Image Based Math to LaTeX Converter

CAP 5415 Computer Vision Course Project

Sujan Katari
University of Central Florida
UCFID: 5543533
su410057@ucf.edu

Fall 2025

Abstract

Math equations appear everywhere in course materials and research content. Most students take screenshots of formulas from textbooks or online notes and then rewrite them manually in LaTeX. This takes time, and mistakes are easy to make, especially for long expressions with nested fractions and exponents. In this project we built a computer vision pipeline that converts a single line math image into LaTeX. The system uses classical OpenCV preprocessing tools to clean the input, then passes the processed image to a pretrained Pix2Tex model that predicts LaTeX directly. We exposed this pipeline through both a command line interface and a small Flask web demo with live MathJax rendering. We evaluated the system on screenshots from online sources and on textbook snippets. The main insight we gained is that careful, well chosen preprocessing improves recognition quality significantly, while overly aggressive thresholding or morphology can make a strong neural model perform much worse.

1 Introduction

Math equations are everywhere in computer science classes and in technical content. Homework problems, lecture slides, PDF textbooks, research papers, and online learning resources all heavily use mathematical notation. When students want to reuse these equations in their own documents or in tools such as LaTeX editors and symbolic solvers, they usually have to type the expression again by hand. For simple equations, this is manageable. For longer expressions with fractions, roots, and nested subscripts, it becomes slow and frustrating, and small mistakes are easy to introduce.

Screenshots are one of the most common ways students capture equations. For example, a student might screenshot part of a PDF page or an online problem explanation. These images look clean to the human eye, but they often have small artifacts. Common issues include light compression noise, slight rotation, inconsistent contrast between regions, and very thin symbols after zooming. Generic OCR tools can recognize plain text but do not understand the math structure at all. More specialized systems such as Pix2Tex are trained on rendered LaTeX images and understand the math structure, but they still expect inputs that are close to the training distribution.

The goal of our project is to bridge this gap in a practical way. We focused on a single line equation images drawn from digital sources, such as screenshots from PDFs or cropped textbook snippets. Given such an image, the system outputs a LaTeX string that the user can copy and paste. Instead of training an adequate model from scratch, we treated Pix2Tex (state of the art model) as a fixed black box and concentrate on the computer vision side of the problem. We build a preprocessing pipeline that makes real screenshots look more like the clean images that Pix2Tex expects.

There were several challenges we had to face along the way. On the vision side, we initially pushed the preprocessing too far with strong thresholding and morphology, which actually made the model perform worse. On the software side we ran into practical details such as model input type mismatches, port conflicts on macOS, and import problems after reorganizing the project into modules. These issues were not glamorous, but solving them taught me a lot about what it feels like to build an end to end vision system that people can actually use.

2 Method

This section describes how the system works. We will first give a high level overview of the pipeline, then go into the computer vision preprocessing, model inference, and the two interfaces.

2.1 System Overview

At a high level, the system has three stages:

1. Preprocess the input image to make the math symbols clearer and more consistent.
2. Run the Pix2Tex OCR model on the cleaned image and generate LaTeX.
3. Display the LaTeX and, in the web app, render it for quick visual checking.

Figure 1 shows the architecture of the whole system. The same core modules are shared between the command line and the web application.

2.2 Preprocessing

Screenshots and scanned textbook snippets do not look exactly like the synthetic images used to train Pix2Tex. Preprocessing helps reduce that gap. We tried several pipelines before settling on a lighter approach that preserves most of the grayscale information.

The final pipeline includes these steps:

- Grayscale conversion.
- Gaussian blur with a small kernel for mild noise reduction.
- CLAHE for local contrast enhancement.
- Optional Otsu thresholding for cases where binarization is beneficial.
- Deskewing using a Hough line based rotation estimate.
- Resizing to a consistent width while keeping aspect ratio.

The grayscale conversion reduces the image from three channels to one. We use the standard OpenCV conversion

$$Y = 0.299R + 0.587G + 0.114B \tag{1}$$

through `cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)`. For math images this is natural, since almost all information is in intensity changes, not color.

Next we applied a small Gaussian blur with a 3×3 kernel to reduce sensor noise and tiny compression artifacts. In OpenCV this is:

```
blurred = cv2.GaussianBlur(gray, (3, 3), 0)
```

The Gaussian kernel smooths high frequency noise while ideally keeping sharp symbol edges.

After that we applied CLAHE, which stands for contrast limited adaptive histogram equalization. Instead of stretching the histogram of the whole image globally, CLAHE divides it into tiles and equalizes each tile separately. This is useful for screenshots or scans where some regions are slightly darker or lighter than others. The code is:

```
clahe = cv2.createCLAHE(clipLimit=2.0,
                        tileGridSize=(8, 8))
enhanced = clahe.apply(blurred)
```

Here the clip limit controls how much local contrast can be amplified before it is clipped. In practice this fixed setting worked well across many test images.

Initially we went further and applied aggressive binarization and morphological operations on top of this. That turned out to be a mistake. A combination of adaptive thresholding and opening with a 2×2 kernel erased thin symbols and serifs. The model then produced nonsensical LaTeX output. After seeing these failures we changed the design. In the final version binarization is optional and controlled by an `aggressive` flag for debugging. The default pipeline stops at the CLAHE enhanced grayscale image, which preserves more of the original signal.

To handle small rotations we estimated the skew angle with a Hough line transform. We first created a temporary binary image using Otsu thresholding, then call `cv2.HoughLines` to detect long lines. Each detected line gives me an angle. We then computed the median angle of all such lines and use that as the estimated rotation. If the magnitude of the angle is larger than one degree, we rotated the image around its center with `cv2.getRotationMatrix2D` and `cv2.warpAffine`. The border mode is set to `BORDER_REPLICATE` so that empty areas are filled by copying the nearest edge pixels rather than with black bands.

Finally we resized the image to a fixed target width. The original pipeline used a width of 1500 pixels, which was overkill and sometimes produced interpolation artifacts. In the final version we used a target width of 800 pixels and scale the height to preserve aspect ratio. When downsampling we used `INTER_AREA` interpolation, which is known to behave well for shrinking images. If the input is already small we skip resizing.

Figure 2 shows an example of how the preprocessing changes the appearance of a real screenshot.

A simplified version of the preprocessing function is shown in Listing 1. The actual code includes additional checks and logging for debugging.

Listing 1: Simplified preprocessing function.

```
def preprocess_image(input_path: str,
                     output_path: str,
                     aggressive: bool = False) ->
                         str:
    img = cv2.imread(input_path)
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

    blurred = cv2.GaussianBlur(gray, (3, 3), 0)

    clahe = cv2.createCLAHE(
        clipLimit=2.0, tileGridSize=(8, 8)
    )
    enhanced = clahe.apply(blurred)

    if aggressive:
        _, enhanced = cv2.threshold(
            enhanced, 0, 255,
            cv2.THRESH_BINARY + cv2.THRESH_OTSU
        )

    angle = estimate_skew_angle(enhanced)
    if abs(angle) > 1.0:
        enhanced = rotate_image(enhanced, angle)

    h, w = enhanced.shape[:2]
    target_w = 800
    if w > target_w:
        scale = target_w / float(w)
        target_h = int(h * scale)
        enhanced = cv2.resize(
            enhanced, (target_w, target_h),
            interpolation=cv2.INTER_AREA
        )
```

```
    cv2.imwrite(output_path, enhanced)
    return output_path
```

2.3 Model Inference with Pix2Tex

For the OCR step we use the Pix2Tex LatexOCR model from the open source project by Blecher. The model is a vision encoder and transformer decoder that maps an input image of an equation to a LaTeX token sequence. We did not modify or train this model. Instead we wrapped it in a small Python class that handles image loading and prediction.

At first we tried to pass the image file path directly into the model call. This produced a confusing error:

```
'str' object has no attribute 'copy'
```

After reading the source code we realized that the model expects a PIL Image object, not a file path string. The fix was to open the image using Pillow and pass the resulting object instead. Listing 2 shows the final model wrapper.

Listing 2: Pix2Tex model wrapper.

```
from pix2tex.cli import LatexOCR
from PIL import Image

class Pix2TexModel:
    def __init__(self):
        # Load once at startup
        self.model = LatexOCR()

    def predict(self, image_path: str) -> str:
        img = Image.open(image_path).convert("RGB"
            )
        latex = self.model(img)
        return latex
```

Loading the model takes roughly five to ten seconds on our laptop but this only happens once, when the program starts. After that each prediction takes around one to three seconds depending on the complexity of the equation.

2.4 Interfaces

I provided two ways to use the system. A simple command line interface and a web based interface.

The command line script lets the user run:

```
python main.py path/to/image.png
```

The script calls the preprocessing function, runs the model, prints the LaTeX to the terminal, and writes the LaTeX to an `output.tex` file. This mode is convenient for batch processing or for users who live in the terminal.

The web application is built with Flask. It exposes a root route that serves a single page with an upload box. The user can drag and drop an image, the browser sends it to the backend, the server saves it to a temporary file, runs preprocessing and inference, and sends back a JSON response containing the LaTeX. The frontend then displays both the raw LaTeX string and a rendered version using MathJax. Temporary files are cleaned up after each request.

Figure 3 shows a screenshot of the web interface.

On macOS we ran into a small but real usability issue. Flask defaults to port 5000 which is sometimes used by AirPlay Receiver. When that happened we got an address in use error even though we thought no other servers were running. The fix was to run the app on port 5001 instead. It is a small change but it is the kind of detail that comes up in real projects.

3 Results

In this section we describe what types of images we used to test the system, how we evaluated it, and what worked and what did not.

3.1 Datasets and Test Cases

I did not create a large labeled dataset because the focus of the project is on pipeline design rather than model training. Instead we collected a small but diverse set of example images that cover typical student use cases.

I grouped the test images into three categories:

- Screenshots of rendered LaTeX from PDF textbooks and online LaTeX editors.
- Cropped equations from scanned textbook pages.
- Miscellaneous online math content such as formulas from learning websites.

For each image we ran two versions of the pipeline. One that sent the raw image directly to Pix2Tex, and one that used the full preprocessing pipeline first. We then compared the predicted LaTeX to a manually written ground truth. For this project we did not compute a full symbol error rate. Instead we focused on qualitative comparisons and on whether the LaTeX compiled and visually matched the original equation.

3.2 Qualitative Examples

Figure 4 shows an example from the easiest category which is rendered LaTeX screenshots. The input is a screenshot of a formula that was originally typeset in LaTeX. These examples are close to the training distribution of Pix2Tex. As expected, the system recovers the exact LaTeX in many of these cases, both with and without preprocessing. Preprocessing still helps a bit by standardizing size and contrast, but the gains are smaller here.

Figure 5 shows a more realistic and slightly harder case. Here the input comes from a scanned textbook page or a photo of a printed page. There is mild blur and some local variation in brightness. In early versions of the pipeline that used strong thresholding and morphology, the predicted LaTeX was completely wrong. After switching to the gentler CLAHE based preprocessing, the system produced mostly correct LaTeX for

many such examples. There are still occasional mistakes in subscripts or small spacing details, but the overall structure is usually right.

A final set of examples comes from random online math content such as formulas embedded in web pages. These are somewhere in between the previous two cases. They can have different fonts, antialiasing styles, or colored backgrounds. The pipeline still works reasonably well here, but we observed that the model sometimes confuses certain symbols such as minus signs and long dashes, or merges close neighbors if the resolution is too low. These failure cases match what we would expect from a model that was trained on a somewhat different distribution.

3.3 Timing and Performance

All experiments were run on a CPU only laptop. The Pix2Tex model is fairly large, around several hundred megabytes, and uses PyTorch under the hood. Loading the model takes about five to ten seconds at startup. Once loaded, inference takes roughly one to three seconds per image for the kind of resolutions we used. The preprocessing pipeline is very fast in comparison. Grayscale conversion, blurring, CLAHE, Hough based angle estimation, and resizing all together usually take less than 100 milliseconds.

This confirms that the overhead of the computer vision pipeline is negligible compared to the cost of running the neural model. That made it easy to experiment with different preprocessing choices without worrying too much about latency.

3.4 Effect of Preprocessing

The main experiments we ran were comparisons between raw inputs and preprocessed inputs. For clean rendered screenshots the difference is modest. The model already performs well, and preprocessing mainly stabilizes minor edge cases.

For scanned textbooks and lower quality screenshots, preprocessing makes a clear difference. CLAHE often turns faint symbols into clearly visible strokes. Deskewing helps the model when the equation is tilted, because many encoders are sensitive to the orientation of features. Resizing to a consistent width also helps, since very large inputs are implicitly downscaled inside the model, where we have less control over the interpolation.

At the same time, the experiments also made it clear that more processing is not always better. Our initial attempt with aggressive thresholding and morphological cleaning looked nice in binary form to our eyes, but it destroyed thin parts of symbols and confused the model. The lesson we took from this is that it is important to think about what the downstream model was actually trained on. In this case, preserving grayscale information and subtle antialiasing matches the training distribution better than hard binarization.

4 Conclusion

In this project we built an image to LaTeX converter that combines classical computer vision with a pretrained deep learning model. The final system can take a single line equation image from a screenshot or a textbook snippet and produce a LaTeX string that is often correct or at least very close. The system is usable from both the command line and a small web interface, which makes it practical for everyday use.

From a computer vision perspective, the project gave me hands on experience with grayscale conversion, Gaussian blur, CLAHE, Otsu thresholding, Hough transforms, and affine rotation. More importantly, it showed me how these basic building blocks behave in a real end to end pipeline rather than in isolated textbook examples. We also learned that classical CV still matters a lot even when you rely on a large neural model. Feeding the model with images that are closer to its training distribution is often more helpful than trying to add more layers on top.

From a software engineering perspective, we ran into a range of small but real issues. These included port conflicts on macOS, incorrect assumptions about model input types, and import path problems after restructuring the code into modules. None of these issues are scientifically exciting, but they are part of building working systems. Solving them forced me to read documentation carefully, inspect stack traces, and step through small experiments, which we think is valuable practice.

If we had more time, we would like to extend the project in at least two directions. First, we would add multi line support by detecting and grouping equation lines within a larger region, then ordering them correctly. Second, we would explore fine tuning a Pix2Tex style model on a mix of clean rendered LaTeX and more realistic screenshot or scan data. That might reduce some of the remaining errors on low quality inputs. Overall, we am happy with how much we was able to accomplish in one semester, and we now have a much clearer intuition for how computer vision preprocessing and deep learning based OCR can work together.

5 My Contribution

This project was completed as a group. We divided the work into several parts, such as model integration, web interface, testing, and computer vision preprocessing. My main responsibility was the preprocessing pipeline and making sure the images we sent to the model were as clean and consistent as possible.

I designed and implemented the preprocessing steps in OpenCV. This included grayscale conversion, Gaussian blur, CLAHE based local contrast enhancement, optional Otsu thresholding, skew estimation with a Hough based method, affine rotation for deskewing, and resizing to a stable target width. We tried different parameter settings and several alternate pipelines, including more aggressive binarization and

morphology, and then compared how each version affected the Pix2Tex predictions.

I also focused on data quality for the model. That meant collecting representative screenshots and textbook snippets, visually inspecting intermediate outputs at each preprocessing stage, and checking that resolution, aspect ratio, and interpolation choices did not accidentally degrade important symbol details. Based on these experiments we tuned the final pipeline to be gentle enough to preserve strokes and structure while still helping the model deal with noise, uneven lighting, and small rotations.

References

[1] G. Bradski. The OpenCV Library. Dr. Dobb's Journal of Software Tools, 2000.

[2] L. Blecher. pix2tex: Using a Vision Encoder Decoder to predict LaTeX. https://github.com/lukas-blecher/LaTeX-OCR, 2021.

[3] A. Ronacher. Flask Web Framework. https://flask.palletsprojects.com/.

[4] D. Cervone. MathJax: A Platform for Mathematics on the Web. https://www.mathjax.org/.

[5] N. Otsu. A Threshold Selection Method from Gray Level Histograms. IEEE Transactions on Systems, Man, and Cybernetics, 1979.

[6] P. V. C. Hough. Method and Means for Recognizing Complex Patterns. United States Patent 3,069,654, 1962.

[7] K. Zuiderveld. Contrast Limited Adaptive Histogram Equalization. In Graphics Gems IV, Academic Press, 1994.

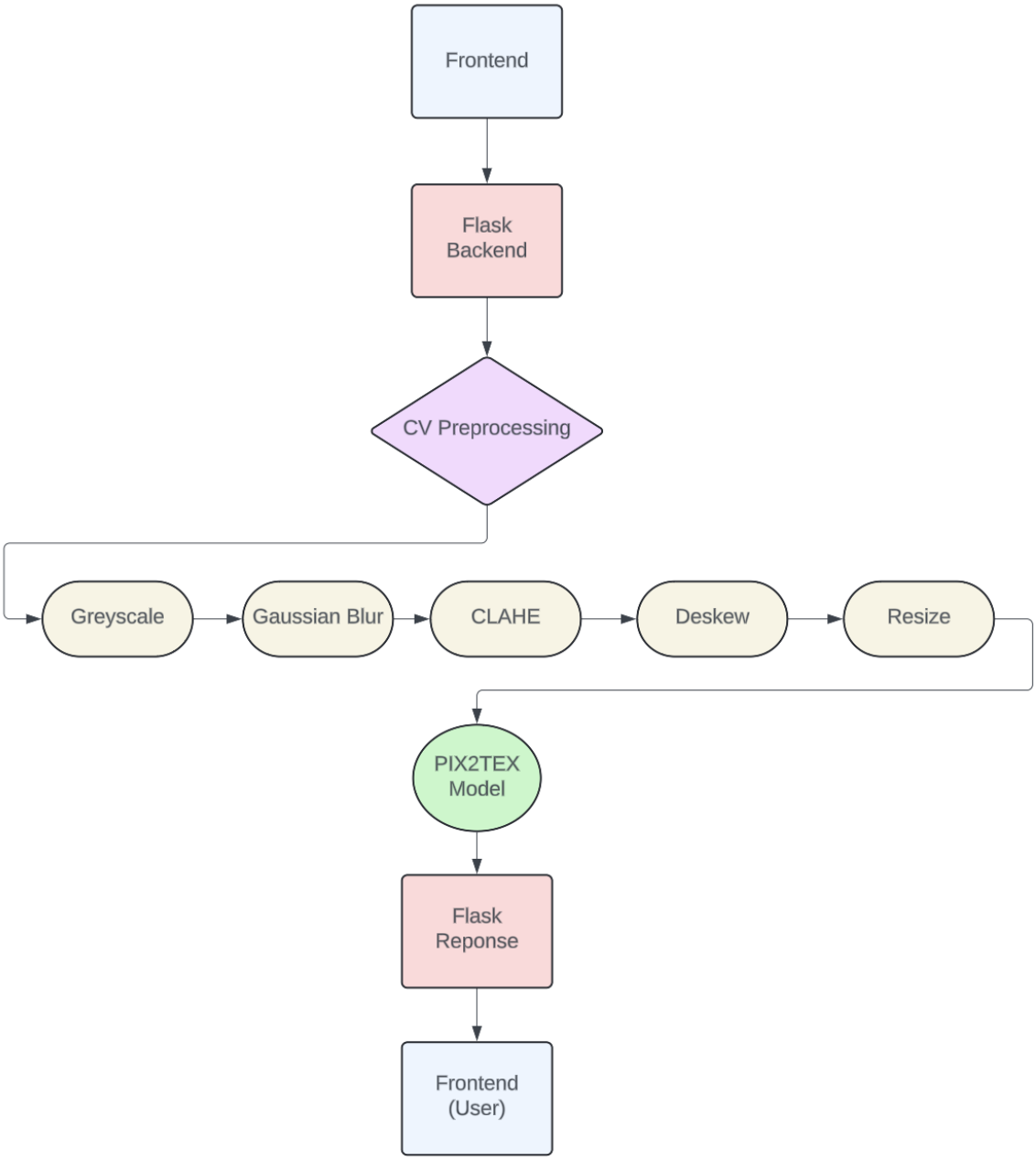

Figure 1: System architecture. Both the command line interface and the Flask web app call into the same preprocessing pipeline written in OpenCV, which then feeds the cleaned image into the Pix2Tex LatexOCR model. The model returns a LaTeX string, which is either written to a file or rendered in the browser using MathJax.

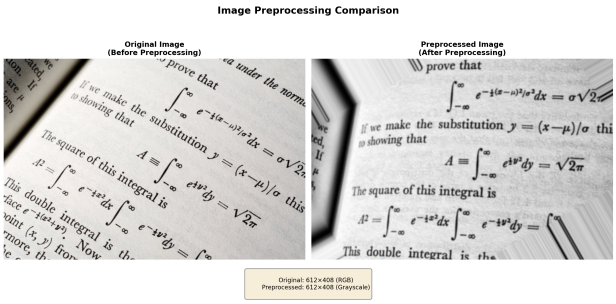

Figure 2: Preprocessing example. The top part shows a raw screenshot of a math expression. The bottom part shows the same region after grayscale conversion, light blurring, CLAHE based contrast enhancement, deskewing, and resizing. Symbols become sharper and more uniform in brightness which makes the OCR model more stable.

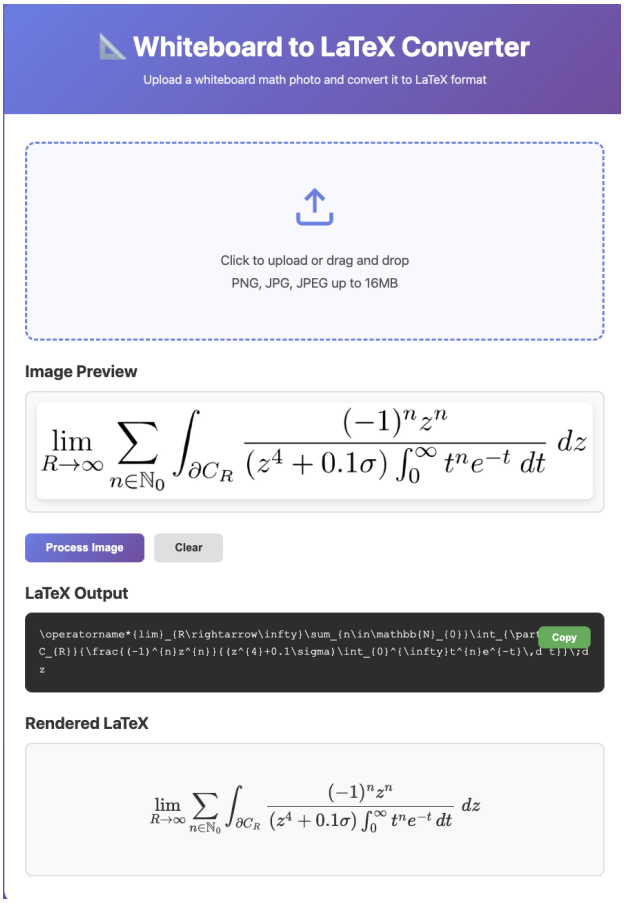

Figure 3: Web interface for the Image to LaTeX converter. Users can upload a screenshot, see the predicted LaTeX, and view the rendered equation in the browser. This makes it easy to verify that the model understood the expression correctly.

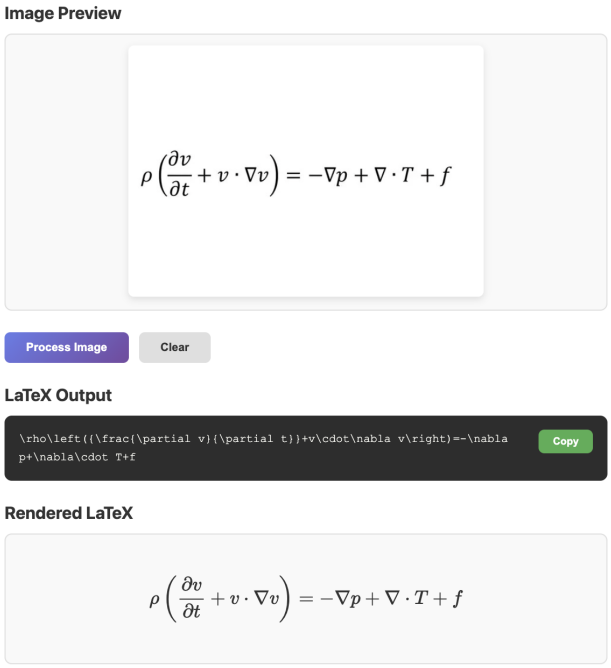

Figure 4: Recognition example from a rendered LaTeX screenshot. The top portion shows the equation as it appears in the digital source. The bottom portion shows the LaTeX predicted by the system. For clean rendered equations the model often recovers the exact original expression.

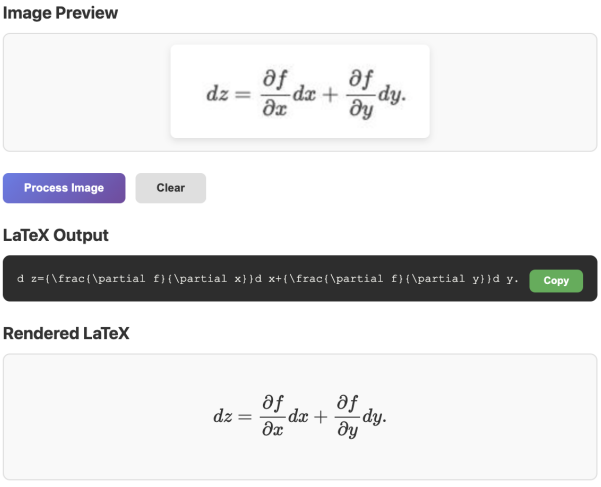

Figure 5: Recognition example from a textbook snippet. The input comes from a scanned or photographed textbook page. After preprocessing the system is often able to recover the structure of the equation and produce usable LaTeX, although small errors in fine details can still occur.