Image Processing Pipeline Implementation

Overview

When I started this project, I wanted to create something that was both powerful and intuitive. I decided on a modular, object-oriented approach that would make it easy to add new processing steps without modifying existing code. The pipeline pattern seemed like a natural fit - users could stack operations like building blocks to achieve complex transformations.

My Development Process and Thought Journey

Initial Research Phase

When I first read the assignment requirements, I knew I needed to understand several key concepts deeply. I started by researching:

1. **Image Processing Fundamentals**
    - Reviewed the OpenCV documentation for color space conversions
      Links:
        i. https://docs.opencv.org/4.x/df/d9d/tutorial_py_colorspaces.html
        ii. https://docs.opencv.org/3.4/d8/d01/group__imgproc__color__conversions.html
    - Studied NumPy's official documentation for efficient array operations
    - Read several articles on GeeksforGeeks about box blur algorithms and integral images
    - Links:
        i. https://www.geeksforgeeks.org/machine-learning/box-blur-algorithm-with-python-implementation/
        ii. https://www.youtube.com/watch?v=4Eh0y3LHTNU&t=2s
2. **Design Patterns**
    - Consulted "Design Patterns: Elements of Reusable Object-Oriented Software" concepts
    - Looked into the Strategy and Pipeline patterns on refactoring.guru
    - Studied Python's ABC (Abstract Base Classes) documentation to understand proper interface design
3. **Performance Optimization**

- Read research papers on integral images (Viola-Jones paper was particularly helpful)
    i. https://www.face-rec.org/algorithms/Boosting-Ensemble/16981346.pdf
    ii. https://www.ipol.im/pub/art/2014/57/article_lr.pdf
    iii. Integral Images: Efficient Algorithms
- Studied Stack Overflow discussions about NumPy vectorization techniques
- Reviewed scipy.ndimage source code on GitHub to understand how professionals implement image filters

Architecture Decisions

My thought process for the architecture went through several iterations:

First Thought: Simple procedural approach with functions

- Quickly realized this wouldn't scale well
- Hard to add new operations without modifying existing code

Second Thought: Class-based approach with inheritance

- Better, but how to enforce consistency?

Final Decision: Abstract base class with concrete implementations

- Provides a contract that all steps must follow
- Makes the codebase extensible and maintainable
- Allows for easy testing and documentation

Technology Stack Selection

Here's how I decided on each technology:

Python + OpenCV + NumPy

- Required by the assignment, but I would have chosen these anyway
- Consulted the official OpenCV-Python tutorials to refresh my knowledge
- Referenced NumPy's broadcasting documentation extensively

FastAPI over Flask

- Researched modern Python web frameworks on Reddit's r/Python
- FastAPI's automatic API documentation and type hints aligned with my code quality goals
- Watched several YouTube tutorials on FastAPI best practices

Vanilla JavaScript for Frontend

- Initially considered React, but decided it was overkill
- Wanted to demonstrate ability to work without heavy frameworks
- Used MDN Web Docs extensively for modern JavaScript features

Implementation Details

1. Image Processing Library (`image_processing.py`)

The Abstract Base Class Pattern

I used Python's ABC module to enforce a consistent interface across all processing steps. Every step must implement:

- `process()`: The actual image transformation logic
- `get_parameters()`: Returns what parameters the step accepts
- `name` and `description` properties: For UI display

This design makes it incredibly easy to add new processing steps - just inherit from the base class and implement the required methods.

My thinking process here was influenced by SOLID principles, particularly the Open/Closed Principle. I wanted the system to be open for extension but closed for modification.

class ImageProcessingStep(ABC):
    """Abstract base class for all image processing steps."""

Resources that helped:

- Python's official ABC documentation
- Real Python's guide on abstract base classes
- Several Medium articles on Python design patterns

Parameter System Design

I created a `StepParameter` dataclass to define parameters in a type-safe way:

This allows each step to declare its parameters with validation constraints, which the UI can then use to generate appropriate input controls automatically.

I went through several iterations:

1. **First attempt**: Simple dictionaries
    - Too error-prone, no type safety

2. **Second attempt**: Regular classes
    - Better, but verbose

Final solution: Dataclasses

```
 @dataclass
class StepParameter:
    name: str
    type: type
    description: str
    default: Any = None
    min_value: Optional[float] = None
    max_value: Optional[float] = None
```

Resources consulted:

- PEP 557 (Data Classes)
- Raymond Hettinger's PyCon talks on Python's data model

2. Box Blur Implementation - The Technical Challenge

The box blur implementation was challenging. The assignment specifically asked for a custom implementation without using OpenCV's blur functions.

Initial Research

1. Started with Wikipedia's article on box blur
2. Found that naive implementation is $O(r^2)$ per pixel
3. Discovered the integral image technique from:
    - The original Viola-Jones paper "Rapid Object Detection"
    - Stack Overflow discussions on efficient blur implementations
    - OpenCV's own documentation on integral images

Implementation Evolution

First attempt (naive approach):

```
# Too slow for large kernels
for each_pixel:
    sum_neighbors_in_kernel()
    average()
```

Second attempt (separable filter):

- Learned that box blur is separable
- Can do horizontal pass then vertical pass
- Still not optimal

Final implementation (integral image):

My approach:

1. Build an integral image using cumulative sums
2. Use NumPy's vectorized operations for parallel computation
3. Calculate box sums in constant time using the integral image formula

This reduces complexity from O(n×m×k²) to O(n×m), making it practical even for large kernel sizes.

```
# Build integral image once
cumsum = np.cumsum(np.cumsum(padded, axis=0), axis=1)

# Then O(1) box sum calculation using:
# sum = bottom_right - bottom_left - top_right + top_left
```

Resources that helped:

- "Summed-area tables" Wikipedia article
- NVIDIA's GPU Gems chapter on integral images
- Multiple Stack Overflow answers on vectorized NumPy operations

3. Performance Optimizations

My optimization process was iterative:

1. **Profiling**: Used cProfile to identify bottlenecks
2. **Memory layout**: Learned about C-contiguous vs Fortran-contiguous arrays from NumPy docs
3. **Vectorization**: Studied NumPy's broadcasting rules extensively

Key resources:

- "From Python to Numpy" online book
- Jake VanderPlas's "Python Data Science Handbook"
- NumPy's performance tips documentation

4. Web Application Development

Backend (FastAPI)

I followed several tutorials and best practices:

- FastAPI's official tutorial (fastapi.tiangolo.com)
- Real Python's FastAPI course concepts
- Miguel Grinberg's blog posts on async Python web development

Key decisions:

- Used Pydantic for automatic validation
- Implemented proper error handling with detailed logging
- Chose base64 encoding for simplicity (though I researched multipart uploads as an alternative)

Frontend Design Process

Here I leveraged modern tools and assistance:

UI/UX Research:

- Studied Glass Imaging's website for brand consistency
- Looked at modern image editing tools like Photoshop's web version

Implementation:

- Used Tailwind CSS documentation extensively
- **Utilized Claude/ChatGPT for**:
    - Generating the initial HTML structure
    - Creating the gradient backgrounds matching Glass Imaging's aesthetic
    - Implementing the animated stripe effects
    - Refining the responsive grid layouts
    - Converting from colorful to monochrome design when needed

The LLM assistance was particularly helpful for:

- Rapid prototyping of different design variations
- Generating complex CSS animations
- Ensuring cross-browser compatibility
- Creating accessible SVG icons

5. Testing and Debugging

My testing approach:

1. Created synthetic test images with known properties

2. Tested edge cases (empty images, single channel, huge kernels)
3. Performance tested with various image sizes

Tools and resources used:

- Python's unittest documentation
- pytest best practices guide
- Various computer vision testing datasets from Kaggle

Challenges and Problem-Solving Process

Challenge 1: Box Blur Performance

Initial problem: Kernel size 31 on a 1920x1080 image took several seconds.Initially, my naive implementation was too slow for large kernels. The integral image approach solved this elegantly.

My debugging process:

1. Added timing decorators to identify the bottleneck
2. Researched "fast box blur" on Google Scholar
3. Found integral image approach
4. Implemented and achieved ~100x speedup

Challenge 2: Frontend Design

Problem: Initial design looked outdated

Solution process:

1. Analyzed modern web applications
2. Used LLM to generate multiple design variations
3. Iteratively refined based on Glass Imaging's brand
4. Converted to dark theme for better visual cohesion

Challenge 3: Memory Management

Problem: Large images causing memory spikes

Research and solution:

1. Read about NumPy's memory management
2. Learned about view vs copy operations
3. Implemented proper array reuse and garbage collection

Learning Resources Summary

Books/Documentation:

- OpenCV-Python Official Tutorials
- NumPy User Guide and Reference
- FastAPI Documentation
- MDN Web Docs for JavaScript

Online Resources:

- Stack Overflow (especially for NumPy optimization)
- GeeksforGeeks (algorithm explanations)
- Real Python (Python best practices)
- CSS-Tricks (modern CSS techniques)

Academic Papers:

- Viola & Jones - "Rapid Object Detection" (integral images)
- Various SIGGRAPH papers on image processing

AI/LLM Assistance:

- Frontend HTML/CSS structure generation
- Complex animation implementations
- Design iterations and refinements
- Cross-browser compatibility checks

Running the Application

To run the application please refer README.md in the repository.

Code Quality

I prioritized:

- **Clean abstractions**: Clear separation of concerns
- **Comprehensive documentation**: Docstrings for all classes and methods
- **Type hints**: Used throughout for better IDE support
- **PEP 8 compliance**: Consistent code style

Reflection

This project challenged me to combine theoretical knowledge with practical implementation. The custom box blur requirement pushed me to dive deep into algorithm optimization, while the UI development let me explore modern web design trends.

Using LLMs for frontend development accelerated my workflow significantly, allowing me to focus more time on the core image processing algorithms. This hybrid approach - combining traditional research, documentation study, and AI assistance - represents how I approach modern software development.

This project was fun to build! I tried to keep things organized and lightweight while still hitting all the requirements. The pipeline is fully modular, the blur is custom-built, and the UI makes it easy to test things quickly.

If I had more time, I'd love to add:

- Live preview for each step
- Support for dragging steps to re-order the pipeline
- User sessions to save past pipelines
- Async background processing for big images