Abbe Center of Photonics | JENA
Friedrich-Schiller-Universität

Institute of Applied Physics
Friedrich-Schiller-Universität Jena

Internship

Report

on *Chromatix*

Author: Ziyi Xiong
Master Student in Photonics, Abbe School of Photonics

Internship Period: 01-09-2024 to 28-02-2025

University 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 each class, function, and module contributes 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.
- **Chapter 2: Chromatix in Detail** - Delves into the design principles, module structure, and theoretical basis of the library.
- **Chapter 3: Core Source Code Analysis** - Explores the main classes and functions in Chromatix, mapping them to wave-optical principles.
- **Chapter 4: Practical Usage Examples** - Demonstrates real-world simulations using Chromatix with extended code snippets.
- **Chapter 5: 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 High-Level 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 centerpiece:

- **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.

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 Package

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

- `propagation`: Single-FFT Fresnel, scalar angular spectrum, convolution-based Fresnel.
- `lenses`: Phase changes simulating thin lenses.
- `polarizers`: Jones matrix transformations for linear, circular, or custom polarization elements.
- `sources`: Plane waves, point sources, or partial coherence fields.
- `phase_masks` / `amplitude_masks`: Introduce user-defined or random phase/amplitude patterns.

3.3 Elements Package

The **elements** package wraps functional transformations into `flax.linen.Module` classes, making them trainable modules. Examples:

- `FFLens`: A lens element that imposes a phase transformation from front to back focal planes.
- `Polarizer` classes for modeling wave plates, polarizer filters, etc.
- `PhaseMask` and `AmplitudeMask`: Potentially trainable masks for inverse design.

3.4 Systems Package

- `OpticalSystem`: A container that sequentially applies elements, enabling complex system simulation (microscopes, telescopes, etc.).
- `Microscopy` and `Optical4FSystemPSF`: Pre-built specialized classes for immediate usage.

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

Practical Usage Examples

In this chapter, I will present three extended examples that illustrate how to create and simulate optical systems with Chromatix. These examples draw from real or hypothetical use cases, accompanied by code snippets.

4.1 Example 1: Widefield PSF with a Lens and Phase Mask

4.1.1 Objective

Simulate a basic widefield point spread function (PSF) by composing an `ObjectivePointSource`, a `PhaseMask`, and a lens element. Investigate how the PSF changes with defocus.

4.1.2 Code Snippet

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

shape = (512, 512)
spacing = 0.3  # microns
spectrum = 0.532  # microns
spectral_density = 1.0
f = 100.0  # focal length in microns
n = 1.33   # refractive index
NA = 0.8   # numerical aperture

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

# Multiple defocus planes:
z_values = jnp.linspace(-5, 5, num=11)

```

```
variables = optical_model.init(jax.random.PRNGKey(4), z_values)
widefield_psf = optical_model.apply(variables, z_values).intensity

# Now 'widefield_psf' is a stack of 11 intensity images.

# Visualization
import matplotlib.pyplot as plt
import numpy as np

fig, axes = plt.subplots(1, len(z_values), figsize=(15, 4))
for i, z in enumerate(z_values):
    psf_img = widefield_psf[i, :, :]
    axes[i].imshow(psf_img, cmap='gray', origin='lower')
    axes[i].set_title(f'z = {z:.1f} microns')
    axes[i].axis('off')

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

Key Observations:

- The lens + phase mask pipeline transforms the source through focal planes.
- Out-of-focus planes show larger spot sizes, while in-focus planes show a tight PSF.
- The entire pipeline is differentiable, so one could optimize the phase mask for a particular PSF shape.

4.2 Example 2: Building a 4f System

4.2.1 Objective

Construct a classic 4f imaging system to illustrate how Chromatix composes multiple elements: a `PlaneWave` source, a lens, a pupil, another lens, and a sensor. This configuration can be used for spatial filtering.

4.2.2 Code Snippet

```
import chromatix
import chromatix.elements
from flax import linen as nn
import jax.numpy as jnp
from chromatix.field import Field

shape = (512, 512)
spacing = 0.3
f = 100.0
n = 1.0

optical_model = chromatix.OpticalSystem([
    chromatix.elements.PlaneWave(shape, spacing, 0.532, 1.0),
    chromatix.elements.FFLens(f, n),
    chromatix.elements.AmplitudeMask(jnp.ones(shape)),  # e.g. a
        transparent mask
    chromatix.elements.FFLens(f, n),
```

```
    chromatix.elements.BasicSensor(shape, spacing)
])

variables = optical_model.init(jax.random.PRNGKey(4))
output_image = optical_model.apply(variables)

# 'output_image' is the intensity measured at the final sensor plane.
```

Key Observations:

- The system is straightforward to read: wave creation, lens, mask, lens, sensor.
- Chromatix seamlessly handles dimension checks, sampling intervals, and wave transforms.
- One could insert a pupil or a specialized filter in the Fourier plane to manipulate the system's transfer function.

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

4.3.1 Objective

Use a binary amplitude mask (mimicking a DMD with ON/OFF states) to generate a target image at a certain propagation distance. This example demonstrates inverse design.

4.3.2 Code Snippet (Highlights)

```
from chromatix.systems import OpticalSystem
from chromatix.elements import AmplitudeMask, PlaneWave
import jax
import jax.numpy as jnp
import optax
from flax.training import train_state

class CGH(nn.Module):
    # This is a simplified illustration.
    shape: tuple = (300, 300)
    spacing: float = 7.56

    @nn.compact
    def __call__(self, z):
        system = OpticalSystem([
            PlaneWave(shape=self.shape,
                      dx=self.spacing,
                      spectrum=0.66),
            AmplitudeMask(trainable_init=jax.nn.initializers.uniform
                (1.5),
                          is_binary=True)
        ])
        return transfer_propagate(system(), z, 1.0)

```

```
# We can define a loss function comparing the resulting field's
    intensity to a target image.
# Then use optax or jax.grad to iteratively update the amplitude mask.
```

Key Observations:

- The amplitude mask is trainable, representing the DMD.
- We define a target image and compute a loss function (e.g., cosine distance) between the simulated intensity and the target.
- Each gradient update modifies the mask, eventually converging on a hologram.

Chapter 5

Conclusions and Future Work

5.1 Conclusions1

Throughout this internship, I dedicated considerable time to analyzing not only Chromatix's core functionalities and theoretical underpinnings but also its 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 and GPU acceleration.

Key accomplishments and insights include:

- **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.
- **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).
- **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.

5.2 Conclusions

Throughout this internship, I thoroughly explored the Chromatix library's architecture, theoretical foundation, and practical use. My main achievements include:

- Mapping each major function to the physical equations it implements (Fresnel, Fraunhofer, lens phase, etc.).

- Verifying the core design principles, especially how JAX integration powers auto-differentiation and GPU acceleration.
- Experimenting with real examples (widefield PSF, 4f system, DMD-based CGH) to confirm the code's clarity and correctness.

Chromatix has proven to be a potent tool for forward and inverse wave-optical simulations, suitable for computational microscopy, digital holography, optical design, and more.

5.3 Limitations

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

- **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.
- **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.
- **Memory Demands**: High-resolution fields for multi-wavelength or large-volume simulations quickly push the memory limits on typical GPUs, making large-scale simulations cumbersome.
- **Incomplete Modules**: Some packages (e.g., `ops` and `systems`) appear less mature than `functional` and `elements`, limiting their broader usefulness for specialized imaging workflows.
- **Steep Learning Curve**: Mastering JAX transformations, wave-optics concepts, and Chromatix's object model all at once can be challenging for newcomers.

5.4 Future Directions

Given Chromatix's potential, there are multiple paths for growth:

- **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.
- **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.

- **Refined Large-Scale Simulations**: Explore multi-GPU or distributed approaches for memory-intensive tasks. This is crucial for highly detailed simulations (e.g., hundreds of megapixels in extended volumes).
- **Integration with Other Ecosystems**: Deeper alignment with frameworks like `JAXopt`, or data pipelines like `tf.data`, could streamline training and large-batch operations. Coupling Chromatix with neural networks for tasks like end-to-end learned optical processing has great promise.
- **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.
- **Community-Driven Development**: Building a more robust user and contributor base can expedite new feature rollouts and drive bug fixes, ultimately ensuring Chromatix remains cutting-edge for wave-optics simulation.

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.

In sum, Chromatix stands at the forefront of differentiable optics simulation, offering a robust, flexible environment for innovation in computational photonics.

End of Report

Appendix A

Appendix

A.1 Supplementary Material

- Additional utility scripts or helper code can be placed here.
- Extended code listings, extra figures, or large data tables.