Rendering Implicit Surfaces Defined by LaTeX Expressions

Robert Boerwinkle

Introduction

Functions of one variable, or 2D functions, are trivial to graph (https://www.desmos.com/calculator). Functions of two variables are a bit tricker, but can be done (https://www.geogebra.org/3d). Rendering surfaces implied by functions of three variables is rare. This is unfortunate for students of higher level calculus and other maths. In a previous paper (Favorite & Boerwinkle, 2022), a method of rendering these surfaces was developed. This paper outlines the process of making it accessible to those unfamiliar with Python.

An implicit surface is one defined as the set of points where a function of three variables (e.g. $f(x, y, z) = \sin x * \sin y * \sin z$, where the fourth dimension is the output variable) is equal to zero. Commonly, the function can be set to a different value, called the threshold. Among software that renders these surfaces, it is common to limit the functions to ones which can be solved for $z = f(x, y)$ (making it a function of two variables / 3D function), or to convert the surface to a triangle mesh.

The implementation in the previous paper is written in Python. It accepts a fourier series or 3D discrete data which gets converted into a fourier series. The fourier series then acts as a function of three variables, and the implicit surfaces are rendered. All rendering is done in a single thread. It outputs a simple NumPy array of RGB values.

Most documents support low level math symbols such as +, -, *, /, etc. More complex equations, however, cannot be written in plain text. LaTeX is the solution to this problem. It takes plain text input and renders it in properly typeset math script (American Mathematical Society, 1999). This is a common backend tool, even if it does not directly face the user. If one were to copy equations out of Desmos, for example, it would be in LaTeX format.

In order to display images and text to a screen, software has to go through many layers. Luckily, these layers are usually hidden behind high level toolkits. These toolkits provide commonly used features in a simple interface. They handle all the background of communicating with the windowing server. “Tk” is a windowing toolkit with libraries for many languages, including Python. It is cross-platform and comes with the Python standard library as tkinter. This means that programs written in Tk can be graphically complex and extremely portable.

With some streamlining and optimization of the existing implementation, it can be made to render user inputted functions in real time. From there, a Graphical User Interface (GUI) makes it fully accessible. The resulting application should be portable, intuitive, aesthetically pleasing, efficient, and provide a unique view into the structure of the function. This paves the way for visualization of 3D fields as well as further development of the renderer.

Underlying Structure

The render previously only took discrete, 3D data or a fourier series. In order to interact with the renderer, a custom wrapper object was made to contain the user script and mimic the fourier series. Whenever the fourier series was called to be evaluated, execution was then re-routed to the user inputted function.

User input can be simplified to one of two actions: inputing a function or requesting a frame. Following Figure 1, these correspond to 'A' and '1'. Whenever a new Python function is inputed, minimal processing is required. If the function is in LaTeX, it has to be sent to the LaTeX converter, which is discussed in a later section (LaTeX Input).

Most of the interaction takes place in the image requesting loop. The GUI will request a new image every time the function is changed, the camera moves, or any other setting changes (resolution, field of view, etc.). When an image is being rendered, the function needs to be evaluated at several points. Since Python is an interpreted language, user code can be run directly via the built-in `eval` function. The inputted code has access to the NumPy library (aliased to `np`) and three variables: `X`, `Y`, and `Z`. These are each composed of a 1D NumPy array of the corresponding component in a list of points to be evaluated. Because of NumPy's syntaxing, these can be treated as if they were single numbers. This makes the code mimic math script.

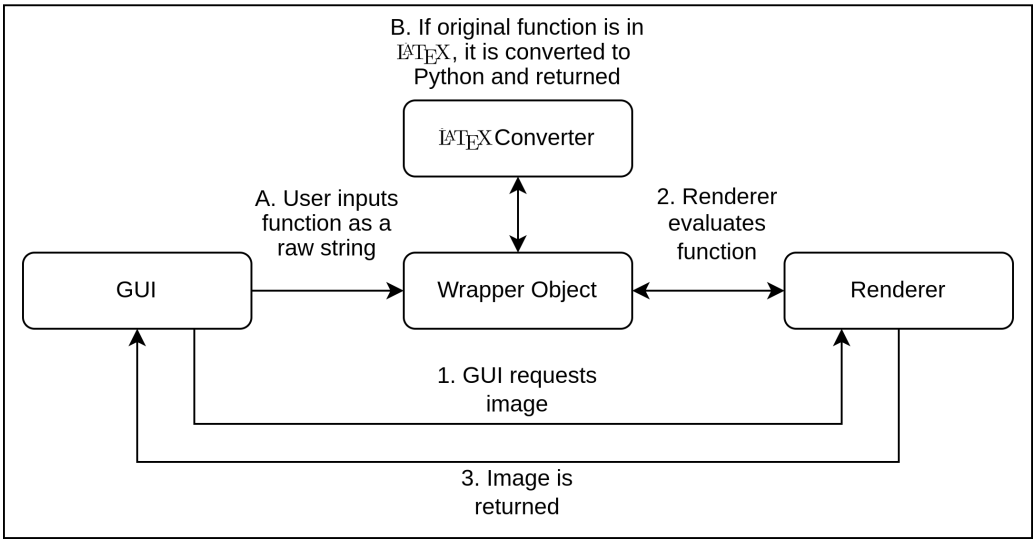

Figure 1: Flow of execution

With the `eval` function comes a lot of flexibility, and it allows very direct interaction with the underlying algorithm. Even very non-standard functions can be rendered. Unfortunately, flexibility means vulnerability, and `eval` is particularly prone to code injection attacks (Heymann, Kohnfelder, & Miller, 2022). This is not particularly concerning, as this software has no mechanism of sharing scripts. If it became a concern for a user, the software could be run with fewer permissions.

While Python is an interpreted language, it still has to be compiled. It does not compile to machine code, it compiles to bytecode, which is somewhere between source and machine code (Bagheri, 2020). The original implementation directly ran `eval` on the user code each time it was called. This compiles the code each time it needs to run. By compiling it beforehand, an almost 10% speed increase was accomplished.

Tk Window Structure

The basic graphical structure of the program consists of a simple diagnostic pane, the view port, and an input box. The diagnostic pane displays information about the camera angle. This is also where error messages are displayed. Errors are annotated with what component threw them (LaTeX converter or renderer, Python interpreter, the renderer itself, etc.). The view port is where the rendered images are displayed. The input box has a text input and a selector to switch between LaTeX and Python. If LaTeX is chosen, an additional pane for rendering the LaTeX appears.

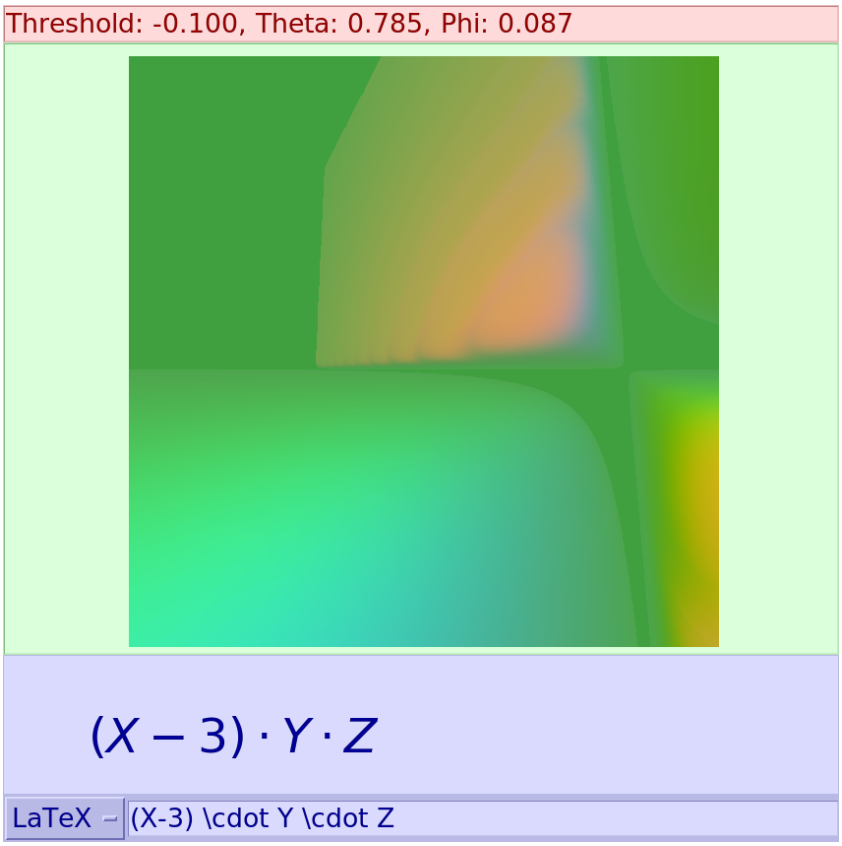

Figure 2: An early window, annotated: top, red - diagnostic pane; middle, green - view port; bottom, blue - input box

When camera translation became available, a position indicator was added. As the number of settings increased, there became a need for a separate settings window. Settings such as the field of view, numerical control of all the camera position parameters, and resolution are accessed via the gear button in the lower left. A gallery development snapshots can be found at the end.

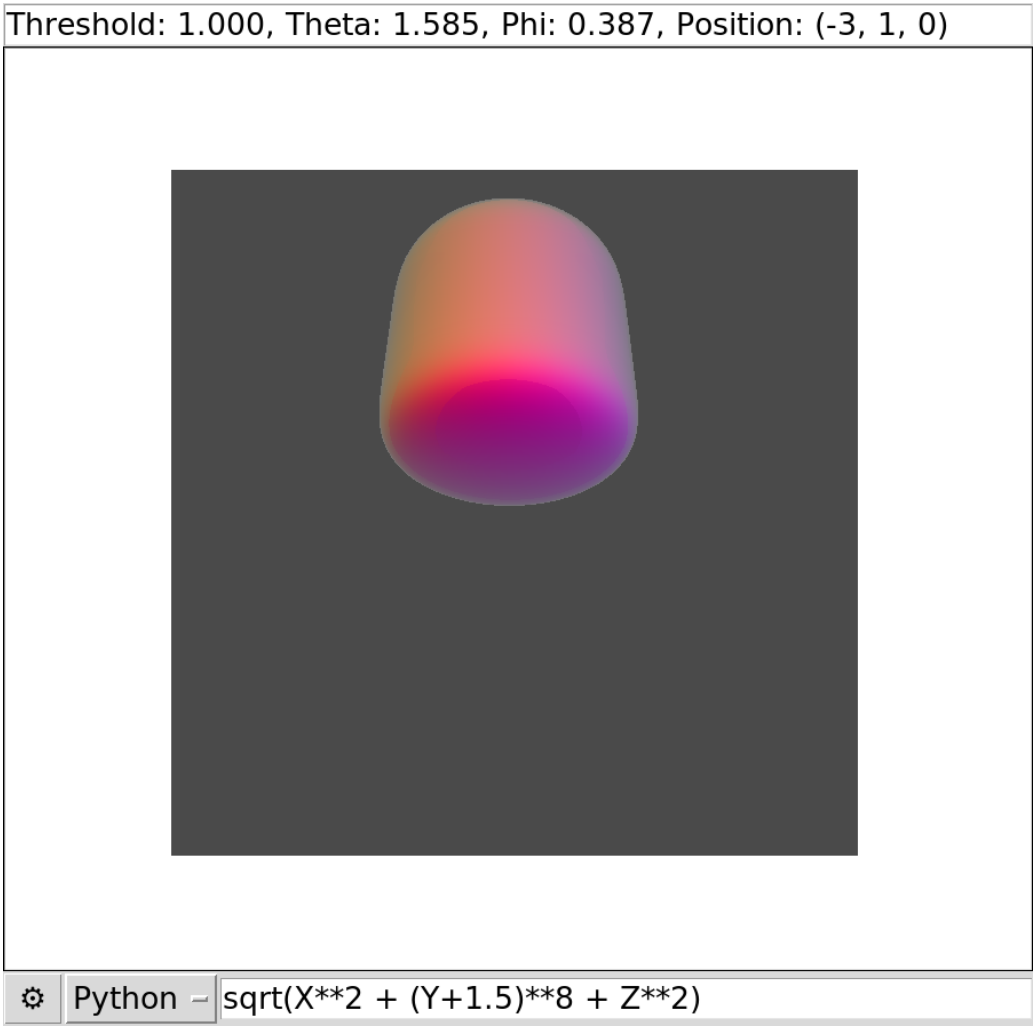

Figure 3: A cylinder in the latest version of the software

LaTeX Input

For more traditional math writeup and typesetting, the go to language is LaTeX. Unfortunately, LaTeX is a typesetting language, so it doesn't have as strict a mathematical meaning as Python script. There is no standard implementation for evaluating a LaTeX expression. There is, however, work done to convert LaTeX expressions into Symbolic Python (SymPy) (Trollbäck et al., 2016). The converter runs on ANTLR, a language parser (https://www.antlr.org/). It breaks down the LaTeX into an expression tree, and then tries to find an equivalent Python function. The SymPy script can then be evaluated.

When developing LaTeX expressions, it is very important to be able to see the standard graphical output. MatPlotLib, a Python library for creating graphs, has a LaTeX interpreter which outputs images. This is normally used to display miscellaneous titles, but can be hijacked to render any expression. This can be displayed alongside the expression and updated in real time.

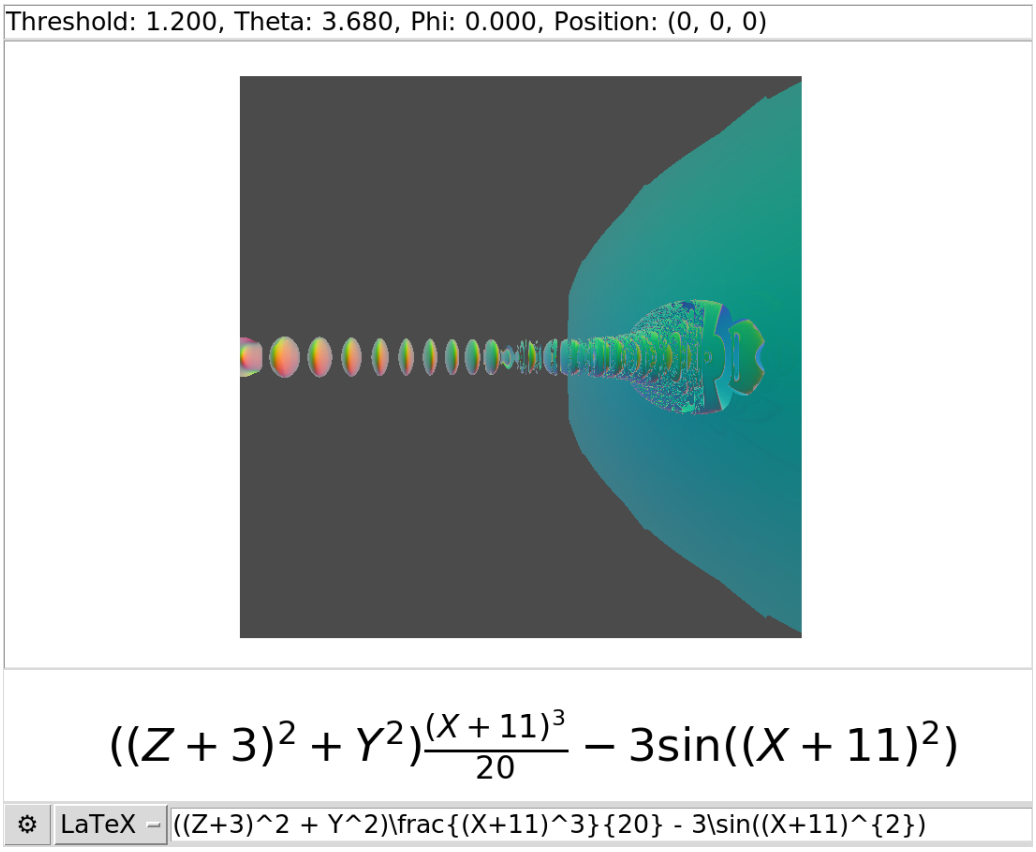

Figure 4: A complex LaTeX rendering

Conclusion

The software accomplished all the goals set forth: It successfully rendered complex LaTeX equations, provided plenty of graphical feedback, and allowed the viewing of 4D functions (or 'functions of three variables'). The conversion from LaTeX to Python was messy and relied on a 3rd party library. It could potentially be rewritten in the future. The other area for improvement is performance. Minor improvements were implemented, but, as stated in the previous paper, rewriting the renderer in a lower level language (C++ is a prime candidate) would increase performance greatly.

Several bugs were removed from the original renderer, but additional testing is certainly required. This GUI should provide a much easier debugging interface. The original interface is, of course, still available and useful. Automated renderings are still only available with custom scripts. This is a big step, but only a step, in the advancement of this rendering technique.

Gallery

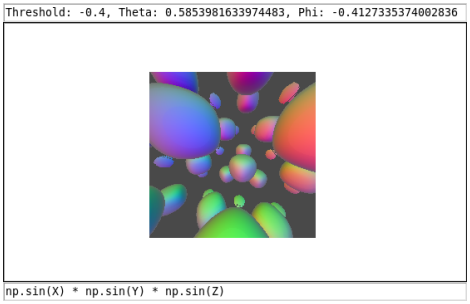

Figure 5: This version only allowed Python

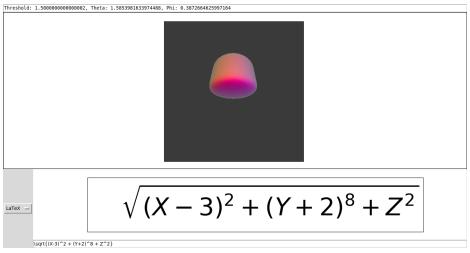

Figure 6: This version had some placement issues

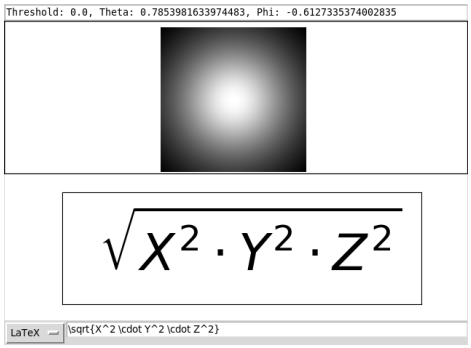

Figure 7: This is the last version with the secondary LaTeX box

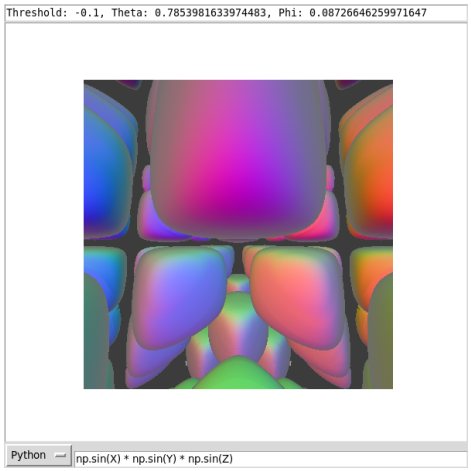

Figure 8: This is only missing some string formatting

References

All images created by the researcher.

American Mathematical Society, LaTeX Project (1999). *User's Guide for the amsmath package.* The LaTeX Project. Retrieved from https://www.latex-project.org/help/documentation/amsldoc.pdf

Bagheri, R. (2020). *Understanding Python Bytecode.* Towards Data Science. Retrieved from https://towardsdatascience.com/understanding-python-bytecode-e7edaae8734d

Favorite, J., & Boerwinkle, R. (2022). *Rendering Implicit Surfaces Using Multidimensional Fourier Series.* GitHub. Retrieved from https://github.com/IsaacLizardKing/3D-Fourier-Transform-Rendering/blob/main/SciPaper.pdf

Heymann, E., Kohnfelder, L., & Miller, B. (2022) Chapter 3.8.3: Code Injections. In *Introduction to Software Security.* University of Wisconsin–Madison. Retrieved from https://research.cs.wisc.edu/mist/SoftwareSecurityCourse/Chapters/3_8_3-Code-Injections.pdf

Trollbäck, A., sindytn, Jimenez, N., & Deep, A. (2016). *latex2sympy.* GitHub. Retrieved from https://github.com/augustt198/latex2sympy