Karlsruhe Institute of Technology

Internship Report

5th May 2025

Multi-Plane Light Conversion

Nickolay Erin Titov
Matriculation number: **2547697**
01/09-30/11/2024
Fraunhofer IOSB
Supervisor: **Dr. Giacomo Sorelli**

My note: This internal report contains information of preliminary character and is intended for internal use exclusively.

Contents

List of Figures

Multi-Plane Light Conversion

Multi-Plane Light Conversion (MPLC) has emerged as a highly versatile method for manipulating the spatial distribution of optical fields, utilizing repeated phase modulations to precisely reshape light [4]. This approach is essential for applications that leverage specific wave properties, such as spatial coherence, wavefront curvature, and modal structure. These properties are crucial in fields like imaging, optical communication, and quantum optics, where differentiating and exploiting these characteristics allow for enhanced performance and functionality.

For example, spatial decomposition remains relatively underdeveloped compared to spectral or polarization decomposition, which are more established methods for separating light based on frequency or electromagnetic orientation. A common spatial decomposition technique is Fourier decomposition, performed using a lens to separate a beam into its linear momentum components in two dimensions. However, this method is limited because it may not provide the most suitable spatial basis for certain applications, especially those requiring precise transformations between specific orthogonal beam sets [2].

MPLC addresses these challenges by employing a series of phase masks interspaced with free-space propagation. This configuration enables highly accurate transformations between orthogonal beam sets through unitary operations. The ability to achieve such precise spatial control of light is opening new possibilities in applications where control over wave properties is critical. In this report, we present two algorithms for calculating phase masks and offer a comprehensive, step-by-step tutorial for implementing each algorithm in Python.

The Wavefront Matching Algorithm [3]

The Wavefront Matching (WFM) method is a promising approach to designing optical waveguides by generating precise refractive index patterns based on desired optical characteristics. Unlike traditional cut-and-try methods, WFM synthesizes an optimized waveguide structure directly from input and output requirements, leading to efficient designs with reduced computational resources [3].

First, for simplicity, we consider a slab waveguide with a single input and output port. In this setup, the input light propagates along the z-axis, with the waveguide situated on the x-z plane. The y-axis is omitted for simplicity in this 2-D model.The objective is to design a waveguide pattern that accurately guides the input beam to the designated output port [3].

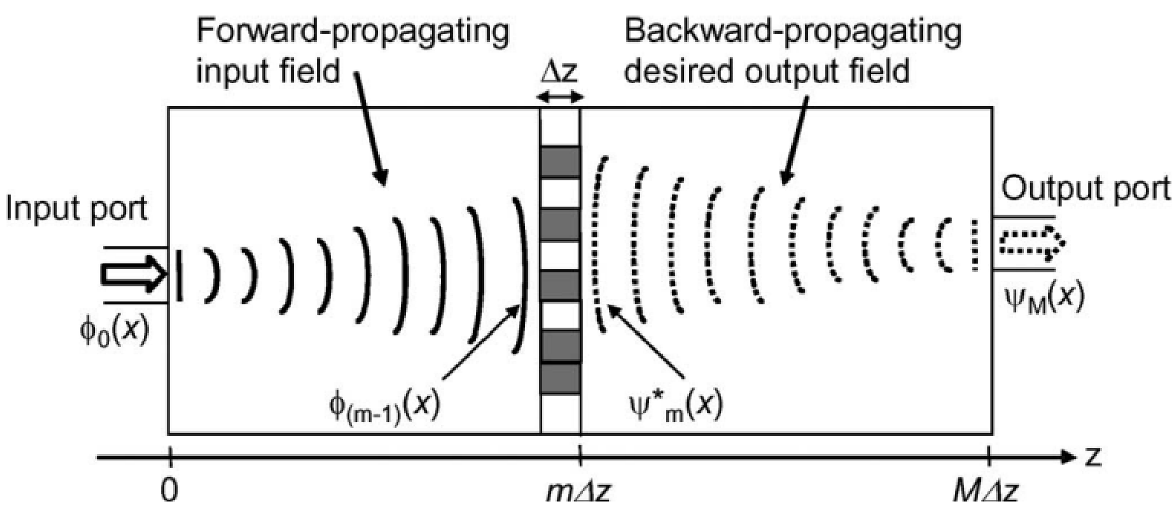


Figure 1 2D Slab waveguide with single input and output port [3]

The input light field is $\phi_0(x)$, and the desired output field is $\psi_M(x)$, where x is the lateral coordinate, and the subscripts 0 and M represent the calculation steps along the z-axis. The optimization region is divided into pixels of size Δz (along z) and Δx (along x), each made of core or cladding material. The coupling efficiency η between the output and target fields is given by the overlap integral:

$$\eta = \left| \int \psi_M^*(x)\phi_M(x)\, dx \right|^2 .$$

where $\phi_M(x)$ is the output field after the optimized region.

By dividing the medium into discrete steps along the propagation direction and simulating the behavior of the optical field as it passes through each step, the output field $\phi_M(x)$ is expressed as:

$$\phi_M(x) = (AB_MA) \cdots (AB_mA) \cdots (AB_1A)\phi_0(x),$$

where:

- A: Free-space propagation operator,
- $B_m(x)$: Phase modulation operator,
- n_{ref}: Reference refractive index,
- $n_m(x)$: Refractive index at $z = m\Delta z$.

Since refractive index, $n(x)$, varies spatially we use reference refractive index, n_{ref}, which provides a stable reference, to avoid recalculating the absolute propagation constant at every point. The calculations are then performed on the relative difference, $n(x)–n_{\text{ref}}$, which represents variations that influence the light's behavior.

Hence the coupling efficiency η becomes:

$$\eta = \left| \int \psi_M^*(x)AB_MA \cdots (AB_mA) \cdots AB_1A\phi_0(x)\, dx \right|^2 .$$

Using the reciprocity,that is transmission of light between two points is invariant if the direction of propagation is reversed, the left side of (AB_mA) is equivalent to the backward-propagating field $\psi_m^*(x)$ simplifing η to:

$$\eta = \left| \int \psi_m^*(x)AB_mA\phi_{m-1}(x)\, dx \right|^2 .$$

To optimize the waveguide design, the refractive index distribution $n_m(x)$ is adjusted to maximize η. Introducing a small change $\delta n_m(x)$, the phase shift operator becomes:

$$B'_m \approx B_m (1 - jk\delta n_m(x)\Delta z) ,$$

assuming $k\delta n_m(x)\Delta z \ll 1$. The updated coupling coefficient η' is:

$$\eta' \approx \eta + 2k\Delta z\sqrt{\eta} \int \Im [\psi_m^*(x)\phi_{m-1}(x)]\, \delta n_m(x)\, dx.$$

This demonstrates that the coupling efficiency can always be improved by setting $\delta n_m(x)$ proportional to $\Im[\psi_m^*(x)\phi_{m-1}(x)]$.In physical terms,to obtain the optimum waveguide pattern we need to match the wavefronts of the forward-propagating input field and the backward-propagating desired output field, by changing local refractive-index distribution.

The basic algorithm described can be extended to handle systems that involve a higher number of optical modes, such as multimode waveguides. In such cases, instead of optimizing the coupling for a single mode, the algorithm adjusts the refractive index distribution to maximize coupling across multiple modes.

Furthermore, instead of physically altering the refractive index distribution in a waveguide, the same principle can be applied using phase masks. Phase masks are computationally designed elements that impose specific phase shifts on the propagating light. These masks can achieve the desired wavefront matching without requiring changes to the material properties of the waveguide.

The design of phase masks involves a numerical computation process. To determine the optimal phase masks for each plane in the optical system, the desired input basis is numerically propagated through the system in the forward direction. Simultaneously, the desired output basis is propagated in the reverse direction. At each step, the overlap between the forward and backward fields is calculated, and the phase mask for the current plane is adjusted to maximize this overlap.

In other words, at each iteration, the phase mask is updated to align with the phase of the superposition of the overlaps between each pair of input mode (ψ^i) and output mode (ϕ^i). Meaning, the phase mask reflects the average phase error between the forward-propagating modes (ψ^i and the backward-propagating modes (ϕ^i). Mathematically, the phase ϕ at a given plane during an iteration is expressed as:

$$\theta_m = \arg\left(\sum_{i=1}^{N}(\psi^i_m(x))^*\phi^i_{m-1}(x)\right),$$

where N is the total number of modes. The phase masks are iteratively updated until convergence is achieved [2].

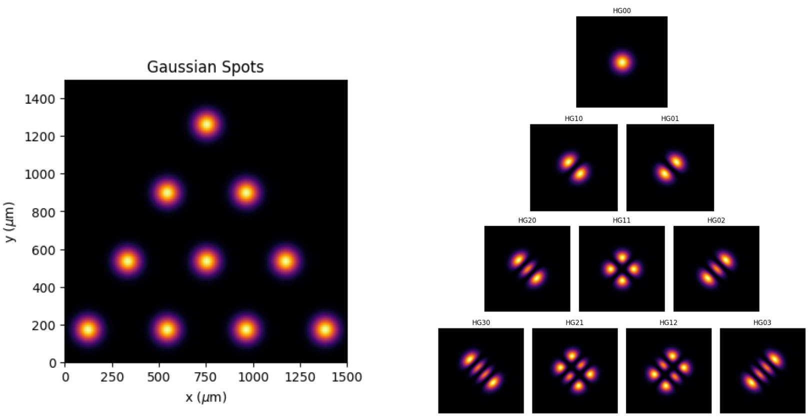
In our setup we choose spatially separated Gaussian beam in triangular shape array as a input modes (ϕ) and spatially co-located Hermite-Gaussian(HG) beam as output modes (ϕ).The reason of our output mode choice stems from the fact that HG modes maintain their characteristic intensity and phase distribution as they propagate through free space. which ensures that the decomposition of a wavefront into HG modes remains consistent over distance, simplifying analysis and correction.For demonstration purposes we choose 10 modes and 10 planes but both these parameters can be changed and tested in simulation easily.

We can initialize the simulation by defining parameters of simulation constrains, input and output basis.

```python
import Propagation as pg
import ModeGeneration as mg
import BasicFunctions as bf
import numpy as np
import matplotlib.pyplot as plt

# Simulation parameters
N = 128  # pixel number
L = 0.0015 # grid size [m]
w = 90e-6  # width of input [m]
w_prime = 200e-6  # width of output [m]
delta = L / N  # grid spacing [m]
wvl = 633e-9  # optical wavelength [m]
z = 0.015 # distance between phase masks [m]
angle= 0.05

maskOffset = np.sqrt(1e-3 / (N * N * 10)) * 0.2
num_planes =10
num_modes = 10

# Space grid
x = np.linspace(-N / 2, N / 2 - 1, N) * delta
y = np.linspace(-N / 2, N / 2 - 1, N) * delta * 1/np.cos(angle)
x, y = np.meshgrid(x, y)

TH,R = bf.cart2pol(x,y)
x0,y0 = bf.pol2cart(TH-np.pi/4,R)

#triangular shape array beams parameter
d = 2e-4 *2.1# Distance between the beams [m]
central_shift = 2e-4*2.6 #location of top beam [m]

def create_gaussian_beams_array(x, y, w, d, central_shift, num_beams):
    beams = []
    beam_count = 0
    layer = 0

    while beam_count < num_beams:
        # Calculate the starting position for the current layer
        start_x = x - layer * d / 2
        start_y = y + central_shift - layer * np.sqrt(3) * d / 2

        # Generate beams in the current layer
        for i in range(layer + 1):
            if beam_count >= num_beams:
                break
            beam_x = start_x + i * d
```

```
            beam_y = start_y
            beams.append(mg.hg_mode(beam_x, beam_y, w, 0, 0))
            beam_count += 1

        # Move to the next layer
        layer += 1

    return beams[:num_beams]


all_gaussian_beams = sum(create_gaussian_beams_array(x, y, w, d, central_shift, num_modes))



# Generate HG modes
def generate_hg_modes_array(x, y, w_prime, mode_count):
    modes = []
    index = 0
    for m in range(mode_count + 1):
        for n in range(m + 1):
            if index >= mode_count:
                return modes
            mode = mg.hg_mode(x, y, w_prime, m - n, n)
            modes.append(mode)
            index += 1
    return modes



input_modes = create_gaussian_beams_array(x, y, w, d, central_shift, num_modes)
output_modes = generate_hg_modes_array(x0, y0, w_prime,num_modes)

#output_modes = mg.sort_and_limit_lg_modes(mg.generate_lg_modes_array(x0, y0, w_prime,3,3),
    num_modes)


# Transfer function of free-space propagation
H0 = bf.transfer_function_of_free_space(x, y, z, wvl)

# Phase masks
mask = np.exp(1j * np.angle(np.ones((N, N))))
phase_masks = []
for i in range(num_planes):
    phase_masks.append(mask)
```

As a result we created input beam basis that is Gaussian beams in triangular shape and output beam basis that is spatially co-located HG modes.



(a) Gaussian beams (b) Spatially co-located HG modes

Figure 2 Gaussian beams and spatially co-located HG modes.

After defining all the necessary parameter for simulation we can use WFM algorithm to compute phase mask. We also includes fidelity of the algorithm which represents the overlap (or coupling efficiency) between the i-th output mode and the j-th desired mode. Mathematically, it is defined as:

$$F_{ij} = \left| \int_{\text{all space}} \psi_j^*(x,y)\phi_i(x,y)\, dx\, dy \right|^2$$

where $\phi_i(x,y)$ is the optical field of the i-th output mode at the output and $\psi_j(x,y)$ is the desired j-th mode field. Off-diagonal elements in the coupling matrix F_{ij} indicate coupling between unintended modes, a phenomenon called mode crosstalk. A well-optimized system minimizes these off-diagonal terms, ensuring that each desired mode is produced without contamination by others.The coupling matrix serves as a feedback mechanism for iterative algorithms like the Wavefront Matching Method (WFM). For this reason we also plot diagonal elements of coupling matrix (Fidelity) versus number of iteration to check the performance of WFM algorithm.

```
opt_list=[]
num_iterations = 50

for iteration in range(num_iterations):

    output_modes = generate_hg_modes(x0, y0, w_prime,num_modes) #backward
    input_modes = create_gaussian_beams(x, y, w, d, central_shift, num_modes)  #forward

    for distance_z in range(len(phase_masks), 0, -1):
        # Reset the output modes for each distance_z
        output_modes = generate_hg_modes(x0, y0, w_prime,num_modes) #backward

        # Reverse loop through the phase masks based on the current distance_z
        for i in range(len(phase_masks) - 1, len(phase_masks) - distance_z , -1):
            # First, propagate the output modes
            output_modes = [propagate(output_mode, np.conj(H0)) for output_mode in output_modes]
            # Apply the current phase mask and propagate again
            output_modes = [output_mode * np.exp(1j*np.angle(phase_masks[i])) for output_mode in output_modes]

        # Final propagation for output_modes at the current distance_z
        output_modes_at_mask = [propagate(output_mode, np.conj(H0))  for output_mode in output_modes]
        pwr_output_modes_at_mask = [np.sum(np.abs(np.conj(modes))**2)  for modes in output_modes_at_mask]

        # Propagate the input modes
        input_modes_at_mask = [propagate(input_mode, H0)  for input_mode in input_modes]
        pwr_input_modes_at_mask = [np.sum(np.abs(modes)**2)  for modes in input_modes_at_mask]


        o_v = []
        dphi =[]
        MSK=0
        for i in range(len(input_modes)):

            o_v.append((input_modes_at_mask[i] * np.conj(output_modes_at_mask[i]))/np.sqrt(pwr_output_modes_at_mask[i]*pwr_input_modes_at_mask[i]))
            dphi.append(np.sum(o_v[i]*np.conj(phase_masks[len(phase_masks)-distance_z])))
        for i in range(len(input_modes)):
            MSK = MSK + o_v[i]*np.exp(-1j*np.angle(dphi[i]))

        phase_masks[len(phase_masks)-distance_z]  = MSK + maskOffset

        # Multiply the input beams by the current updated phase mask
        input_modes = [inputs * np.exp(-1j*np.angle(phase_masks[len(phase_masks)-distance_z])) for inputs in input_modes_at_mask]

    output_modes = generate_hg_modes(x0, y0, w_prime,num_modes)
    input_modes = create_gaussian_beams(x, y, w, d, central_shift, num_modes)  #forward




    for distance_z in range(len(phase_masks), 0, -1):
        # Reset the input modes for each distance_z
        input_modes = create_gaussian_beams(x, y, w, d, central_shift, num_modes)  #forward

        # Loop forward through the phase masks based on the current distance_z
        for i in range(0, distance_z - 1, 1):
```

```
            # First, propagate the input modes
            input_modes = [propagate(input_mode,H0)  for input_mode in input_modes]
            # Apply the current phase mask and propagate again
            input_modes = [input_mode * np.exp(-1j*np.angle(phase_masks[i])) for input_mode in input_modes]

        # Final propagation for input_modes at the current distance_z
        input_modes_at_mask = [propagate(input_mode, H0)  for input_mode in input_modes]
        pwr_input_modes_at_mask = [np.sum(np.abs(modes)**2)  for modes in input_modes_at_mask]

        # Propagate the output modes
        output_modes_at_mask = [propagate(output_mode, np.conj(H0))  for output_mode in output_modes]
        pwr_output_modes_at_mask = [np.sum(np.abs(np.conj(modes))**2)  for modes in output_modes_at_mask]

        o_v = []
        dphi =[]
        MSK=0
        for i in range(len(input_modes)):

            o_v.append((input_modes_at_mask[i] * np.conj(output_modes_at_mask[i]))/np.sqrt(pwr_output_modes_at_mask[i]*pwr_input_modes_at_mask[i]))
            dphi.append(np.sum(o_v[i]*np.conj(phase_masks[distance_z - 1])))

        for i in range(len(input_modes)):
            MSK = MSK + o_v[i]*np.exp(-1j*np.angle(dphi[i]))

        phase_masks[distance_z - 1] = MSK + maskOffset

        # Multiply the output beams by the current updated phase mask
        output_modes = [outputs * np.exp(1j*np.angle(phase_masks[distance_z - 1])) for outputs in output_modes_at_mask]

    output_modes = generate_hg_modes(x0, y0, w_prime,num_modes)
    input_modes = create_gaussian_beams(x, y, w, d, central_shift, num_modes)  #forward

    for i in range(len(phase_masks),0, -1):
        output_modes = [ propagate(output_mode,np.conj(H0)) for output_mode in output_modes]
        output_modes = [output_mode * np.exp(1j*np.angle(phase_masks[i - 1])) for output_mode in output_modes]

        pwr_output_modes = [np.sum(np.abs(modes)**2)  for modes in output_modes]
        for i in range(len(output_modes)):
            output_modes[i] = output_modes[i]/np.sqrt(pwr_output_modes[i])

    final_output = [ propagate(output_mode,np.conj(H0)) for output_mode in output_modes]

    tr_matrix=[]

    for i in range(len(output_modes)):
        tr_matrix.append(np.sum(np.abs(input_modes[i]*np.conj(final_output[i]))))
    tr_matrix = np.sum(tr_matrix)/len(output_modes)
    opt_list.append(tr_matrix)
```

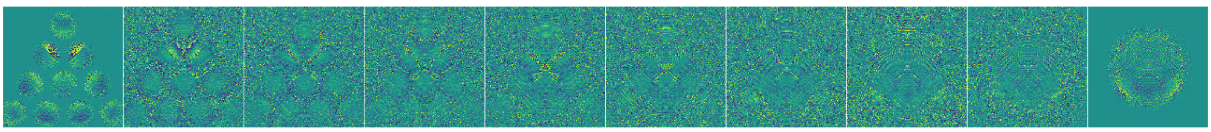


Figure 3 Phase masks generated by Wavefront Matching Algorithm

After running algorithm we were able to get phase masks as shown in Figure 3. Then, we propagate spatially co-located HG modes(2) through these phase mask and got following results:

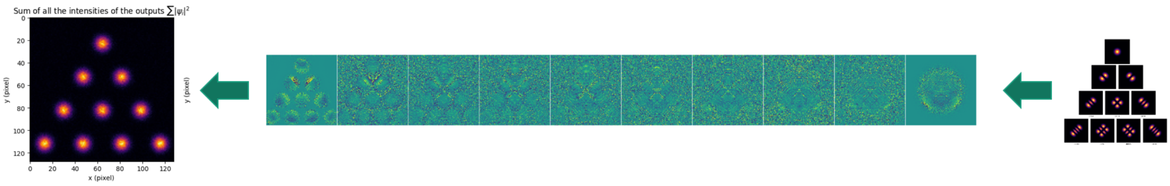
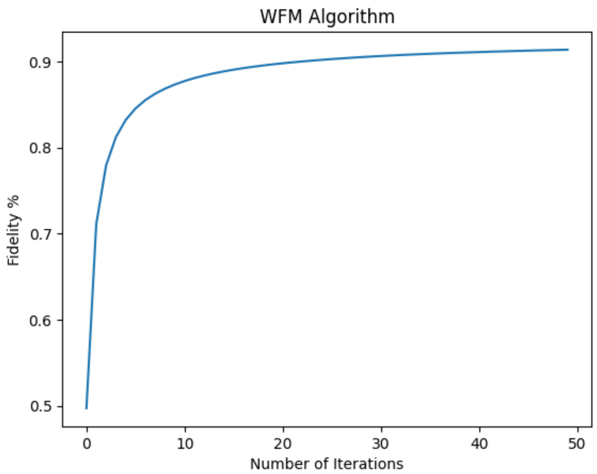
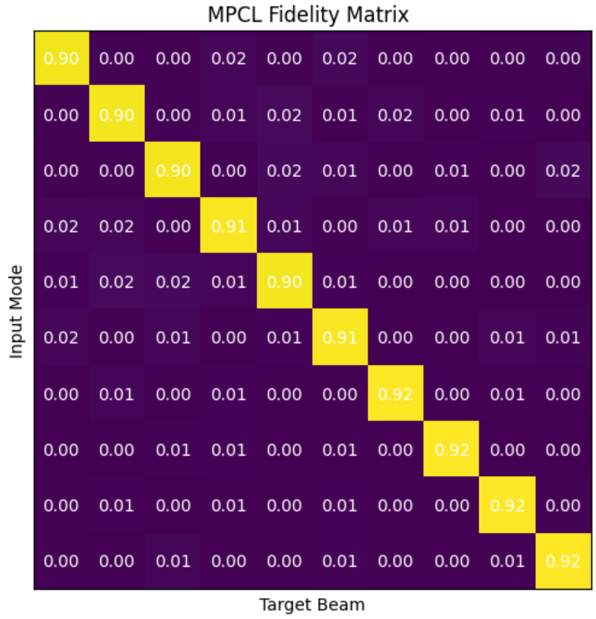


Figure 4 Spatially co-located HG modes propagated through computed phase masks



(a) WFM Performance



(b) Fidelity Matrix

As can be seen on fidelity matrix we achieved %90 fidelity and maximum % 2 cross-talk by using 10 modes, 10 phase masks with 50 iteration.

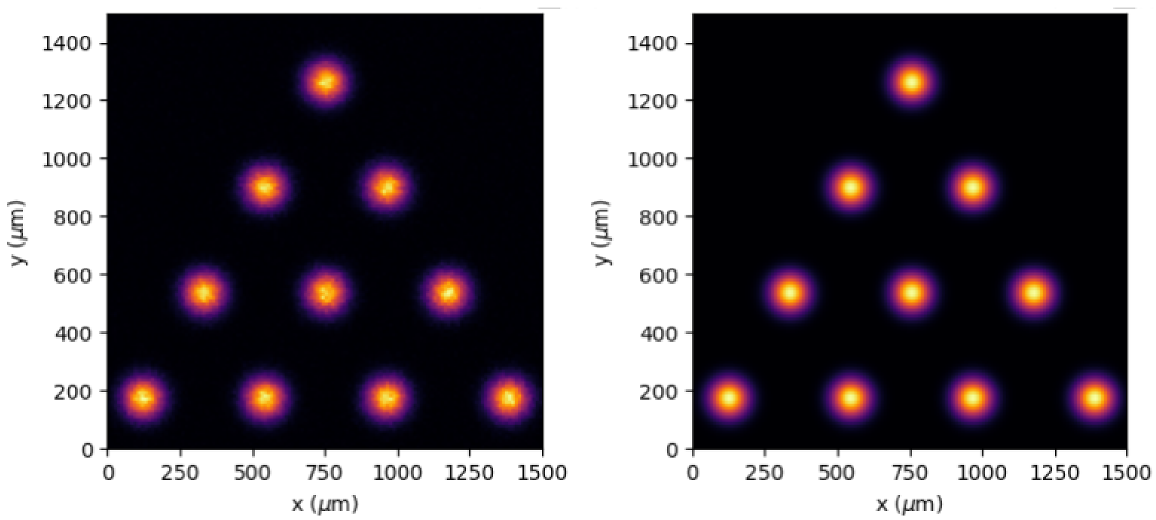
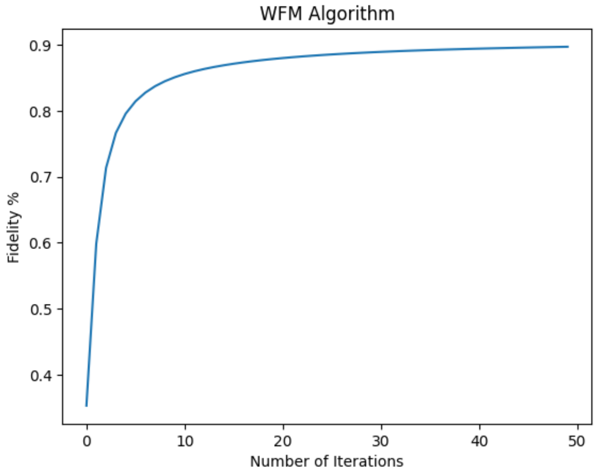
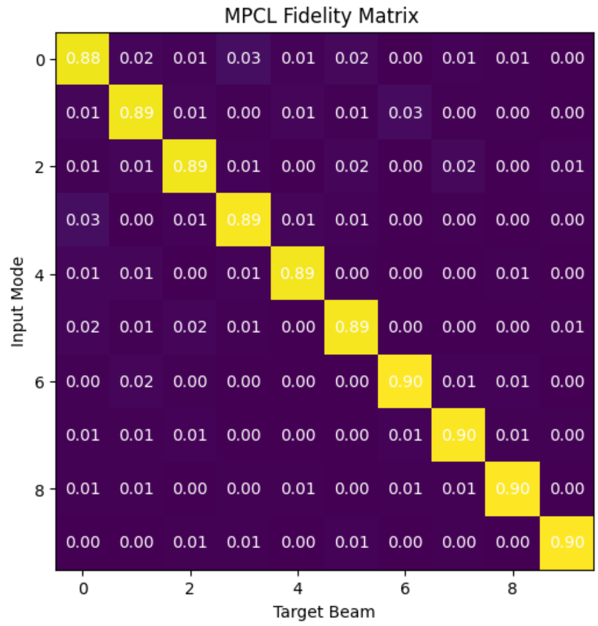


Figure 5 Sum of all the intensities of (on the left) output mode and (on the right) desired output using HG basis

Similarly, the simulation can be compiled with different orthogonal modes. With Laguerre-Gaussian (LG) basis using the same configuration 10 modes, 10 phase mask and 50 iteration we achieved %90 fidelity and maximum % 3 cross-talk.



(a) WFM Performance



(b) Fidelity Matrix

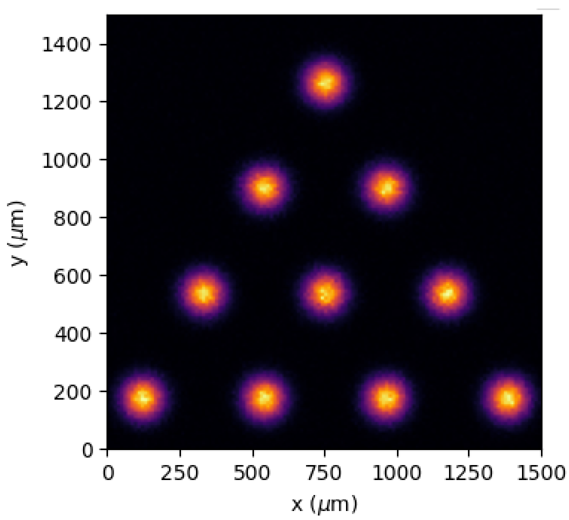


Figure 6 Sum of all the intensities of (on the left) output mode and (on the right) desired output using LG basis

Gradient Ascent-Based Algorithm [1]

The gradient ascent-based (GAB) optimization algorithm is a robust method used to design the phase profiles of phase masks. This algorithm iteratively updates the phase values of the masks to maximize a predefined objective function. By doing so, it ensures high-fidelity mode sorting while accounting for cross-talk and efficiency [1]. The objective function used in this optimization balances fidelity, cross-talk suppression, and efficiency. It is expressed as:

$$F_T = \sum_{i=1}^{N} F_i,$$

where F_i is the contribution from the i-th mode and is defined as:

$$F_i = \alpha \left| \psi_i^* \cdot \phi_i \right|^2 - \beta \Re \left[\psi_i^* \cdot \psi_i^{\mathrm{cr}} \right] + \gamma \Re \left[\psi_i^* \cdot \psi_i^{\mathrm{bk}} \right].$$

Here:

- ψ_i: Forward-propagated field for the i-th mode.
- ϕ_i: Target output mode.
- $\psi_i^{\text{cr}} = \psi_i \odot \phi_{\text{cr}}$: Cross-talk contribution for the i-th mode, derived using the mask ϕ_{cr} that defines regions corresponding to incorrect output channels.
- $\psi_i^{\text{bk}} = \psi_i \odot \phi_{\text{bk}}$: Background light contribution, derived using the mask ϕ_{bk} for regions outside all output channels.
- α, β, γ: Weights for fidelity, cross-talk suppression, and efficiency, respectively.

The term ψ_{bk} represents the portion of light directed into a designated background region outside the defined output channels and is used in the efficiency term of the objective function. It is defined as $\psi_{\text{bk}} = \psi_n \odot \phi_{\text{bk}}$, where $\odot$ denotes the Hadamard product with a mask ϕ_{bk} that defines the background region, having values of 1 in the background and 0 within the output channels. Increasing ψ_{bk} allows deliberate scattering of light to non-output regions, which can improve fidelity or reduce cross-talk. On the other hand, ψ_{cr} represents the portion of light contributing to cross-talk between output channels and is part of the cross-talk penalty term in the objective function. It is defined as $\psi_{\text{cr}} = \psi_n \odot \phi_{\text{cr}}$, where the mask ϕ_{cr} has values of 1 in regions corresponding to incorrect output channels and 0 in the correct channel and other areas. Minimizing ψ_{cr} reduces light leakage into unintended channels.

The gradient of F_i with respect to ψ_i is calculated as:

$$\frac{\partial F_i}{\partial \psi_i} = \alpha \left(\psi_i^\dagger \cdot \phi_i\right) \phi_i^\dagger - \frac{\beta}{2}\psi_i^{\text{cr}\dagger} + \frac{\gamma}{2}\psi_i^{\text{bk}\dagger}.$$

where F_i depends on the complex output vector ψ_n and its conjugate transpose ψ_n^*. A small change in the m-th phase mask, $P_m \to P_m + \delta P_m$, induces a change in the transfer matrix:

$$\delta S = (B_M A \cdots B_{m+1} A) \cdot \delta B_m \cdot (A B_{m-1} \cdots A B_1) = S_> \cdot \delta B_m \cdot S_<.$$

where A is a free-space propagation matrix.This change modifies the output fields as:

$$\delta\psi_i = \delta S \cdot \chi_i,$$

where χ_i is the input field. Consequently, the objective function changes as:

$$\delta F_T = \sum_{i=1}^{N} \left[\frac{\partial F}{\partial \psi_i} \cdot \delta\psi_i + \delta\psi_i^* \cdot \frac{\partial F}{\partial \psi_i^*}\right].$$

Using the property that $a + a^* = 2\Re(a)$ for a complex number a, this simplifies to:

$$\delta F_T = 2\Re\left[\sum_{i=1}^{N} \frac{\partial F}{\partial \psi_i} \cdot \delta\psi_i\right].$$

Substituting $\delta\psi_i$ into this expression, we get:

$$\delta F_T = 2\Re\left[\sum_{i=1}^{N} \left(\frac{\partial F}{\partial \psi_i} \cdot S_>\right) \cdot \delta P_m \cdot (S_< \cdot \chi_i)\right].$$

Writing this in terms of pixel-by-pixel contributions, the change becomes:

$$\delta F_T = 2\sum_{p=1}^{P} \Re\left[(\delta P_m)_{p,p} \sum_{i=1}^{N} (v_{i,<})_p (v_{i,>})_p\right],$$

where $v_{i,<} = S_< \cdot \chi_i$ and $v_{i,>} = \frac{\partial F}{\partial \psi_i} \cdot S_>$. Using $P_m = \text{diag}[e^{i\theta_{m,p}}]$, the change in the phase of each pixel is given by:

$$\delta P_m = \text{diag}[i\delta\theta_{m,p} e^{i\theta_{m,p}}],$$

and we have:

$$\delta F_T = -2\sum_{p=1}^{P} \delta\theta_{m,p} \Im\left[e^{i\theta_{m,p}} \sum_{i=1}^{N} a_i (v_{i,<})_p (v_{i,>})_p\right].$$

To ensure $\delta F_T > 0$, we choose:

$$\text{sign}(\delta\theta_{m,p}) = -\text{sign}\left\{\Im\left[e^{i\theta_{m,p}}\sum_{i=1}^{N} a_i (v_{i,<})_p (v_{i,>})_p\right]\right\}.$$

An incremental adjustment to the phase plane, δP_m, is made by applying a small phase change of fixed magnitude, $\delta\theta$, to each pixel. The sign of $\delta\theta$ is determined individually for each pixel to ensure that all contributions are positive, thereby increasing the objective function.

We can use this algorithm with previous configuration. The original algorithm can be found in [1]. The following code is modified version so that it matches with previous configuration.

```
n_of_modes = 10 # number of modes to sort
Planes = 10 # number of phase masks to design
iterations = 50 # number of iterations to run for (number of times each phase mask gets
    updated during the design process)

# Alpha, beta and gamma factors to adjust the objective function
alpha = 1.0
beta = 2.0
gamma = 0.0

first_n_iterations = 10 # use a bigger step size for the first 10 iterations
delta_theta_1 = 2*math.pi/255 # usual step size
delta_theta_0 = delta_theta_1*10 # bigger step size

Nx = 128 # resolution of a field of view in pixels (x)
Ny = 128 # resolution of a field of view in pixels (y)
pixelSize = 11.72e-6 # pixel pitch, in m (chosen to match a pixel pitch of the Hamamatsu
    X13138-01 SLM)
wavelength = 633.e-9 # wavelength, in m

reprW = Nx*pixelSize # physical width of a field of view, in m
reprH = Ny*pixelSize # physical height of a field of view, in m

d_in = 0e-3 # free-space propagation distance from the plane where all the inputs are
    generated to the first phase mask of the MPLC, in m (used to simulate inputs being
    slightly defocused)
d = 0.015  # free-space propagation distance between each pair of phase masks, in m

calc_perf_every_it = 10 # calculate and print out sorter's performance every 10 iterations
crs_delta = 0.0001*calc_perf_every_it # stop the algorithm before the target number of
    iterations has been reached if the cross-talk hasn't improved by this amount since the
    last time it has been calculated

equalize_efficiency = 1 # 1 - on, 0 - off. sometimes the algorithm gets pulled towards sorting
     some of the modes from a set more efficiently than the other ones. use
    equalize_efficiency = 1 to compensate this.
plot_eff_distribution = 0 # plot the efficiency distribution every time sorter's performance
    gets calculated during the design process

smoothing_switch = 0 # 1 - on, 0 - off. mask the regions of the phase masks where there is
    almost no incedent light.
# as usually these regions are next to the edge of the field of view, this also allows to
    prevent light being scattered to the outside of the field of view and coming back from the
     other side
OffsetMultiplier = 1 # tweak this to adjust the strength of 'masking' the phase masks
maskOffset = OffsetMultiplier*np.sqrt(1e-3/(Nx*Ny*n_of_modes))


Speckle_basis = np.stack(generate_hg_modes(x0, y0, w_prime,n_of_modes), axis=0)


Gaussian_basis = np.stack(create_gaussian_beams(x, y, w, d, central_shift, n_of_modes), axis
    =0)


Gaussian_Masks = np.stack([phi_1, phi_2, phi_3,phi_4, phi_5, phi_6,phi_7, phi_8, phi_9,phi_10
    ], axis=0)

# take subsets of the basis arrays in case the number of modes sorted is less than 55.
Speckle_basis = Speckle_basis[0:n_of_modes,:,:]
Gaussian_basis = Gaussian_basis[0:n_of_modes,:,:]
Gaussian_Masks = Gaussian_Masks[0:n_of_modes,:,:]
```

```

# convert arrays from numpy ndarrays to torch tensors
Speckle_basis_torch = torch.from_numpy(Speckle_basis)
Gaussian_basis_torch = torch.from_numpy(Gaussian_basis)
Gaussian_Masks_torch = torch.from_numpy(Gaussian_Masks)

# if the chosen resolution of the field of view Nx, Ny larger than the resolution input/output
     bases were generated in, pad bases with zeros to match the resolution.
if Nx or Ny >  128:
    Speckle_basis_torch = nn.functional.pad(Speckle_basis_torch, (int((Nx-128)/2),Nx-128-int((
    Nx-128)/2),int((Ny-128)/2),Ny-128-int((Ny-128)/2)), mode='constant', value = 0)
    Gaussian_basis_torch = nn.functional.pad(Gaussian_basis_torch, (int((Nx-128)/2),Nx-128-int
    ((Nx-128)/2),int((Ny-128)/2),Ny-128-int((Ny-128)/2)), mode='constant', value = 0)
    Gaussian_Masks_torch = nn.functional.pad(Gaussian_Masks_torch, (int((Nx-128)/2),Nx-128-int
    ((Nx-128)/2),int((Ny-128)/2),Ny-128-int((Ny-128)/2)), mode='constant', value = 0)

# calculate phi_bk - the binary mask outlining the backgroud region where there are no target
    output channels
phi_bk = 1 - torch.sum(Gaussian_Masks_torch, axis = 0)

# calculate phi_cr - the binary mask outlining all the wrong output channels for each mode in
phi_cr = torch.zeros((n_of_modes, Ny, Nx), dtype = torch.double)
for i in range(n_of_modes):
    phi_cr[i,:,:] = torch.sum(Gaussian_Masks_torch, axis = 0) - Gaussian_Masks_torch[i,:,:]

phi = Gaussian_basis_torch


# calculate wavenumber and its x, y, z components, create XY coordinate grids
k = (2 * np.pi) / wavelength

nx = pixelSize*np.linspace(-(Nx-1)/2, (Nx-1)/2, num=Nx)
ny= pixelSize*np.linspace(-(Ny-1)/2, (Ny-1)/2, num=Ny)
X,Y = np.meshgrid(nx,ny)
X_torch = torch.from_numpy(X)
Y_torch = torch.from_numpy(Y)

nx = np.linspace(-(Nx-1)/2, (Nx-1)/2, num=Nx)
ny = np.linspace(-(Ny-1)/2, (Ny-1)/2, num=Ny)
kx, ky = np.meshgrid(2*np.pi*nx/(Nx*pixelSize),2*np.pi*ny/(Ny*pixelSize))
kz = np.sqrt(k**2 - (kx**2 + ky**2))
kz = kz.astype(np.cdouble)
kz_torch = torch.from_numpy(kz)

# Transfer function of free-space propagation
H0 = transfer_function_of_free_space(x, y, z, wvl)


Masks = torch.zeros((Planes,Ny,Nx)) # use zero phases as starting guesses for the phase masks
Masks_complex = torch.exp(1j*Masks) # complex representation of the phase masks with amplitude
     = 1 everywhere

# create placeholder arrays to store every input and every output field in each plane
Modes_in = torch.zeros((Planes, n_of_modes, Ny, Nx), dtype = torch.cdouble)
Modes_out = torch.zeros((Planes, n_of_modes, Ny, Nx), dtype = torch.cdouble)

overlap = torch.zeros((n_of_modes), dtype = torch.cdouble)
eff_distribution = torch.ones((n_of_modes), dtype = torch.double)
dFdpsi = torch.zeros((Planes, n_of_modes, Ny, Nx), dtype = torch.cdouble)
crs_array_convergence = torch.zeros((iterations//calc_perf_every_it), dtype = torch.double)
conv_count = 0

# store input modes directly in front of the 1st phase plane of the MPLC
Modes_in[0, :, :, :] = propagate_HK(Speckle_basis_torch, H0)
# store output modes straight after the last phase plane of the MPLC. We simulate a system
    where the last plane of the MPLC and the output plane are separated by a Fourier transform
     lens
Modes_out[Planes-1, :, :, :] = propagate_HK(Gaussian_basis_torch, H0)

# iterate
for i in range(1, iterations+1):

    # change the step size depending on the current iteration number
    if i < first_n_iterations:
        delta_theta = delta_theta_0
```

```
else:
    delta_theta = delta_theta_1

# update all the phase masks on this iteration in an ascending order
for mask_ind in range(Planes):

    # propagate_HK input modes forward up to the last plane
    modes = torch.zeros((n_of_modes, Ny, Nx), dtype = torch.cdouble)
    for pl in range(Planes-1):
        modes = Modes_in[pl, :, :, :]*Masks_complex[pl, :, :] # add a phase masks to all the incoming modes at once
        modes = propagate_HK(modes, H0) # propagate_HK all the modes at once distance d through free space
        Modes_in[pl+1, :, :, :] = modes
    modes_forw_last_plane = Modes_in[Planes-1, :, :, :]*Masks_complex[Planes-1, :, :] # add the last phase mask
    psi = modes_forw_last_plane # go from the last plane of the MPLC to the output plane in a Fourier plane of it

    # calculate differentials of the objective functions dFdpsi for every input-output pair of modes
    for j in range(n_of_modes):
        overlap = torch.sum(torch.squeeze(psi[j,:,:])*torch.conj(torch.squeeze(phi[j,:,:])))
        a = (phi[j, :, :])*overlap
        psi_cr = (torch.squeeze(psi[j,:,:]))*torch.squeeze(phi_cr[j,:,:])
        psi_bk = (torch.squeeze(psi[j,:,:]))*phi_bk
        dFdpsi[Planes-1,j,:,:] = - alpha*a + (beta*psi_cr - gamma*psi_bk)*0.5 # store dF/dpsi in the output plane before propagating it back to the phase plane of interest

    dFdpsi[Planes-1,:,:,:] = dFdpsi[Planes-1,:,:,:] # get from the output plane to the last plane of the MPLC

    # propagate_HK dF/dpsi back through the MPLC up to the phase plane of interest and store it in every intermediate plane
    for pl in range(Planes-1, mask_ind, -1):
        dFdpsi_prop = dFdpsi[pl,:,:,:]*torch.conj(Masks_complex[pl, :, :]) # subtract phase mask from the fields propagate_HKd backwards
        dFdpsi_prop = propagate_HK(dFdpsi_prop, np.conj(H0)) # propagate_HK fields distance -d backwards
        dFdpsi[pl-1, :, :, :] = dFdpsi_prop

        # do the same not only to the dF/dpsi arrays, but also to the actual output modes phi as this will be needed to apply smoothing
        phi_prop = Modes_out[pl, :, :, :]*torch.conj(Masks_complex[pl, :, :])
        phi_prop = propagate_HK(phi_prop, np.conj(H0))
        Modes_out[pl-1, :, :, :] = phi_prop

    # if equalize_efficiency is on, make a sum in (1) a weighted sum, where the weights are 1/(relative_efficiency_i) for each particular mode
    if equalize_efficiency == 1:
        weighted_overlaps = torch.zeros((Nx,Ny), dtype = torch.cdouble)
        for mode in range(n_of_modes):
            weighted_overlaps = weighted_overlaps + (1/eff_distribution[mode])*torch.squeeze(Modes_in[mask_ind, mode, :, :])*torch.conj(torch.squeeze(dFdpsi[mask_ind, mode, :, :]))
        delta_P = delta_theta*torch.sign(torch.imag(Masks_complex[mask_ind]*weighted_overlaps))
    else:
        delta_P = delta_theta*torch.sign(torch.imag(Masks_complex[mask_ind]*torch.sum(torch.squeeze(Modes_in[mask_ind, :, :, :])*torch.conj(torch.squeeze(dFdpsi[mask_ind, :, :, :])), axis = 0)))

    #  if smoothing_switch is on, mask the regions of the phase masks where there is almost no incedent light, based on the overlap of input and output modes at this plane
    if smoothing_switch == 1:
        ovrlp_in_out = torch.abs(torch.sum(torch.squeeze(Modes_in[mask_ind, :, :, :]*torch.conj(Modes_out[mask_ind, :, :, :])), axis = 0))
        # add phase delta_P to a current guess of the certain phase mask.
        # for a complex representation, set an overlap of inputs/outputs as an amplitude for a resulting phase
        mask_cmplx = ovrlp_in_out*torch.exp(1j*(Masks[mask_ind, :, :] + delta_P))
        mask_cmplx = mask_cmplx + maskOffset # add a real-valued offset to mask the regions with almost no incident light
```

```
                Masks[mask_ind, :, :] = torch.angle(mask_cmplx) # take a phase of a result and store it as a current guess of a certain phase mask
        #  if smoothing_switch is off, just add phase delta_P to a current guess of the certain phase mask
        else:
            Masks[mask_ind, :, :] = Masks[mask_ind, :, :] + delta_P

        # store the resulting current guess of the phase mask as a complex array, with amplitude = 1 everywhere
        Masks_complex[mask_ind, :, :] = torch.exp(1j*torch.squeeze(Masks[mask_ind, :, :]))


    # calculate and print out sorter's performance after every iteration (or every K iterations to save time)
    if i % calc_perf_every_it == 0:
        # propagate_HK all the input modes through the MPLC system after all of the phase masks were updated
        # on the current iteration to calculate sorter's performance
        for pl in range(Planes-1):
            modes = Modes_in[pl, :, :, :]*Masks_complex[pl, :, :]
            modes = propagate_HK(modes, H0)
            Modes_in[pl+1, :, :, :] = modes

        modes = modes*Masks_complex[Planes-1,:,:] # add the last phase mask
        psi = modes # go from the last plane of the MPLC to the output plane
        psi_int_only = (torch.abs(psi))**2 # intensities in the output plane

        # calculate and print out sorter's performance
        # return an average localized fidelity and a full list of localized fidelities
        fid, fid_list = performance_loc_fidelity(psi, Gaussian_Masks_torch, phi)
        # return an average cross-talk, a list of average cross-talks for each mode and a cross-talk matrix
        crs, crs_list, crs_matrix = performance_crosstalk(psi_int_only, Gaussian_Masks_torch)
        # return an average efficiency and a full list of efficiencies
        eff, eff_list = performance_efficiency(psi_int_only, Gaussian_Masks_torch)

        print('iteration', i, ': loc. fidelity =', round(fid.detach().numpy().item(),2), ', crosstalk =', round(crs.detach().numpy().item(),2), ', efficiency =', round(eff.detach().numpy().item(),2))
        crs_array_convergence[conv_count] = crs # store calculated cross-talk to an array to then plot it against the number of iterations

        # stop iterating if the algorithm is no longer improving cross-talk by more than a certain value after a certain iteration
        if i > (iterations/3) and (crs_array_convergence[conv_count-1] - crs_array_convergence[conv_count]) < crs_delta:
            break
        conv_count = conv_count + 1

        # store a list of a relative efficiency of every output on the current iteration to try to equalize them on the next run
        if equalize_efficiency == 1:
            eff_distribution = eff_list/torch.max(eff_list)
            # plot efficiency distribution if plot_eff_distribution is on
            if plot_eff_distribution == 1:
                plt.plot(eff_distribution)
                plt.title('efficiency distribution')
                plt.ylim((0,1))
                plt.show()

# calculate and print out sorter's performance after the last iteration
fid, fid_list = performance_loc_fidelity(psi, Gaussian_Masks_torch, phi)
crs, crs_list, crs_matrix = performance_crosstalk(psi_int_only, Gaussian_Masks_torch)
eff, eff_list = performance_efficiency(psi_int_only, Gaussian_Masks_torch)

print('Final performance: loc. fidelity =', round(fid.detach().numpy().item(),3), ', crosstalk =', round(crs.detach().numpy().item(),3), ', efficiency =', round(eff.detach().numpy().item(),3))
```

With same configuration that is 10 HG modes, 10 phase mask plane and 50 iteration we achieved %88 fidelity and maximum % 3 cross-talk.

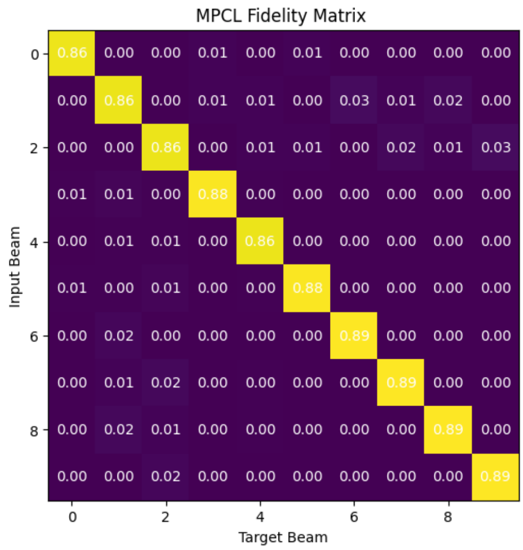


Figure 7 Fidelity Matrix of GAB

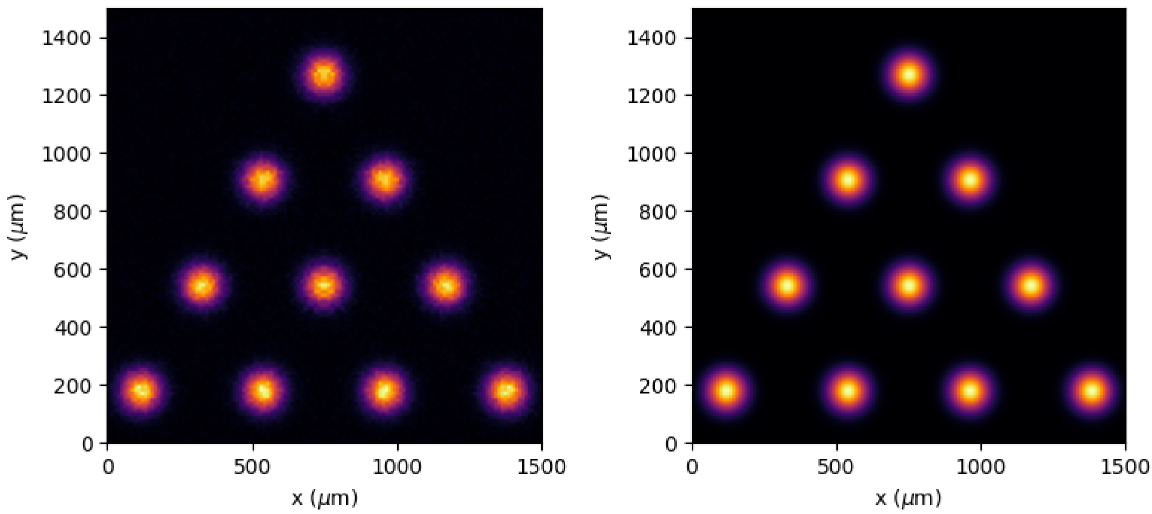


Figure 8 Sum of all the intensities of (on the left) output mode and (on the right) desired output using HG basis with GAB algorithm

Comparison Between Gradient Ascent and Wavefront Matching

The wavefront matching method (WMM) optimizes the overlap between the target output modes ϕ_i and the actual output modes ψ_i. The objective function is defined as

$$F_T = \left| \sum_{i=1}^{N} (\phi_i^* \psi_i) \right|^2 .$$

Substituting $\psi_n = S \cdot \chi_n$ and factoring S into components that isolate the m-th phase mask, we have

$$F_T = \left| \sum_{i=1}^{N} \phi_i^* S \chi_i \right|^2 = \left| \sum_{i=1}^{N} v_{i,>} \cdot P_m \cdot v_{i,<} \right|^2 .$$

Expanding this as a sum over pixels p, the function becomes

$$F_T = \left| \sum_{p=1}^{P} e^{i\theta_{m,p}} \sum_{i=1}^{N} (v_{i,>})_p (v_{i,<})_p \right|^2 .$$

The phase for each pixel is updated as

$$\theta_{m,p} = -\arg\left[\sum_{i=1}^{N}(v_{i,>})_p(v_{i,<})_p\right].$$

The optimal phase values for all P pixels on plane m can be calculated simultaneously using this method, with the wavefront matching method (WFM) iteratively cycling through all M planes until convergence.

WFM shares similar forward and backward field propagation steps with the gradient ascent optimizer, as both are forms of adjoint optimization. However, unlike gradient ascent, which computes δF_T and incrementally adjusts the phase of each plane, WMM directly computes larger, spatially varying phase changes for each plane in one step. This direct computation allows WFM to converge more rapidly compared to gradient ascent.

It is important to note that this direct calculation of optimal phase adjustments is feasible only for objective functions based on maximizing the overlap integral between output and target spatial modes.

Conclusion

Multi-Plane Light Conversion (MPLC) is a powerful technique for manipulating optical fields, offering precise transformations between spatial modes through phase mask optimization. This report presented two algorithms, the Wavefront Matching Method (WFM) and the Gradient Ascent-Based (GAB) method, for computing the phase masks necessary for spatial mode sorting.

The WFM demonstrated rapid convergence due to its direct computation of optimal phase adjustments, achieving 90% fidelity with minimal cross-talk (<3%) for Hermite-Gaussian (HG) modes. In contrast, the GAB method offered a flexible approach to balancing fidelity, cross-talk suppression, and efficiency, achieving comparable performance (88% fidelity) but requiring more iterations due to its incremental phase update mechanism. Both algorithms underscore the importance of optimizing phase masks to enhance light control for applications in imaging, quantum optics, and optical communications.

By comparing the algorithms, this study highlighted that while WFM is computationally efficient for scenarios prioritizing overlap integral maximization, GAB provides a more customizable framework for diverse objective functions. The findings demonstrate the versatility of MPLC in achieving high-performance mode sorting, paving the way for further innovations in optical system design and implementation.

Bibliography

[1] David B. Phillips Hlib Kupianskyi Simon A. R. Horsley. “High-dimensional spatial mode sorting and optical circuit design using multi-plane light conversion”. In: (2023).

[2] Haoshuo Chen David T. Neilson Kwangwoong Kim Joel Carpenter Nicolas K. Fontaine Roland Ryf. “Laguerre-Gaussian mode sorter”. In: (2019).

[3] Toshikazu Hashimoto Hiroshi Takahashi Yohei Sakamaki Takashi Saida. “New Optical Waveguide Design Based on Wavefront Matching Method”. In: (2007).

[4] Nicolas K. Fontaine Yuanhang Zhang. “Multi-Plane Light Conversion: A Practical Tutorial”. In: (2023).