Generalized Hadamard Gates

February 14, 2025

1 Introduction to $\mathbf{LO_v}$-Calculus

The $\mathrm{LO_v}$-calculus is a graphical language designed to represent and reason about linear optical quantum circuits, particularly those that preserve photon polarization. This calculus provides a structured framework for designing, optimizing, and verifying quantum algorithms that leverage photonic states and optical transformations. It is particularly useful for quantum computations that rely on manipulating photons through basic optical components such as beam splitters and phase shifters.

1.1 Key Components

1.1.1 Beam Splitters

Beam splitters are fundamental optical devices in quantum computing, enabling the division of a photonic beam into two separate paths. This capability is crucial for creating quantum superpositions and interference, which are essential for many quantum algorithms.

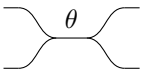


Figure 1: Schematic representation of a beam splitter. The beam splitter divides an incoming photonic state into two possible paths, facilitating quantum superposition and interference.

Mathematically, a beam splitter can be described by a unitary transformation matrix:

$$B(\theta) = \begin{pmatrix} \cos\theta & i\sin\theta \\ i\sin\theta & \cos\theta \end{pmatrix}$$

where θ is a parameter that determines the reflectivity and transmissivity of the beam splitter. This matrix acts on the photonic states, dictating the probability amplitudes for a photon to be transmitted or reflected.

1.1.2 Phase Shifters

Phase shifters are optical elements that introduce a phase change to the photonic state. They are critical for adjusting the relative phases of quantum states, enabling constructive or destructive interference between different photonic paths.

Figure 2: Schematic representation of a phase shifter. The phase shifter alters the phase of an incoming photonic state, which is essential for controlling quantum interference patterns.

A phase shifter is represented by the operation:

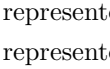
$$P(\phi) = e^{i\phi}$$

where ϕ is the phase shift introduced to the photonic state. This operation is essential for implementing various quantum gates and operations in photonic quantum computing.

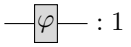
1.1.3 Identity Generator

The identity generator represents the operation where the photonic state remains unchanged. It is essential for composing more complex operations and ensuring the preservation of quantum states through certain parts of a circuit.

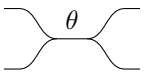
1.2 Syntax

In order to formalise linear optical quantum circuits, we use the formalism of PROPs. A PRO is a strict monoidal category whose monoid of objects is freely generated by a single X: the objects are all of the form $X \otimes ... \otimes X$, and simply denoted by n, the number of occurrences of X. PROs are typically represented graphically as circuits: each copy of X is represented by a wire and morphisms by boxes on wires, so that $\oplus$ is represented vertically and morphism composition "$\circ$" is represented horizontally. For instance, D_1 and D_2 represented as  and can be horizontally composed as $D_2 \circ D_1$, represented by , and vertically composed as $D_1 \oplus D_2$, represented by . A PROP is the symmetric monoidal analogue of PRO, so it is equipped with a swap .

Definition 1 $\mathbf{LO_v}$ *is the PROP of lov-circuits generated by*



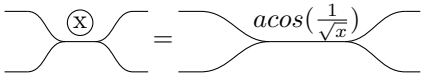
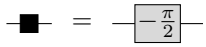
$: 1 \to 1$



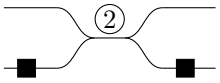
$: 2 \to 2$

where $\theta, \varphi \in \mathbb{R}$. When the parameters θ and φ are omitted we take them to be equal to $-\pi/2$. The tensor of the monoidal structure is denoted with $\oplus$, and the identity, swap and empty circuit (unit of $\oplus$) are denoted as follows: $\text{——} : 1 \to 1$, $: 2 \to 2$, $: 0 \to 0$.

2 Hadamard for small primes

We define  and 

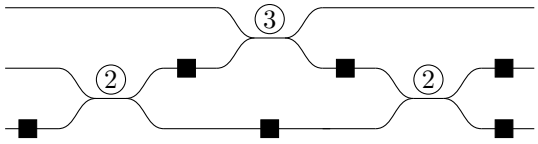
2.1 $d = 2$

$H_2 =$ 

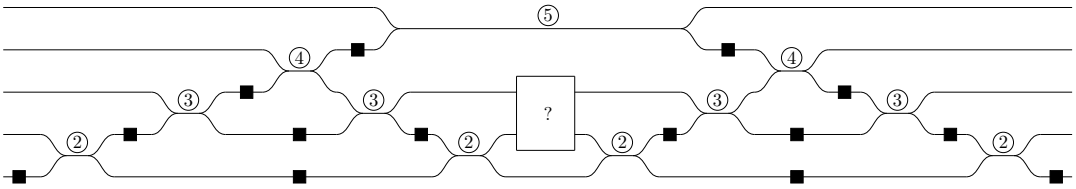
Let's compute it as an example

$$\begin{array}{cc} |0\rangle & |0\rangle \\ |1\rangle & -i|1\rangle \end{array} \qquad \begin{array}{cc} \frac{1}{\sqrt{2}}(|0\rangle + i|1\rangle) & \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \\ \frac{1}{\sqrt{2}}(|0\rangle - i|1\rangle) & \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) \end{array} \tag{1}$$

2.2 $d = 3$

$H_3 =$ 

2.3 $d = 5$

$H_5 =$ 

Where the ? box is a unitary I don't know how to nicely decompose

2.4 $d = 7$ and beyond

What will remain is the pyramidal structure of the circled integers, what is absolutely not clear is the interior of this pyramid.

3 Recursive Construction of Generalized Hadamard Gates

The construction of generalized Hadamard gates for any dimension d can be efficiently achieved through a recursive approach, leveraging oracles for decomposition. Specifically, if you possess an oracle capable of decomposing H_d, this oracle can serve as a foundational element to recursively build H_{d^k} for any integer k. Furthermore, if additional oracles for decomposing $H_{d'}$ are available as well, it becomes possible to construct $H_{d^k \times d'^{k'}}$ for any integers k and k'. Essentially, to construct H_d for a given dimension d, it is sufficient to understand the construction for each prime factor of d, utilizing these constructions as oracles in the recursive process.

We describe here an algorithm to construct it on optical quantum circuits but this could be easily translated on quantum circuits where each wire would be on a prime factor dimension. For example a 12-dimensional Hadamard gate could be implemented using 2 qubit wires and 1 qutrit wire (instead of the 12 wires we use in the optical setting).

3.1 Algorithm Outline

The following pseudocode outlines the recursive construction of generalized Hadamard gates:

Algorithm 1 Recursive Construction of Generalized Hadamard Gates

```
function GEN_HAD_AUX(d)
    if d == 1 then
        return eye(1)
    end if
    b ← smallest_prime_divisor(d)
    root ← exp(2 × π × 1/d)
    H ← eye(d)                                  ▷ Create the Identity matrix
    for i ∈ range(d//b) do
        indices ← [i + k × step for k ∈ range(b)]
        H ← H ∗ apply_had_b_on_rows(indices)
    end for
    for k ∈ range(b) do
        H ← H ∗ apply_phase_on_row_i(indices[k], root ∗ ∗(k × 1))
    end for
    Mat ← simplify(gen_had_aux(d//b, b))
    for pos ∈ range(b) do
        H ← H ∗ insert_mat(Mat, pos, b)
    end for
    return H
end function
```

Algorithm 2 Fully Mixing Permutation

```
function FULLY_MIXING_PERMUTATION(S)
    N ← len(S)
    d ← smallest_prime_divisor(N)
    if N < 2 or d == N then
        return S
    end if
    groups ← [[] for _ ∈ range(d)]
    for index, element ∈ enumerate(S) do
        group_index ← index%d
        groups[group_index].append(element)
    end for
    for i ∈ range(d) do
        if len(groups[i]) > 1 then
            groups[i] ← fully_mixing_permutation(groups[i])
        end if
    end for
    permuted_S ← []
    for group ∈ groups do
        permuted_S.extend(group)
    end for
    return permuted_S
end function
```

Algorithm 3 Decompose Permutation

```
function DECOMPOSE_PERMUTATION(p)
    N ← len(p)
    visited ← [False] * N
    transpositions ← []
    for i ∈ range(N) do
        if not visited[i] then
            cycle ← []
            j ← i
            while not visited[j] do
                visited[j] ← True
                cycle.append(j)
                j ← p[j]
            end while
            if len(cycle) > 1 then
                for k ∈ range(len(cycle) − 1, 0, −1) do
                    transpositions.append((cycle[0], cycle[k]))
                end for
            end if
        end if
    end for
    return reversed(transpositions)
end function
```

Algorithm 4 Apply Permutation

```
function APPLY_PERMUTATION(H@bookinbookID, author = author, title = title, booktitle = booktitle, da
    d ← shape(H)[0]
    S ← list(range(d))
    permuted_S ← fully_mixing_permutation(S)
    for (u, v) ∈ decompose_permutation(permuted_S) do
        H ← H * swap(u, v, d)
    end for
    return H
end function
```

4 Applications and Extensions

4.1 Compatibility with LOv-Calculus and Linear Optical Systems

The proposed method aligns closely with the principles of LOv-calculus, a framework that constrains operations to two-level interactions using basic optical elements like beam splitters and phase shifters. By decomposing higher-dimensional Hadamard gates into recursive operations acting on subsets of levels, this method ensures compatibility with LOv-calculus' restricted operational model. This compatibility facilitates the direct implementation of generalized Hadamard gates in linear optical quantum computing (LOQC), where two-level interactions dominate due to physical constraints.

The recursive construction leverages the modularity of the LOv framework, where each layer of the decomposition corresponds to a combination of beam splitter transformations and phase adjustments. These properties enable efficient implementation in LOQC systems, reducing the need for custom components while preserving the unitarity and symmetry of the gate. Applications in this domain include generating high-dimensional entanglement, implementing quantum Fourier transforms, and preparing resource states for photonic quantum computing protocols.

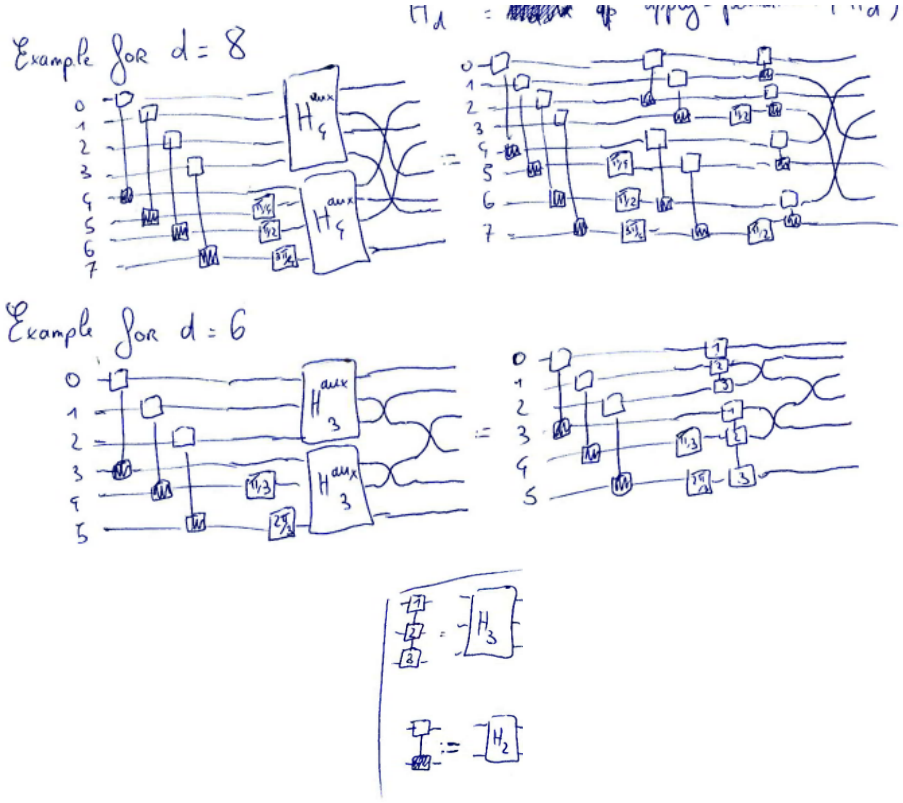


Figure 3: Example diagrams for $d = 8$ and $d = 6$.

5 Appendix : Scanned pdf version

3 Algorithm Description

3.1 Algorithm Outline

The following pseudocode outlines the recursive construction:

Listing 1: Recursive Construction of Generalized Hadamard Gates

```
1
2  def gen_had_aux(d):
3          if d == 1:
4                  return eye(1)
5          b = smallest_prime_divisor(d)
6          root = exp(2 * pi * I / d)
7
8  # Create the Identity matrix
9          H = eye(d)
10
11 # We slice our matrix in b equal parts, and we will apply
12 # the Hadamard for p on each matching position
13         for i in range(d // b):
14                 indices = [i + k * step for k in range(b)]
15                 H *= apply_had_b_on_rows(indices)
16
17 # On the first row, we will have 1, 1, ..., 1 since the Hadamard
18 # applies the superposition.
19 # But for the other rows we will have root^x, root^(x-1), ...
20 # so we will apply a phase such that we cancel the rotation and get
21 # something looking like 1, root, root^2, ...
22                 for k in range(b):
23                         H *= apply_phase_on_row_i(indices[k], root ** (k * i))
24
25 # Now if we call recursively the algorithm on each part, this will
26 # apply a superposition part by part, because right now we only have
27 # an equal superposition of the parts we worked with
28         Mat = simplify(gen_had_aux(d // b, b))
29         for pos in range(b):
30                 H = H * insert_mat(Mat, pos, b)
31         return H
32
33 # Main call
34 b = smallest_prime_factor(d)
35 H = gen_had_aux(d)
36
37 # We need to swap the rows at the end
38 # because it eases the recursivity
39 H = apply_permutation(H)
```

Listing 2: Recursive Construction of Generalized Hadamard Gates

```
def fully_mixing_permutation(S):
        N = len(S); d = smallest_prime_divisor(N)
        # No permutation needed for empty, single-element lists or if N is prime
        if N < 2 or d == N return S

        # Step 1: Divide the sequence into d groups
        groups = [[] for _ in range(d)]
        for index, element in enumerate(S):
                group_index = index % d
                groups[group_index].append(element)

        # Step 2: Recursively apply permutation to each group
        for i in range(d):
                if len(groups[i]) > 1:
                groups[i] = fully_mixing_permutation(groups[i])

        # Step 3: Concatenate the groups in order
        permuted_S = []
        for group in groups:
                permuted_S.extend(group)

        return permuted_S

def decompose_permutation(p):
        N = len(p)
        visited = [False] * N
        transpositions = []

        for i in range(N):
                if not visited[i]:
                        cycle = []
                        j = i
                        while not visited[j]:
                                visited[j] = True
                                cycle.append(j)
                                j = p[j]
                        if len(cycle) > 1:
                                # Decompose the cycle into transpositions
                                # Example: (0,1,2) -> (0,2), (0,1)
                                for k in range(len(cycle)-1, 0, -1):
                                        transpositions.append((cycle[0], cycle[k]))
        return reversed(transpositions)

def apply_permutation(H):
        d = shape(H)[0]
        S = list(range(d))
        permuted_S = fully_mixing_permutation(S)
        for (u,v) in decompose_permutation(permuted_S):
                H *= swap(u, v, d)
        return H
```

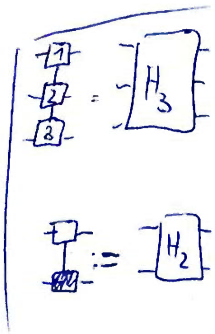
→ decompose into a sequence of pairwise swaps
(012) ⇝ (01)(02)

4 Applications and Extensions

4.1 Compatibility with LOv-Calculus and Linear Optical Systems

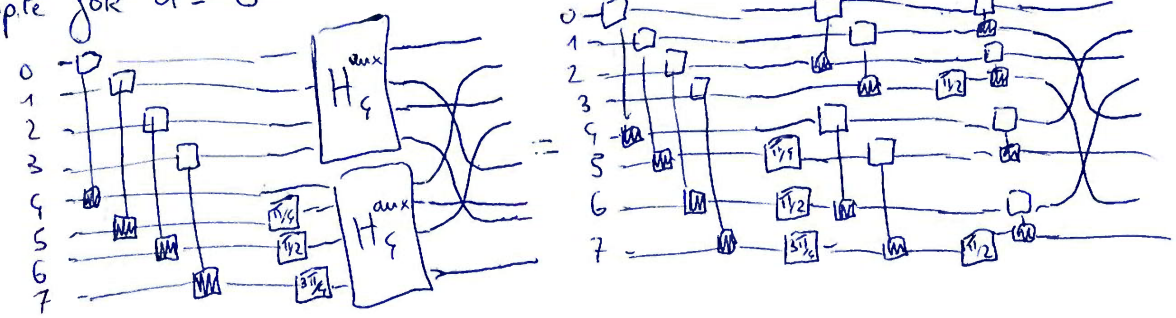
The proposed method aligns closely with the principles of LOv-calculus, a framework that constrains operations to two-level interactions using basic optical elements like beam splitters and phase shifters. By decomposing higher-dimensional Hadamard gates into recursive operations acting on subsets of levels, this method ensures compatibility with LOv-calculus' restricted operational model. This compatibility facilitates the direct implementation of generalized Hadamard gates in linear optical quantum computing (LOQC), where two-level interactions dominate due to physical constraints.

The recursive construction leverages the modularity of the LOv framework, where each layer of the decomposition corresponds to a combination of beam splitter transformations and phase adjustments. These properties enable efficient implementation in LOQC systems, reducing the need for custom components while preserving the unitarity and symmetry of the gate. Applications in this domain include generating high-dimensional entanglement, implementing quantum Fourier transforms, and preparing resource states for photonic quantum computing protocols.



H_d^{aux} := apply-permutation (H_d)

Example for d = 8



Example for d = 6