Internship Report

Topic: Report on Chromatix

Author:

Ziyi Xiong

Master Student in Photonics at ASP

Internship Period:

01-09-2024 to 31-03-2025

Supervisor:

Prof. Dr. Vladan Blahnik, Institute of Applied Physics

Industry Advisor:

Prof. Dr. Frank Wyrowski, President of LightTrans GmbH

Contents

Chapter 1

Introduction

1.1 Context of the Internship

The internship was conducted under the joint guidance of my university supervisor, Prof. Dr. Vladan Blahnik, and my industry advisor, Prof. Dr. Frank Wyrowski. The project entailed a deep-dive analysis into the open-source Python library known as **Chromatix**, which was originally developed at HHMI Janelia Research Campus. This library leverages wave-optics and advanced computational frameworks (notably JAX and FLAX) to enable *differentiable* simulations of optical systems.

My role was to thoroughly understand Chromatix, from its underlying wave-optical theory to the structure of its Python source code, and to explore real use cases. Specifically, I examined how classes, functions, and modules contribute to building optical elements, performing wave propagation, and enabling inverse-design tasks such as optimizing lens parameters and masks.

1.2 Goals and Objectives

The primary goals of this internship were:

1. **Comprehensive Code Analysis:** Understand and document the organization and functionality of the Chromatix library (classes, functions, modules).
2. **Theoretical Foundations:** Relate each function's or class's implementation to the fundamental physics of wave optics (e.g., Fresnel diffraction, Fourier transforms, Jones calculus, etc.).
3. **Practical Demonstrations:** Demonstrate usage through multiple example systems (simple widefield PSF simulation, a 4f system, and a CGH setup using a Digital Micromirror Device (DMD)).
4. **Evaluation & Future Prospects:** Critically assess strengths, limitations, and propose future directions for library development.

1.3 Structure of This Report

This report is divided into the following chapters:

- **Chapter 1: Introduction** - Provides an overview of the internship objectives and a brief introduction to Chromatix. Delves into the design principles, module structure, and theoretical basis of the library.
- **Chapter 2: Core Source Code Analysis** - Explores the main classes and functions in Chromatix, mapping them to wave-optical principles.
- **Chapter 3: Practical Usage Examples** - Demonstrates real-world simulations using Chromatix with extended code snippets.
- **Chapter 4: Conclusions & Future Work** - Summarizes key findings and offers recommendations for further improvements.

Chapter 2

Chromatix in Detail

2.1 Overview

Chromatix is a differentiable wave-optics library designed to model complex optical systems while allowing gradient-based optimization, thanks to the JAX ecosystem. Key features include:

- **GPU Acceleration**: Offloading large-scale computations to GPUs or TPUs.
- **Inverse Design**: Using auto-differentiation to iteratively refine system parameters (e.g., lens curvature, phase masks).
- **Modular Architecture**: Dividing code into multiple packages (`data`, `functional`, `elements`, etc.) for clarity and maintainability.
- **Multi-Wavelength & Polarization Handling**: Scalar and vector fields, and spectral weighting.

2.2 Package Breakdown

Chromatix is organized into six major packages plus two additional modules:

- **data**: Generates fundamental optical patterns (radial shapes, grayscale images) and can create permittivity tensors for materials. While not central to wave-propagation, it provides ready-made data for simulation.
- **functional**: Implements physical transformations like Fresnel or Angular Spectrum propagation, lens-induced phase changes, amplitude/phase masks, polarizers, and sensor operations. This package is the mathematical backbone, reliant heavily on JAX (`jax.numpy`).
- **elements**: Encapsulates optical components as trainable modules (subclasses of `flax.linen.Module`). For example, *lens elements*, *phase mask elements*, etc. This design allows each element to be optimized via gradient-based methods.
- **systems**: Provides the `OpticalSystem` class to combine multiple *elements* into a pipeline. Example systems include a standard microscope (`Microscopy`) and a 4f system (`Optical4FSystemPSF`).

- **ops**: Supplies typical image-processing utilities (filters, noise, quantization) that can be integrated into pipeline simulations.
- **utils**: Hosts a variety of helper functions (FFT wrappers, 2D/3D shape generation, Zernike polynomials, wavefront aberrations, etc.) to support the core simulation tasks.
- **field.py**: Defines the `Field` class (with `ScalarField` and `VectorField`) that serve as the data structure for complex wave fields. Includes sampling intervals, wavelength arrays, and advanced indexing.
- **__init__.py**: Basic initializations, ensuring that the library is well-structured upon import.

2.3 Core Theoretical Concepts

Chromatix leverages fundamental wave optics:

1. **Fourier Optics**: The library heavily uses Fourier transforms to handle propagations (Fresnel, Fraunhofer, Angular Spectrum).
2. **Jones Calculus**: Particularly for vector fields, polarizers, and wave plates.
3. **Sampling Theory**: Ensuring that the discretized fields respect sampling requirements, particularly for FFT-based methods.
4. **Differentiable Programming**: JAX's automatic differentiation engine allows every wave transform to be differentiable w.r.t. design parameters.

Chapter 3

Core Source Code Analysis

3.1 Field Class (`field.py`)

The **Field** class (and its scalar/vector variants) is the center piece:

- **u**: A complex array representing the electromagnetic field samples. Typically shaped as $(B..., H, W, C, [1|3])$, where $H \times W$ is the spatial grid, C is the number of spectral channels, and $[1|3]$ indicates scalar or vector fields.
- **_dx**: Sampling intervals (y, x) specifying real-space pixel sizes.
- **_spectrum**: One or more wavelengths used in the field (multi-wavelength simulations supported).
- **_spectral_density**: Respective weights of each wavelength.

Key Methods include:

- `grid()`: Returns spatial coordinate arrays, centered in the middle of the array.
- `k_grid()`: Returns frequency coordinates (Fourier domain) suitable for propagation computations.
- `phase()`: Extracts the phase of the complex field.
- `intensity()`: Computes the intensity $I = |u|^2$.
- `amplitude()`: Computes the amplitude $|u|$.
- Arithmetic operators (`__add__`, `__mul__`, etc.) that permit easy combinations of fields.

Physical Rationale: By bundling sampling intervals and spectral data within `Field`, Chromatix simplifies transform-based operations (Fourier transforms, lens phases, etc.). The methods ensure consistent coordinate handling.

3.1.1 Why JAX?

JAX transforms each function into a computation graph that can be differentiated. For wave optics, each transformation (e.g., lens phase or Fresnel propagation) is re-cast in a way that JAX can keep track of partial derivatives. This is extremely useful for tasks such as inverse design, where we want to adjust lens surfaces or mask patterns to achieve a target field.

3.2 Functional - Module & Function Overview

The **functional** package provides static (pure) functions implementing standard optical transformations:

1. amplitude_masks.py

- `amplitude_change`: This function perturbs a Field object by a given amplitude array.

2. convenience.py

- `optical_fft`: This `optical_fft` function performs a Fresnel propagation on a Field object using a Fast Fourier Transform (FFT) or Inverse Fast Fourier Transform (IFFT) depending on the propagation direction.

3. lenses.py

- `thin_lens`: This function is to calculate the field after passing a thin lens.
- `ff_lens`: This function simulates a situation where a field propagates a distance f to a thin lens, then it passes through the thin lens and then propagates another distance f to the focal plane of the lens.
- `df_lens`: This function simulates a more general case of 2f setup, where the first f becomes d meaning that the initial field starts from any distance before the thin lens.

4. phase_masks.py

- `spectrally_modulate_phase`: This function ensures accurate modelling of optical systems involving multiple wavelengths.
- `phase_change`: This function aims to perturb the input field by an input phase array.
- `wrap_phase`: The function serves to adjust phase values so that they fall within a specified range.

5. polarizers.py

- `jones_vector`: This function creates a Jones vector with the inputs theta and beta, where theta is the polarization angle and beta is the phase delay.
- `linear`: This function is for generating a Jones vector for linearly polarized light.
- `left_circular`: This function generates a Jones vector for left-hand circularly polarized light.
- `right_circular`: This function generates a Jones vector for right-hand circularly polarized light.

- `polarizer`: This function is deigned to simulate polarizers such as linear polarizer, left circular polarizer, and right circular polarizer.
- `linear_polarizer`: This function applies a linear polarizer to the input field at a given angle with respect to the horizontal.
- `left_circular_polarizer`: This function applies a left circular polarizer to the incoming field.
- `right_circular_polarizer`: This function applies a right circular polarizer to the incoming field.
- `phase_retarder`: This simulates a phase retarder by applying a general phase retarder Jones matrix to the input field to modify its polarization state.
- `wave_plate`: This function simulates a general wave plate, also known as linear phase retarder by calling the phase_retarder function.
- `halfwave_plate`: Halfwave plate makes the light's orthogonal components have a phase difference of π (half a wave) after the light passer through the phase retarder.
- `quarterwave_plate`: Quarter wave plate makes the light's orthogonal components have a phase difference of $\pi/2$.
- `universal_compensator`: A universal compensator, also known as universal polarizer, can be used to generate any polarization.

6. propagation.py

- `transform_propagate`: This function performs Fresnel propagation of an optical field over a specific distance using the Single-FFT Fresnel Propagation Method (SFT-FR).
- `compute_sas_precompensation`: This is to compute the precompensation factor used in the Scaled Angular Spectrum (SAS) method.
- `kernel_propagate`: This function simulates to propagate a field using a given propagator with Fourier convolution.
- `transform_propagate_sas`: This function offers a method to do the single-FFT Fresnel propagation method with the precompensation factor.
- `compute_transfer_propagator`: This function computes the propagation kernel (transfer function) for the Fresnel propagation.
- `transfer_propagate`: This function simply calls the function compute_transfer_porpgator to compute the propagator and the function kernel_propgate to compute the propagated field.
- `compute_exact_propagator`: Compute propagation kernel for propagation with no Fresnel approximation.

- `exact_propagate`: This function simply calls the function compute_exact_porpgator to compute the propagator and the function kernel_propgate to compute the propagated field.
- `compute_asm_propagator`: This propagation kernel is also without Fresnel approximation, but the phase of the kernel could contain a shift in output plane.
- `asm_propagate`: Propagate field for a distance z using angular spectrum method.
- `compute_padding_transform`: This is to compute the padding for transform propagation.
- `compute_padding_transfer`: Automatically compute the padding required for transfer propagation.
- `compute_padding_exact`: Automatically compute the padding required for exact propagation.

7. pupils.py

- `circular_pupil`: This function creates a circular mask and applies it to the incoming field.
- `square_pupil`: This function creates a squared mask similarly to the circular_pupil function.

8. samples.py

- `jones_sample`: The jones_sample function simulates how an incoming vector field is perturbed when it passes through a thin sample.
- `thin_sample`: This is to simulate the perturbation to an incoming scalar field based on the thin sample approximation.
- `multislice_thick_sample`: The purpose of this function is to simulate a scalar field propagating through a thick sample by modelling the thick sample as multiple thin samples.
- `fluorescent_multislice_thick_sample`: This function simulates the propagation of a scalar field through a thick, transparent, and fluorescent sample.
- `PTFT`: This function is to create a projection tensor for transverse wave (PTFT). This projection tensor projects vectors onto the plane perpendicular to the wave vector $\mathbf{k}$.
- `thick_sample_vector`: This function simulates a vectorial wave field with scattering potential propagating through a thick sample.

9. sensors.py

- `basic_sensor`: Produces an intensity image from an incoming field or intensity array and simulates shot noise.

10. sources.py

- `point_source`: This function can generate a scalar field or vectorial field located at a distance z from the point source.
- `objective_point_source`: This function simulates the field that just after passing through a lens with focal length f and numerical aperture NA. The field is originated from the position which is deviated a distance z from the object space's focal plane.
- `plane_wave`: This is to create field simulating a plane wave.
- `generic_field`: The function creates either a scalar field or vectorial field with a self-defined phase which determines the shape of the wavefront.

3.3 Elements Package

The **elements** package wraps functions into `flax.linen.Module` classes, making them trainable modules.

1. amplitude_masks.py

- `AmplitudeMask`: Applies an amplitude mask to an incoming Field by calling the function amplitude_change from the functional package. The amplitude can be learned pixel by pixel.

2. convenience.py

- `Flip`: This element flips the incoming field upside down using jnp.flip().
- `ScaleAndBias`: This class simply adds a bias to the incoming field and multiplies the sum by a scale.
- `Binarize`: This class aims to binarize the incoming 'Field' by calling the function binarize from chromatix.ops.quantization.
- `Quantize`: This class applies the quantize function from chromatix.ops.quantization to the incoming Field to quantize the data to a specific bit depth.

3. lenses.py

- `ThinLens`: It models the behavior of a thin lens in an optical system by taking three attributes (can be learned) to apply the effect of a thin lens to an incoming field and returns the modified field.
- `FFLens`: Applies a thin lens placed a distance f after the incoming field. This element returns the field a distance f after the lens. The attributes f, n, and NA can be learned.
- `DFLens`: Applies a thin lens placed a distance d after the incoming field. This element returns the field a distance f after the lens. The attributes d, f, n, and NA can be learned.

4. phase_masks.py

- `PhaseMask`: Applies a phase mask to an incoming field by calling the phase_change function. The attributes phase, f, n, NA can be learned.
- `SpatialLightModulator`: Simulates a spatial light modulator (SLM) applied to an incoming field. This element acts as if the SLM is phase only and transmits a field, rather than reflecting it.
- `SeidelAberrations`: Applies Seidel phase polynomial to an incoming field. The coefficients of the polynomials can be learned.
- `ZernikeAberrations`: Applies Zernike aberrations to an incoming field. The coefficients of the polynomials can be learned.

5. propagation.py

- `Propagate`: Free space propagation that can be placed after or between other elements. The method of propagation can be chosen from transform, transfer, exact, asm. The attributes z and n can ba learned
- `KernelPropagate`: This class performs free space propagation using a precomputed propagation kernel (propagator) to propagate an incoming field.

6. samples.py

- `ThinSample`: This class is basically a wrapper that wraps the function thin_sample for training. All the three attributes(absorption, dn, thickness) it takes could be either predefined as an array or a callable for training.

7. sensors.py

- `BasicSensor`: The aim of this class is to produce an image representing the intensity of the incoming field with optional shot noise.

8. sources.py

- `PointSource`: Generates field due to point source a distance "z" away. It can also be given pupil. The attributes "z", "n", "power", and "amplitude" can be learned.
- `ObjectivePointSource`: Generates field due to a point source defocused by an amount z away from the focal plane, just after passing through a lens with focal length f and numerical aperture NA. The attributes f, n, NA, and power can be learned.
- `PlaneWave`: Generates plane wave of given phase and power.It can also be given pupil and kykx vector to control the angle of the plane wave. The attributes kykx, power, and amplitude can be learned.
- `GenericField`: This is a class built upon the generic_field function that can be used to generate an electromagnetic field with arbitrary amplitude and phase. Among the attributes, amplitude, phase, and power can be learned.

8. utils.py

- `Trainable`: This Trainable class is used to mark and store a value as a trainable parameter which should be learned and optimized.
- `trainable`: This function wraps an input x into a Trainable object to signal to a Chromatix element (lenses, sensors and so on) that x should be used to initialize a trainable parameter.
- `register`: This is used to register attributes of a Flax module, namely the classes defined with nn.Module after the class name, as trainable parameters or fixed state variables.
- `parse_init`: This function simply defines an inner function init to turn x into an initializer function when x is not a function.

3.4 Systems Package

1. optical_system.py

- `OpticalSystem`: The aim of this class is to combine a sequence of optical elements defined in the elements package in to a single module. Therefore, it can simulate a complex optical system which accepts a field as input, and the field will be manipulated by the sequence of elements one by one, then returned as the output of the optical system.

3.5 Performance Considerations

Chromatix performance depends on array shapes and the complexity of transformations. JAX and XLA (Accelerated Linear Algebra) can compile repeated operations for speed. However, high-resolution fields can be memory-intensive, especially if running multiple parallel wave calculations (e.g., multi-wavelength or batch). GPU usage is recommended.

Chapter 4

Application Examples

4.1 Usage Example 1: Simulating a Simple Optical System

This example demonstrates how to simulate a basic optical system in Chromatix using three optical elements: an objective point source, a phase mask, and a thin lens.

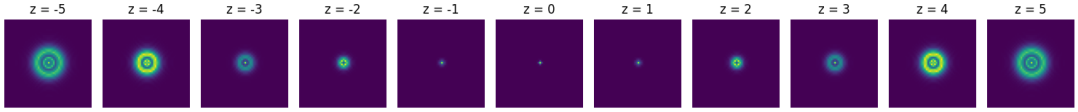
```
import chromatix
import chromatix.elements
import jax
import jax.numpy as jnp

shape = (512, 512)
spacing = 0.3
spectrum = 0.532
spectral_density = 1.0
f = 100.0
n = 1.33
NA = 0.8

optical_model = chromatix.OpticalSystem([
    chromatix.elements.ObjectivePointSource(shape, spacing, spectrum, spectral_densit
    chromatix.elements.PhaseMask(jnp.ones(shape)),
    chromatix.elements.FFLens(f, n)
])

variables = optical_model.init(jax.random.PRNGKey(4), jnp.linspace(-5, 5, 11))
widefield_psf = optical_model.apply(variables, jnp.linspace(-5, 5, 11)).intensity
```

In this code:

- `ObjectivePointSource` models a point source imaged through an objective lens.
- `PhaseMask` applies a uniform phase mask (no actual phase change).
- `FFLens` simulates free-space propagation from the front to back focal plane of a lens.

- The PSF is computed across 11 defocus distances using JAX parallelization.

To visualize the point spread function (PSF), the output is plotted across the range of defocus values. The intensity profile matches expected optical behavior — a focused point at `z = 0` and rings with increased defocus.



4.2 Usage Example 2: Building a 4f Optical System

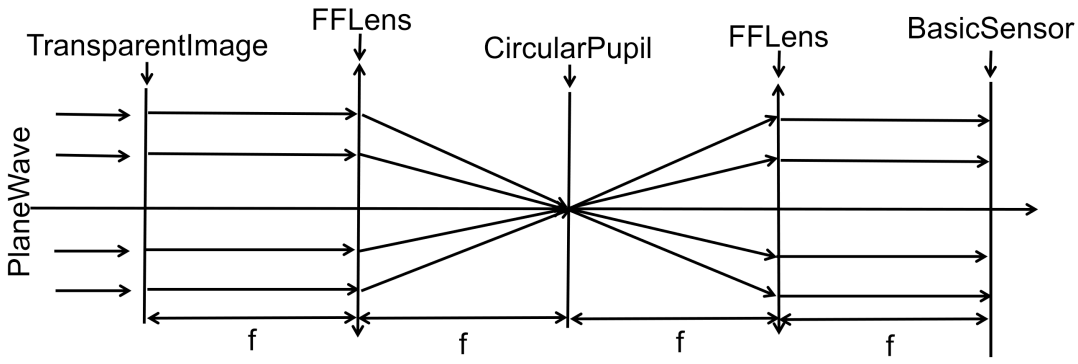
In this example, we construct a 4f imaging system using `Chromatix`. Since Chromatix does not provide built-in classes for a transparent image or circular pupil, we define them manually.

Custom Components

```
def transparent_image(field, image, transparency):
    # Resize image to match field and apply transparency
    image = _broadcast_2d_to_spatial(image, field.ndim)
    image = center_crop(image, [...])  # Center crop to match field size
    return field.replace(u=field.u * jnp.sqrt(image * transparency))

class TransparentImage(nn.Module):
    image: Array
    transparency: float
    def __call__(self, field):
        return transparent_image(field, self.image, self.transparency)

class CircularPupil(nn.Module):
    w: float
    def __call__(self, field):
        return circular_pupil(field, self.w)
```

4.2.1 Constructing the 4f System

We are going to create such a 4f system:

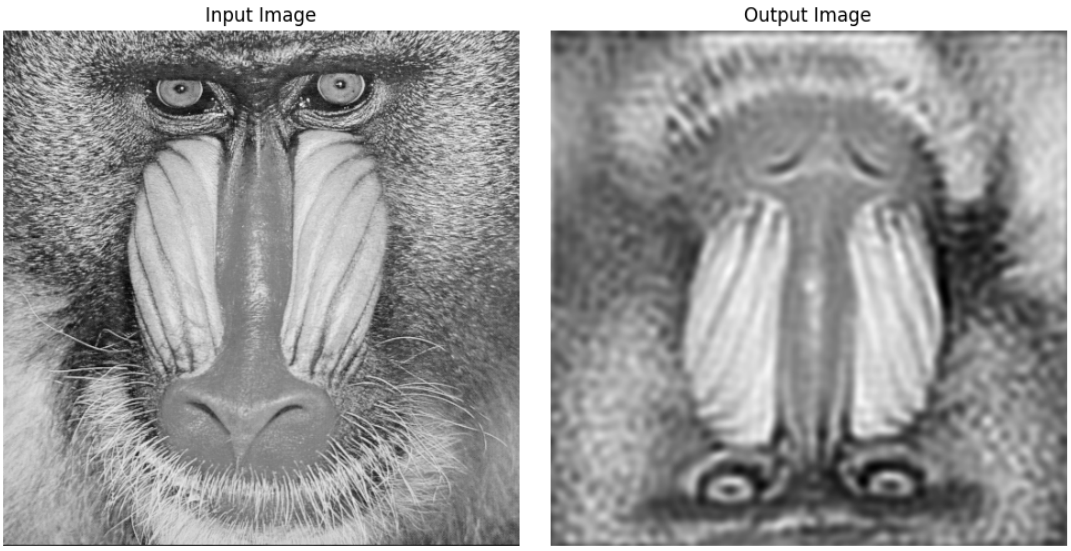


```
import chromatix
import chromatix.elements
import jax.numpy as jnp
from PIL import Image

# Load input image and define parameters
image = jnp.array(Image.open("Monkey.png"))
shape = (512, 512)
spacing = 0.3
spectrum = 0.532
spectral_density = 1.0
f = 100.0
n = 1.33

# Define optical model with 4f configuration
optical_model = chromatix.OpticalSystem([
    chromatix.elements.PlaneWave(shape, spacing, spectrum, spectral_density),
    TransparentImage(image, transparency=0.1),
    chromatix.elements.FFLens(f, n),
    CircularPupil(w=20),
    chromatix.elements.FFLens(f, n),
    chromatix.elements.BasicSensor(shape, spacing)
])

# Initialize and simulate
variables = optical_model.init(jax.random.PRNGKey(4))
output_image = optical_model.apply(variables)
```

4.2.2 Visualization of Input and Output

```
import matplotlib.pyplot as plt

fig, axes = plt.subplots(1, 2, figsize=(10, 5))
axes[0].imshow(image.squeeze(), cmap='gray')
axes[0].set_title("Input Image")
axes[0].axis("off")

axes[1].imshow(output_image.squeeze(), cmap='gray')
axes[1].set_title("Output Image")
axes[1].axis("off")

plt.tight_layout()
plt.show()
```

The code above loads an image, constructs a 4f optical system, and simulates the output image. The input image is displayed on the left, while the output image is shown on the right.



The result confirms the 4f system's theoretical prediction: the image passes through a lens-pupil-lens sequence and forms an output image at the sensor plane.

4.3 Usage Example 3: Computer-Generated Holography (CGH) with DMD

This example, provided by one of the Chromatix authors, demonstrates a method for creating a computer-generated hologram using a binary amplitude mask implemented via a digital micromirror device (DMD).

4.3.1 Background

In CGH, a wavefront is modulated to form a target image at a certain propagation distance. A DMD acts as a binary amplitude mask with ON/OFF states, approximating this modulation. The optimization of such a mask allows the reconstruction of the desired image at the target plane.

4.3.2 The CGH Class Implementation

```
from chromatix.systems import OpticalSystem
from chromatix.elements import AmplitudeMask, PlaneWave, trainable
from chromatix.functional import transfer_propagate

class CGH(nn.Module):
    amplitude_init: Callable = jax.nn.initializers.uniform(1.5)
    shape: Tuple[int, int] = (300, 300)
    spacing: float = 7.56
    f: float = 200.0
    n: float = 1.0
    N_pad: int = 0
    spectrum: Array = 0.66
    spectral_density: Array = 1.0
```

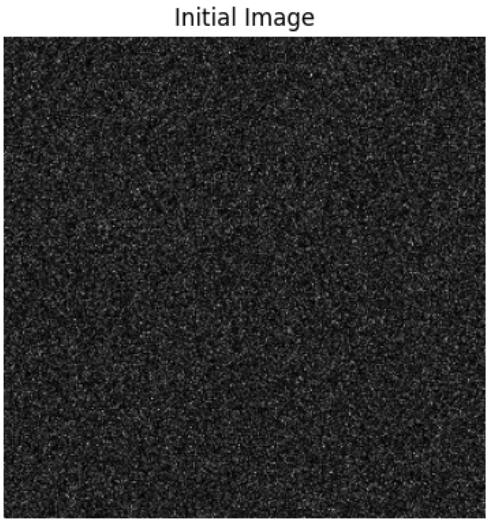
```
    @nn.compact
    def __call__(self, z: Union[float, Array]) -> Field:
        system = OpticalSystem([
            PlaneWave(shape=self.shape, dx=self.spacing, spectrum=self.spectrum,
                      spectral_density=self.spectral_density),
            AmplitudeMask(trainable(self.amplitude_init), is_binary=True)
        ])
        return transfer_propagate(system(), z, self.n, self.N_pad)
```

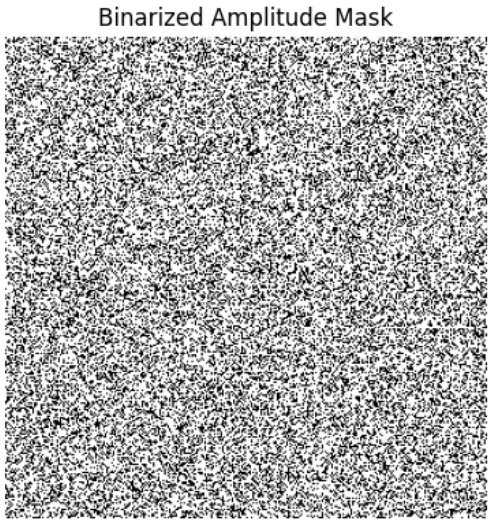
4.3.3 Initialization and Visualization

In this part, we create a CGH system using Chromatix:

```
model = CGH()
z = 13e4
variables = model.init(jax.random.PRNGKey(4), z)
params, state = variables["params"], variables["state"]

init_img = model.apply({"params": params, "state": state}, z).intensity.squeeze()
init_amp_mask = np.float32(params["AmplitudeMask_0"]["_amplitude"] > 0.5)
```

The initial image and binarized amplitude mask are visualized using `matplotlib`. This provides a baseline before training begins:





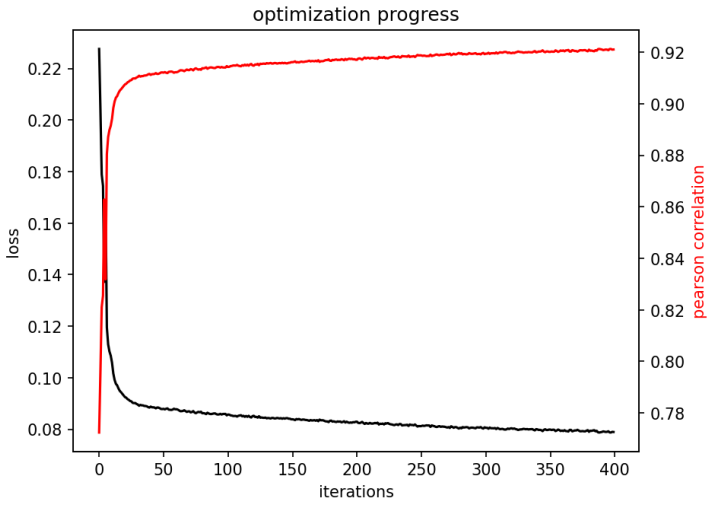
4.3.4 Target Image and Loss Function

```
from skimage.data import cat
img = cat().mean(2)[:, 100:400]
data = jnp.array(img)

def loss_fn(params, state, data, z):
    approx = model.apply({"params": params, "state": state}, z=z).intensity.squeeze()
    loss = optax.cosine_distance(approx.reshape(-1), data.reshape(-1)).mean()
```

```
    correlation = jnp.sum(approx * data) / (
        jnp.sqrt(jnp.sum(approx**2) * jnp.sum(data**2)) + 1e-6)
    return loss, {"loss": loss, "correlation": correlation}
```

The target image is a cut gray cat image:

And the loss function is defined as the cosine distance between the generated and target images. The correlation metric is also computed to track optimization progress:



4.3.5 Training the Hologram

```
from flax.training.train_state import TrainState

trainstate = TrainState.create(
    apply_fn=model.apply,
    params=params,
    tx=optax.adam(learning_rate=2)
)

grad_fn = jax.jit(jax.grad(loss_fn, has_aux=True))
```

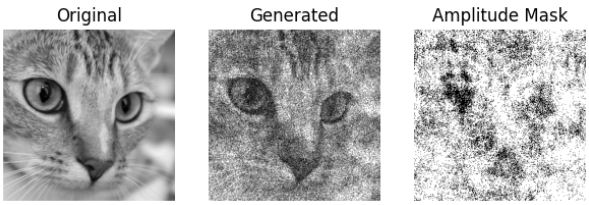
```
for iteration in range(400):
    grads, metrics = grad_fn(trainstate.params, state, data, z)
    trainstate = trainstate.apply_gradients(grads=grads)
    ...
```

The optimization loop uses the Adam optimizer to update the amplitude mask values and track loss and correlation over iterations:

4.3.6 Results

The final generated image closely matches the target, and the amplitude mask converges to a binary pattern suitable for DMD implementation. The optimization history is visualized via line plots of loss and correlation. A comparison of the target and generated images is also provided.

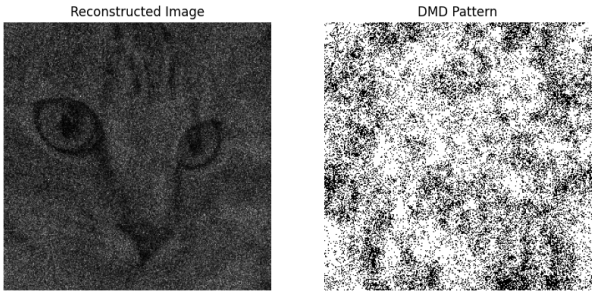
```
plt.subplot(1, 4, 1); plt.imshow(data, cmap="gray")
plt.subplot(1, 4, 2); plt.imshow(approx1, cmap="gray")
plt.subplot(1, 4, 3); plt.imshow(np.float32(dmd > 0.5), cmap="gray")
```



4.3.7 Alternative: Manual Implementation

An equivalent model was also built manually using FFT-based Fresnel propagation and optimized using Optax's Adam optimizer, confirming the results obtained with Chromatix is better than that obtained with normal adam algorithm.

Here is the result of the manual implementation:



This demonstrates Chromatix's strength in modeling and optimizing complex optical systems using differentiable programming techniques and neural network.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

Throughout this internship, I dedicated considerable time to analyzing not only Chromatix's core functionalities and theoretical underpinnings but also its architecture and broader potential for driving innovation in optical research. By carefully studying every line of code and mapping it to the physics of wave optics, I gained a deep appreciation for how Chromatix harnesses JAX's auto-differentiation.

Key accomplishments and insights include:

1. **Deep Understanding of Code-Physics Correlation**: The direct relationships between Fourier-based propagation methods and the library's functions are clearly documented, revealing an elegant balance between computational efficiency and physical accuracy.

2. **Hands-on Experiments**: The practical examples I ran—like the 4f system and the CGH simulation—highlight how Chromatix can serve both forward modeling (predicting system outputs) and inverse tasks (optimizing masks or system configurations).

3. **Potential for Cross-Disciplinary Use**: While originally conceived for optics research, the modular design and open-source nature suggest it could be extended or integrated with adjacent fields (e.g., computational imaging, deep learning for inverse problems) quite seamlessly.

From a practical standpoint, Chromatix presents a healthy balance between user-friendliness and advanced capabilities. Even so, the library's potential hinges on continued development and collaboration. By blending wave-optics simulations with the JAX ecosystem, it paves the way for new paradigms in designing, simulating, and optimizing complex photonic systems.

Despite its strong foundation, Chromatix still faces several notable challenges:

1. **Limited Optical Elements**: While the `elements` package covers common components like lenses and basic polarizers, many conventional and advanced elements (e.g., specialty diffractive optical elements, gradient-index lenses, complex polarization optics) are absent. This restricts the variety of systems that can be directly simulated out of the box

2. **Approximate Modeling**: Chromatix often relies on paraxial or thin-lens approximations. Real-world scenarios that demand full vector wave solutions, non-paraxial

modeling, or thick-lens corrections may only be partially captured without additional code.

3. **Steep Learning Curve**: Mastering JAX transformations, wave-optics concepts, and Chromatix's object model all at once can be challenging for newcomers.

5.2 Future Work

1. **Broader Element Coverage**: Introduce new classes for advanced lenses (thick, GRIN), diffractive elements, polarization manipulators (birefringent crystals, Faraday rotators), and partial coherence modeling. This would significantly widen the scope of practical optical systems that can be studied.

2. **Enhanced Physical Accuracy**: Address non-paraxial regimes, volumetric scattering, nonlinear effects, and other factors beyond simple approximations. Techniques like beam propagation methods (BPM) or finite-difference time-domain (FDTD) expansions (if feasible) might further increase realism.

3. **Extended Documentation and Tutorials**: Publish advanced example notebooks demonstrating specialized setups (microscopy with aberration compensation, computational holographic displays, metasurface optimization, etc.). This would attract a broader user community and encourage collaboration.

In conclusion, Chromatix stands at an exciting intersection of optical physics and machine learning. By capitalizing on JAX's auto-differentiation, it paves the way for novel inverse-design methodologies in photonics. Continued enhancements—particularly around element diversity, accuracy, performance, and ease of use—would solidify its role as a powerful engine for both academic research and industrial optical design.

Chapter 6

Acknowledgments

I wish to express my sincere gratitude to my supervisor, Prof. Dr. Vladan Blahnik, for supervising my internship.

Special thanks to Prof. Dr. Frank Wyrowski, CEO of LightTrans GmbH, for providing the internship topic, invaluable insights, and continuous support throughout the project. And thanks to my colleagues especially Mr. Dominik and Mr. Christian at LightTrans GmbH for their assistance and collaboration.

Appendix A

Additional Material

The Report on Chromatix is written in JupyterNotebook which can be found on my github repository. The report contains all the code snippets and figures used in this report. The link to the repository is as follows: `https://github.com/Xiong-Ziyi/Chromatix-Report/blob/main/Report_on_Chromatix.ipynb`