-0.24459285j  0.02931785-0.11273776j  0.77764982-0.19091662j
   0.34318679+0.39346774j]
 [-0.43602107+0.73257048j  0.0831171 -0.22514706j  0.30707269+0.1592711j
   0.22356318-0.21444127j]]'''

# Construct the circuit:
#    a. Apply Squeezed Vacuum states
#    b. Apply the Interferometer
#    c. Measure in the Fock basis
with boson_sampling_circuit.context as q:
    # Apply Squeezed Vacuum states to each mode
    # The 'r' parameter is the squeezing parameter (strength of squeezing)
    # A common range is 0.1 to 1.5, or even higher for strong squeezing.
    # We can apply a S2gate (two-mode squeezing) for a single mode by leaving
the second mode implicit, or by applying Sgate for single-mode squeezing. S2g
ate is often used in context of initial states for GBS if you consider paired mode
s, but for a simple squeezed vacuum state per mode, Sgate is more direct.
    Sgate(1) | q[0]
    Sgate(1) | q[1]
    Sgate(1) | q[2]
```

```
    Sgate(1) | q[3]

    # The Interferometer (the unitary matrix U) operation takes the unitary matrix
and applies it across the specified wires.
    Interferometer(U_interferometer) | tuple(q) # Apply U to all wires

    # Measure in the Fock basis (photon number resolving detection)
    # This will return the number of photons detected in each of the 4 modes.
    MeasureFock() | q

#Select the Strawberry Fields Gaussian backend
eng = sf.Engine('gaussian')

#'shots' defines how many times to run the experiment to collect samples
num_shots = 1000
results = eng.run(boson_sampling_circuit, shots=num_shots)

print(f"\nSampled photon number outcomes (first {min(10, num_shots)} out of
{num_shots}):")
print(results.samples[:min(10, num_shots)]) # Print the first few samples

# You can also get other information, like the means of the distribution
# if you don't perform MeasureFock (e.g., if you only simulate the Gaussian stat
e).
# For GBS, the 'samples' are the primary output.

#The output here looks like this
'''
Sampled photon number outcomes (first 10 out of 1000):
[[0 0 0 0]
 [2 1 0 3]
 [0 0 0 2]
 [0 2 0 2]
 [0 0 0 2]
 [0 2 0 0]
 [0 0 0 0]
 [3 2 4 1]
 [4 2 0 0]
 [0 0 0 0]]'''
```

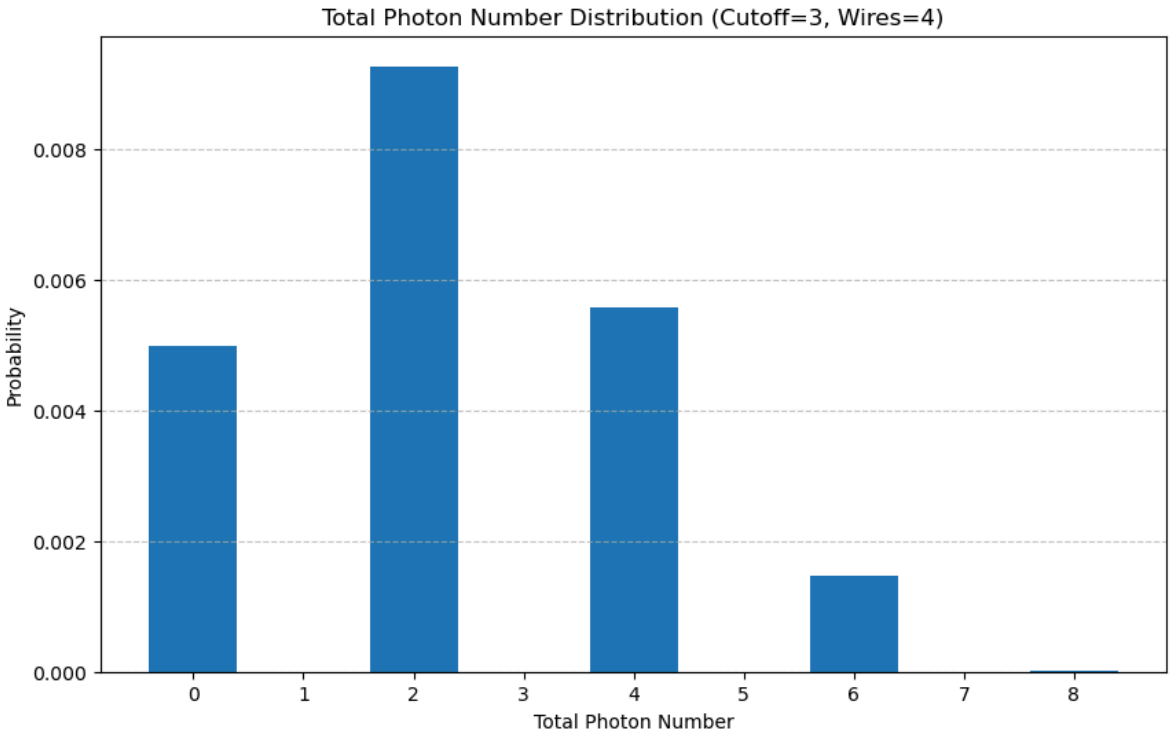
```
 #Instead if we were to the following we get the associated probability for each
state
 #Construct the circuit:
with boson_sampling_circuit.context as q:
    Sgate(1) | q[0]
    Sgate(1) | q[1]
    Sgate(1) | q[2]
    Sgate(1) | q[3]

    Interferometer(U_interferometer) | tuple(q)

#Select the Strawberry Fields Gaussian backend
eng = sf.Engine('gaussian')

# When not specifying 'shots', eng.run() returns a Result object containing the
quantum state.
results = eng.run(boson_sampling_circuit)

state = results.state #This is a BaseGaussianState object

#Calculate the Fock state probabilities using all_fock_probs()
#This method takes the cutoff as a keyword argument
flat_probs = state.all_fock_probs(cutoff=cutoff)

#Reshape the flat probabilities into a multi-dimensional tensor
probs = flat_probs.reshape([cutoff] * n_wires)
print(f"\nShape of the probability tensor (probs.shape): {probs.shape}")

#Define the Fock states to measure
measure_states = [
    (0, 0, 0, 0),
    (1, 1, 0, 0),
    (0, 1, 0, 1),
    (1, 1, 1, 1),
    (2, 0, 0, 0),
]

print("\nProbabilities of specific Fock states:")
for i in measure_states:
    if all(photon_num < cutoff for photon_num in i):
```

```
        prob = probs[i]
        if prob < 1e-10:
            print(f"|{''.join(str(j) for j in i)}>: ~0 (exact: {prob:.2e})")
        else:
            print(f"|{''.join(str(j) for j in i)}>: {prob:.6f}")
    else:
        print(f"|{''.join(str(j) for j in i)}>: (State not in current Fock space cutoff of {cutoff})")

#This output of the code looks as follows
'''
Shape of the probability tensor (probs.shape): (4, 4, 4, 4)

Probabilities of specific Fock states:
|0000>: 0.176378
|1100>: 0.007339
|0101>: 0.000611
|1111>: 0.000597
|2000>: 0.032246
'''
```



Note that the above values can be verified by calculating the Hafnian and using the above mentioned measurement formula to find the obtained values.

We can use Walrus library for better calculation of the Hafnian and Mr Musturd for more accurate and faster simulations of the measurements.



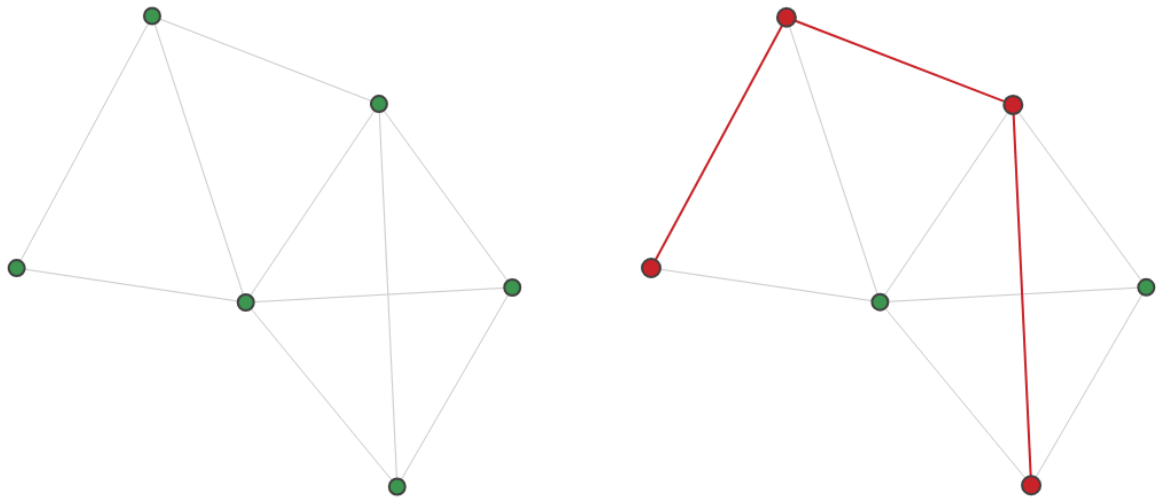
Part 2: Sampling Subgraphs from GBS

A GBS device can be programmed to sample from any symmetric matrix A.

Sampling functionality is provided in the `sample` module.

```
from strawberryfields.apps import plot
import networkx as nx
import plotly

adj = np.array(
    [
        [0, 1, 0, 0, 1, 1],
        [1, 0, 1, 0, 1, 1],
        [0, 1, 0, 1, 1, 0],
        [0, 0, 1, 0, 1, 0],
        [1, 1, 1, 1, 0, 1],
        [1, 1, 0, 0, 1, 0],
    ]
)
```

```
graph = nx.Graph(adj)
plot.graph(graph)

#This is a 6-node graph with the nodes [0, 1, 4, 5] fully connected to each othe
r. We expect to be able to sample dense subgraphs with high probability.

n_mean = 4
samples = 20 #(Number of samples we want to get)

s = sample.sample(adj, n_mean, samples)

print(s[:5]) #First 5 samples

#The output is like this
#[[0, 0, 0, 0, 0, 0], [1, 1, 1, 0, 1, 1], [0, 1, 1, 1, 0, 1], [0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0,
0]]

#However, the number of clicks in GBS is a random variable and we are not alw
ays guaranteed to have enough clicks in a sample for the resultant subgraph to
be of interest. We can filter out the uninteresting samples using the postselect()
function

min_clicks = 3
max_clicks = 4

s = sample.postselect(s, min_clicks, max_clicks)

print(len(s)) #Gives output 5 (not 20)
s.append([0, 1, 0, 1, 1, 0])

subgraphs = sample.to_subgraphs(s, graph)
#Will convert something like [0, 1, 0, 1, 1, 0] to [1,3,4]
print(subgraphs)
#Prints:[[1, 2, 3, 5], [0, 1, 4, 5], [0, 1, 2], [0, 1, 2, 4], [0, 1, 2, 5], [1, 3, 4]]

plot.graph(graph, subgraphs[0])
```

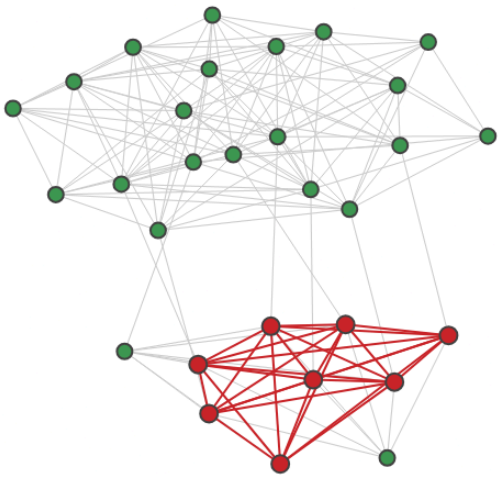


Dense subgraph problem

- The density of a k-node subgraph is equal to the number of its edges divided by the maximum possible number of edges. Identifying the densest graph of a given size, known as the densest-k subgraph problem, is NP-Hard.
- It was found that a defining feature of GBS is that when we encode a graph into a GBS device, it samples dense subgraphs with high probability. This property can be used to find dense subgraphs by sampling from a GBS device and postprocessing the outputs.
- The challenge is to find dense subgraphs of different sizes; it is often useful to identify many high-density subgraphs, not just the densest ones.
- This is the purpose of the `search()` function in the `subgraph` module: to identify collections of dense subgraphs for a range of sizes.
- The output of this function is a dictionary whose keys correspond to subgraph sizes within the specified range. The values in the dictionary are the top subgraphs of that size and their corresponding density.

```
dense = subgraph.search(samples, pl_graph, 8, 16, max_count=3)  # we look at top 3 densest subgraphs
for k in range(8, 17):
    print(dense[k][0])  # print only the densest subgraph of each size

#The output is something like
'''
```

```
(1.0, [21, 22, 24, 25, 26, 27, 28, 29])
(0.9722222222222222, [21, 22, 23, 24, 25, 26, 27, 28, 29])
(0.9333333333333333, [20, 21, 22, 23, 24, 25, 26, 27, 28, 29])
(0.7818181818181819, [17, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29])
(0.696969696969697, [0, 2, 3, 5, 6, 8, 9, 10, 14, 16, 17, 18])
(0.6666666666666666, [2, 3, 6, 8, 9, 10, 12, 13, 14, 15, 16, 17, 18])
(0.6483516483516484, [0, 3, 6, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18])
(0.6285714285714286, [0, 3, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18])
(0.6083333333333333, [0, 3, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18])
'''
```



Smallest subgraph `plot.graph(pl_graph, dense[8][0][1])`

💡 Results Dense Subgraph

Resources used

1. https://strawberryfields.ai/photonics/
2. https://pennylane.ai/qml/demos/gbs
3. https://strawberryfields.ai/photonics/demos/run_gaussian_boson_sampling.html
4. https://strawberryfields.ai/photonics/demos/run_boson_sampling.html
5. https://strawberryfields.ai/photonics/apps/run_tutorial_sample.html
6. https://strawberryfields.ai/photonics/apps/run_tutorial_dense.html

7. https://strawberryfields.ai/photonics/demos/tutorial_X8.html
8. https://pennylane.ai/qml/demos/tutorial_gaussian_transformation
9. https://the-walrus.readthedocs.io/en/latest/index.html
10. https://strawberryfields.ai/photonics/conventions/states.html#squeezed-state
11. https://arxiv.org/pdf/2402.16341 - Sampling problems on a Quantum Computer
12. https://wp.optics.arizona.edu/opti646/wp-content/uploads/sites/55/2022/12/Pizzimenti_Term_Paper.pdf - Gaussian Boson Sampling and it's applications
13. https://arxiv.org/pdf/1801.07488 - A detailed study of Gaussian Boson Sampling