Rendering Implicit Surfaces Using Multidimensional Fourier Series

Joshua Favorite, Robert Boerwinkle

Introduction

This paper proposes a new method for rendering 3D data relying on a Discrete Fourier Transform (DFT) of the data. Marching Cubes, the leading method of rendering 3D data, is an aging algorithm. This method was developed as a final step in finding a way to optimize the Marching Cubes algorithm for real time applications.

The Marching Cubes algorithm takes discrete data and generates a triangular mesh. This was originally used to create 3D visualizations of MRI scans. It has seen wider applications such as mesh generation in 3D modeling software as well as terrain generation in the gaming industry. However, Marching Cubes loses its efficacy when attempting to triangulate continuous data that needs to be calculated on the fly. This is due to the fact that a majority of calculations done by the algorithm will not contribute to the triangulation of the data. The Marching Cubes algorithm best handles pre-calculated discrete data of limited scope. For dynamic resolution of renders, the data must be continuous. In the case of rendering arbitrary data, the algorithm is stuck with the resolution of the input.

A fourier transform turns a value function into a frequency function. The domain is the frequency, and the output value is the magnitude of the frequency in the original function. This can be done to discrete data (a DFT). There is an algorithm, called a fast fourier transform (FFT), which can compute the fourier transform. There are numerous implementations of this algorithm including ones from NumPy, SciPy, MATLAB, and others. It returns an array of the same size and shape as the data. It is composed of complex numbers. There is a commonly implemented inverse FFT algorithm which returns the original (discrete) data. An inverse transform does not have to be discrete though. A continuous but slow inverse exists, but there are no readily available implementations of this function.

Root finding algorithms use approximations to find the roots of nonlinear equations. The roots of equations such as multidimensional fourier transforms cannot be solved in closed form, thus a root finding algorithm must be employed. The Newton approximation method assumes the equation is linear at the current point and uses that assumption to approach the nearest root of the equation.

The goal of this project is to render fourier transforms of discrete arbitrary data using root finding algorithms. Converting discrete arbitrary data into continuous data using a fourier transform and using Newton approximations to find the roots of that data will allow for the rendering of these complex implicit surfaces with dynamic resolution.

1 Camera Setup

The camera contains a 2D array of 3D vectors to represent rays, and the coordinates of the camera. For initialization, a grid of 3D coordinates is defined as being spaced evenly on the X and Y planes. The Z coordinates are uniform and represent the distance from the camera to the middle of the grid. As Z increases, the field of view decreases. Z is calculated with this equation based on the desired field of view and the height and width of the vector grid:

$$z = \sqrt{\left(\frac{W}{2}\right)^2 + \left(\frac{H}{2}\right)^2 + \sqrt{\left(\frac{W}{2}\right)^2 + \left(\frac{H}{2}\right)^2}} * \cos\frac{FOV}{2}$$

The rotation matrix currently only makes use of spherical coordinates, but Euler angles could be implemented for more freedom in rendering. The rotation matrix is the product of the following:

$$\begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & -\sin\phi \\ 0 & \sin\phi & \cos\phi \end{bmatrix}$$

After the camera is rotated, the camera vectors are normalized. This just uses a spherical coordinate conversion to get the angles of each vector; then, they are pushed to the sphere by converting them back to Cartesian coordinates by factoring ρ out of the equation. To test that the coordinates were projecting onto the sphere correctly, spheres were rendered in closed form. There were initially some issue with the rotations, as one version of the program stored the camera vectors in spherical coordinates, causing the vectors to converge at the poles when ϕ was changed. The camera performed well on these tests, but perhaps not as well as a camera that could use GPU instruction. Rendering a moving sphere at a resolution of 400 x 400 rays had a maximum framerate of 15fps.

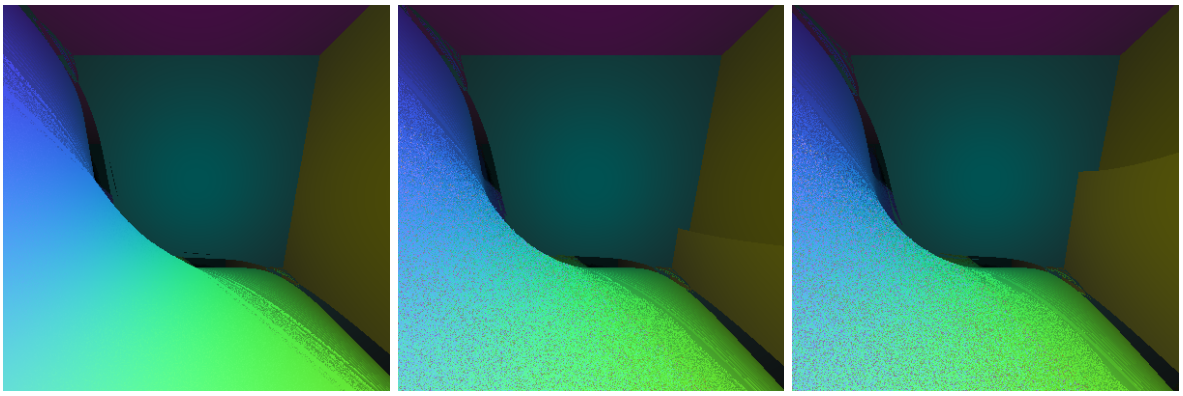
2 Newton Approximation

The Newton approximation is just the function $x_{n+1} = x_n - \frac{f(x)}{f'(x)}$. Getting the slope of a normal sine function is rather simple, and there are plenty of algorithms that can take derivatives of functions. However, implementing such an algorithm was beyond the scope of the project, so derivatives were either hard coded in for specific functions, or computed using the definition of derivative ($\frac{f(x+\Delta x)-f(x)}{\Delta x}$) with a hard coded Δ. This works only for finding the nearest root. If the origin is near local minima or maxima that do not intersect the X axis, it loses its efficacy. Additionally, if the slope of the function approaches 0 and the value of the function does not, the newton approximation function will step into infinity. To prevent this, piecewise sigmoid function was implemented to confine the magnitude of the step function. Where the step function was greater than 1, this function was used instead.

$$\frac{f(x) * \frac{f'(x)}{|f'(x)|}}{e^{|f'(x)|}}$$

For multiple dimensions, the step function is treated as a magnitude. The coordinates will move in proportion to the magnitude of each corresponding vector axis. Using this procedure, the eggcrate function ($w = \sin x \sin y \sin z$) was rendered.

$$\begin{aligned} \Delta w &= \frac{f(x,y,z)}{f'(x,y,z)} \\ x &= x - \Delta w * v_x \\ y &= y - \Delta w * v_y \\ z &= z - \Delta w * v_z \end{aligned}$$



(a) The camera's origin is at $x = \frac{\pi}{2}, y = \frac{\pi}{2}, z = \frac{\pi}{2}$.

(b) The camera has shifted left on the X axis by $\frac{2\pi}{180}$. A second surface appears in front of the yellow surface.

(c) The initial yellow surface continues moving right as the new surface covers it and begins moving left toward the camera.

Figure 1: Early renders of the eggcrate function, $(sinxsinysinz = 0)$.

The initial results are puzzling. The slope to the left of the camera is omnipresent. In theory, the camera should be at the absolute maximum of the function, but the slope cannot be explained under this assumption. The camera continued to move away from the slope until passing through the wall on the right, after which the camera appeared to move to the left. Attempts to occlude the slope by combining traditional ray tracing techniques with the Newton approximation only generated more anomalous results. The depth field indicates that the points on the slope are extremely close to the camera, meaning that there is an implicit surface near the peak of the function. This is not the case, as later renders revealed that the surface of the function at 0 was a perfect cube.

3 Monotonically Increasing Newton Approximation

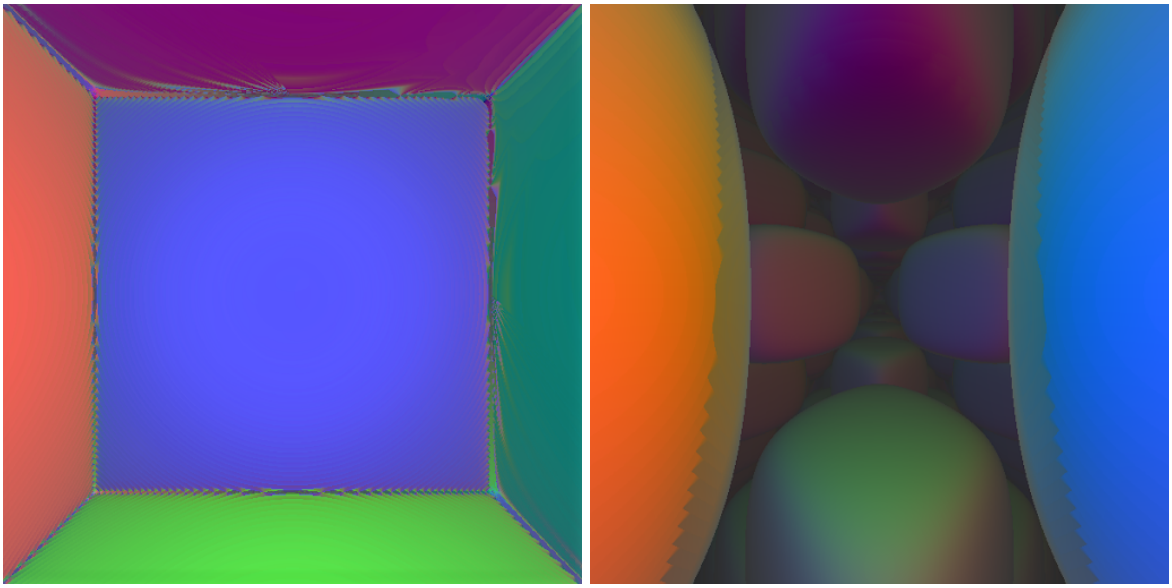
The classical Newton approximation method did not guarantee that the surfaces it found were legitimate implicit surfaces, nor did it guarantee that those surfaces would be in front of the camera. To remedy this, the step function was modified to only allow steps forward, imposing a minimum and maximum step size and taking the absolute value of the step magnitude. The rays continue to march until the sign of the function at the current position does not equal the sign of the function at the origin.

This method ensures that all surfaces are in front of the viewer, and that points will never march into infinity. Additionally, a limit was imposed on how far the points could march from the origin, as the program could render infinite distance without every ray intersecting a surface.

Immediately following the implementation of these procedures, the rendering function stopped having issues with the anomalous slope produced by the classic Newton approximation. However, the approximations appeared to struggle in specific situations.

The monotonically increasing newton approximation function proved much more reliable in finding the desired roots of the egg crate equation.

As for the artifacting in figure 2, nonadjacent half-cycles have equivalent signs. This allows rays to pass through corners into nonadjacent half-cycles. In figure 2a, in the absence of sign changes, the rays crossing into nonadjacent half-cycles continue to march until passing into adjacent half-cycles, hence the fractal-esque behavior near the corners of the cube. In figure 2b, there is a hard transition between the colors on the backsides of the cuboids. There are no explanations for this other than the imprecise nature of imposing a lower limit onto the step magnitude.



(a) Threshold of 0; The hard corners of the cube show fractal-esque behavior.

(b) Threshold of -0.1; The cuboids show hard transition between the colors on their adjacent sides.

Figure 2: Renders of the eggcrate ($w = \sin x \sin y \sin z$) using the monotonically increasing Newton approximation without using classical Newton approximations for refining surfaces.

4 Fourier Transform

In order to approximate the intercepts, a function for finding the value at a given point needs to be established. A discrete fourier transform (DFT) can be used to interpolate values from discrete data. Finding the DFT is done with NumPy's `fft.fftn` function. It returns an array of complex numbers that is the same size and shape of the original data. These complex numbers represent sinusoidal functions. The formula for the inverse fourier transform, where F is the transform, is:

$$f(x,y,z) = \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} \sum_{w=0}^{O-1} F(u,v,w) e^{j2\pi\left(\frac{ux}{M} + \frac{vy}{N} + \frac{wz}{O}\right)} \tag{1}$$

The implementation in python is very similar to the equation, except $i2\pi$ was distributed and the division by M, N, and O was precomputed. The original code could only evaluate one point at a time. This required looping through the camera vectors to evaluate them.

NumPy is more efficient at doing operations on large arrays than python looping, so the function was modified to accept an array of points. However, since complex numbers occupy 128 bits each, and evaluating the camera vectors in parallel requires the algorithm to broadcast the transform data onto the vector data, the memory usage could exceed 16GB quickly. A 32 x 32 x 32 signed distance field of an octahedron was attempted, using a camera with a resolution of 10000 pixels (or a 100 x 100 camera). The transform then must have been 16^3 or 4,096 complex numbers. For the parallel evaluation, the transform is broadcast to the 10,000 camera vectors, which are cast from 32 bit floats to 128 bit complex numbers. This creates a 2D array of 10,000 x 4,096 complex numbers. That, theoretically, requires 5,242,880,000 bits, or 655 megabytes for one evaluation. In spite of this, computing the function in parallel is consistently faster than computing it sequentially.

The normals of the surface defined by the threshold are needed to shade it. Finding the derivative is an important step in determining normals. It is easy to find the derivative of the eggcrate ($w = \sin x \sin y \sin z$), but with the fourier transform, it is nearly impossible to find the slope of the function in closed form. Instead, the slope is approximated using a given offset.

A side effect of using a fourier transform as an approximation is that sharp edges develop overshoots which cannot be removed. This is known as the Gibbs phenomenon. A binary data set will cause overshoots. An alternative is a signed distance field (SDF). When given a surface defined by an equation or an STL, an array of distances from the surface can be created. The "sign" is whether the point is inside the model or not. A negative value typically denotes the inside of the surface, while a positive value is outside the surface.

This method of rendering can take arbitrary data, but is quite slow (see figure below)

resolution	avg. time/frame (sec)		
	125 term Octahedron	Eggcrate bounded	Eggcrate unbounded
50 x 50 pixels	1.186	0.094	0.171
100 x 100 pixels	4.592	0.294	0.545
500 x 500 pixels	116.0	10.53	19.66

5 Applying Domain

Because fourier transforms are based on sinusoidal waves, they are always periodic functions. Even with a limited set of input data, the render will always produce an infinitely tiled version of the surface. Bounding the fourier series to the original domain ensures only one copy of the data is displayed. The camera vectors are monotonically increasing, so a ray that leaves the bounding box will be unable to pass through it again. This can be leveraged to trim the number of vectors calculated per increment of the newton approximation.

The unnecessary computations were trimmed using the maximum render distance. If a vector goes outside of the range of the camera, it is culled. Every increment, the camera vectors were culled based on their relation to the bounding box. culled vectors were pushed away from the camera with the maximum render distance.

This bounding can be applied to mathematically defined data as well. The only difference was that the bounds are not intrinsic to the data and need to be defined separately. This bounding was performed on the eggcrate function ($w = \sin x \sin y \sin z$).

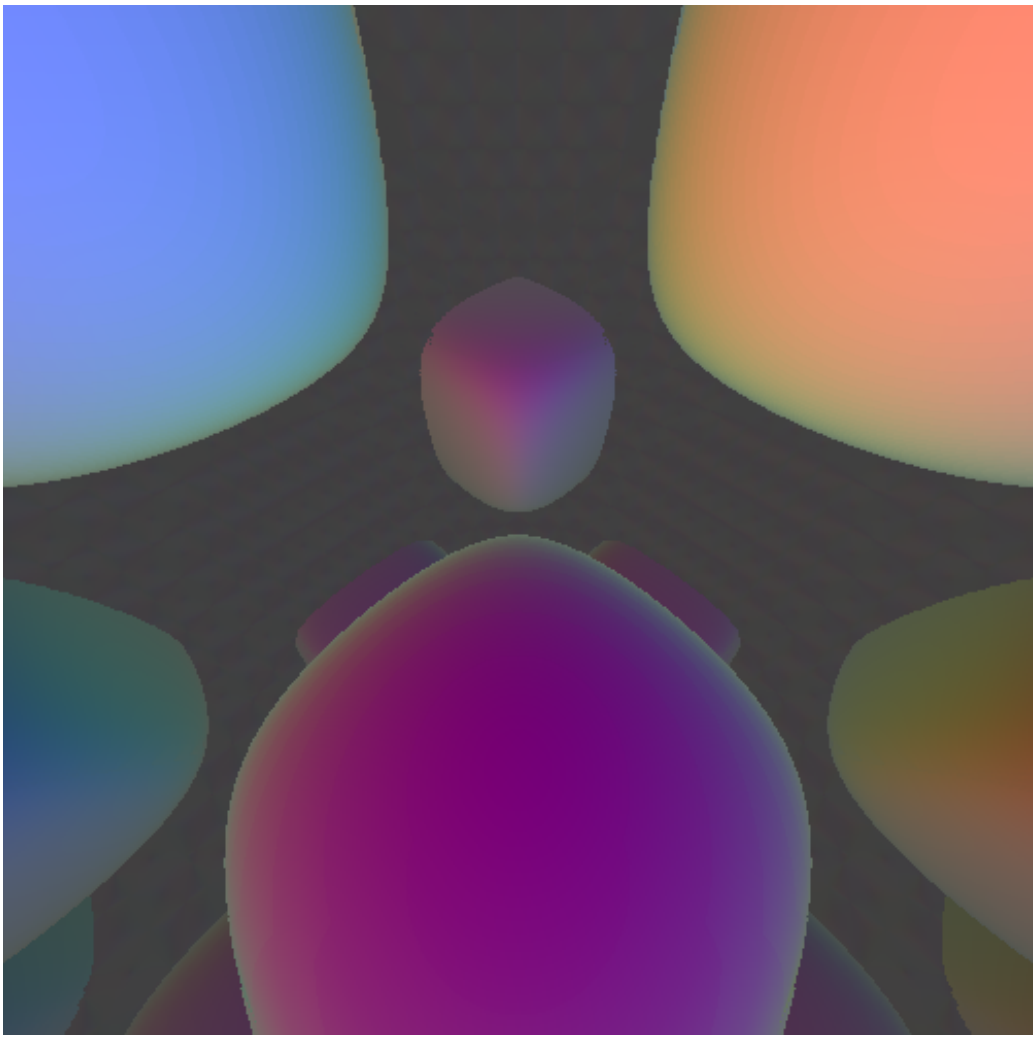


Figure 3: The eggcrate ($-0.1 = \sin x \sin y \sin z$) bounded by a box from -5 to 5 in all axes.

Conclusion

Triangles dominate the field of computer graphics. That includes 3D data representation. Unfortunately, triangles are sub-optimal for visualizing arbitrary data, as precision comes at a steep price, and polygonization algorithms can be arduous to develop and implement. Ray Tracing is the only alternative, but it requires real data, such as functions of x, y, and z. Therefore, converting discrete arbitrary data into continuous data would allow for the implementation of a ray tracing algorithm.

Fourier transforms are capable of performing that conversion. The fourier transform can then be evaluated at arbitrary points. Using a root finding algorithm, such as the Monotonically Increasing Newton Approximation method, the data can be rendered with dynamic resolution.

Arbitrary data was successfully rendered using a fourier transform, but the memory required to perform that render imposed a restriction on the resolution of the data and, consequently, the precision of the fourier transform. In theory, this could be fixed by employing a less general purpose algorithm than NumPy's fast fourier transform and developing an algorithm that emphasizes surfaces as opposed to matching the values of a signed distance field. It proves less efficient for rendering discrete data than triangulation algorithms. This method appears better suited for rendering mathematically defined surfaces, such as 3D sinusoidal functions, and could easily be modified to dynamically render user-inputted equations, given the implementation of an interpreter.

In the future, implementing any method of reducing the memory footprint of fourier transforms would increase the efficiency of this algorithm; however, there are currently no readily available studies attempting to do so. Making the Newton approximations massively parallel, as well as recreating the algorithm in a lower level language such as C, may offer significant performance improvements. Additionally, using lower sampling rates for initial approximations can be used to provide good initial guesses for the root finding algorithm. To ensure that the root finding algorithm is fully optimized, the algorithm will be recreated in a 3D rendering platform to visualize the root finding process for refinement.

6 Sources

18.11.2.2 Algorithms (Inverse 2D FFT) - Originlab Corporation. (n. d.). Retrieved February 20, 2022, from `https://www.originlab.com/doc/Origin-Help/InverseFFT2-Algorithm`

Multidimensional Newton - MIT. (n.d.). Retrieved February 20, 2022, from `https://web.mit.edu/18.06/www/Spring17/Multidimensional-Newton.pdf`