A Report
on

Curve Tracing Using Fourier Analysis

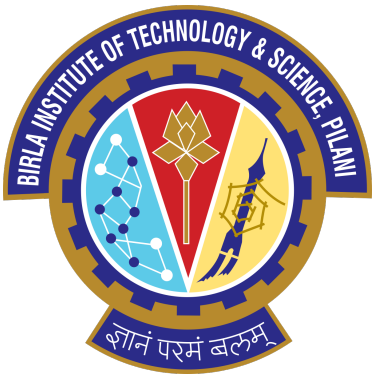
By

Sharanya Ranka **2018A7PS0215G**

Under the guidance of

Prof. Anup Basil Matthew

At



Birla Institute of Technology and Science, Pilani
K.K.Birla Goa Campus
November 30, 2020

Acknowledgements

I would like to express my deep gratitude to Professor Anup Basil Matthew, professor in the Computer Science Department, for providing me the opportunity to take up an SOP under his guidance. I want to thank him for taking some time out of his busy schedule every week and discussing with me the theory and problems during implementation.

I am thankful to my university, BITS Pilani, for providing opportunities to students to take up SOP's under professors, so that we may learn from their expertise and have a rich one-on-one interaction with them.

I like to thank the 3Blue1Brown channel on Youtube. Seeing its video on "*But what is a Fourier series?*" inspired me to try my own implementation of the complex fourier series, and also attempt to generalise to more dimensions.

I would also like to thank my friends Danish and Atharv for taking the time to go through my project and pointing out places where my explanation was lacking or unclear.

Abstract

The aim of this project is to explore Fourier Analysis, and how any general function can be approximated using sines and cosines.

This theory is then applied to curve tracing, and a custom user-given curve is approximated to an arbitrary degree using its Fourier Series. An implementation in Python using complex-valued functions is shown.

Later, we also explore a method to extend the concept to more general functions (i.e. curves in higher dimensions) and attempt to implement the 3D version using Unity with scripting in C#

Contents

Chapter 1

Introduction

1.1 Aim of the Project

The aim of the project is to explore Fourier Analysis, especially in relation to Fourier Series, and use this theory to implement a program that traces out approximations of user-input curves.

An example of the implementation is given below. We also want to extend

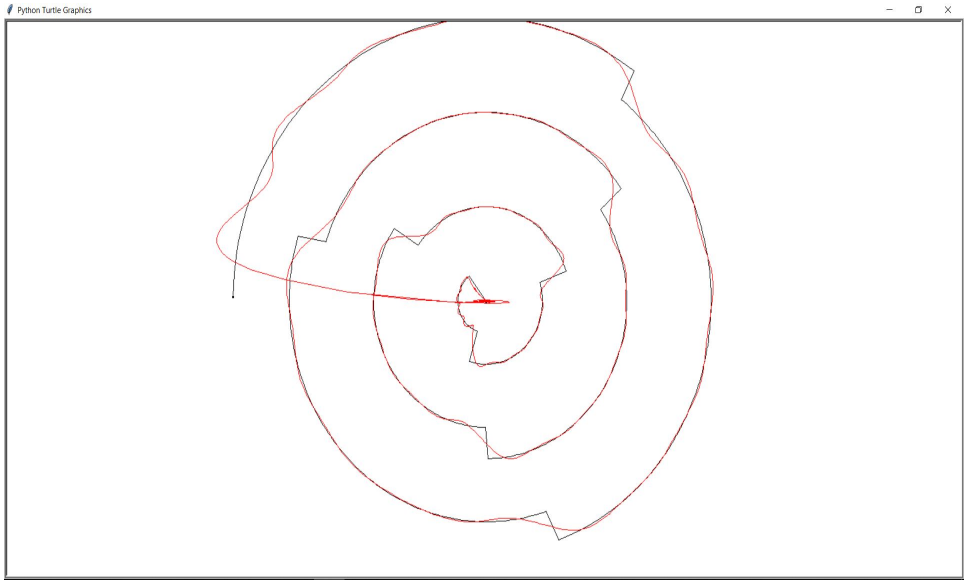


Figure 1.1: Input: Black Curve, Fourier Series approximation: Red Curve

the method to higher dimensional curves.

Please note that the explanations given here are not completely rigorous proofs, but are presented to give reasoning for why the methods work.

1.2 What is Fourier Analysis?

In mathematics, Fourier analysis is the study of the way general functions may be represented or approximated by sums of simpler trigonometric functions (i.e. sines and cosines).

Fourier analysis grew from the study of Fourier series, and is named after Joseph Fourier[1], who showed that representing a function as a sum of trigonometric functions greatly simplifies the study of heat transfer. As an example, we want to know whether a general function $f(x)$ can be written as -

$$f(t) = \sum_{k=1}^{n} a_k \sin(2\pi kt) + b_k \cos(2\pi kt) \tag{1.1}$$

For some n where a_k and b_k can be chosen appropriately. The 2π is used to make the functions periodic with period 1 and k can be thought of as the frequency of the sine and cosine functions. This is called its Fourier Series. If f can indeed be written in such a way, we are interested in finding out a method to find a_k's and b_k's.

[1] Jean-Baptiste Joseph Fourier (21 March 1768 – 16 May 1830) was a French mathematician and physicist born in Auxerre and best known for initiating the investigation of Fourier series, which eventually developed into Fourier analysis and harmonic analysis, and their applications to problems of heat transfer and vibrations.

1.3 A Note on functions considered

To successfully apply fourier series to a function f, it needs to be periodic with some period P. For convenience, in this project, *P=1 unit*

In the last section, we saw that the sine and cosine functions contained the argument $2\pi kt$. We can check that for all integer values of k, the sine and cosine functions are periodic with period 1 (their fundamental periods may be different). Therefore, we only consider functions which are periodic with period 1.

We begin with functions that take in one real number as an input and output one real number i.e. $f : \mathbf{R} \to \mathbf{R}$. Later on we will comment about complex-valued functions $f : \mathbf{R} \to \mathbf{C}$. We will also attempt to generate a method to find an approximating curve to functions of the type $f : \mathbf{R} \to \mathbf{R}^n$ for any n using fourier analysis techniques.

1.4 Using Complex Exponentials

Instead of using simple sine/cosine functions to write the Fourier Series, we will instead use complex exponentials. It may seem unnecessarily complicated, but will greatly reduce complexity when we try to find the coefficients for the fourier series. First of all note that-

$$e^{i(2\pi kt)} = \cos(2\pi kt) + i \cdot \sin(2\pi kt) \tag{1.2}$$

Therefore we can write the cosine and sine functions as

$$\cos(2\pi kt) = \frac{e^{i(2\pi kt)} + e^{-i(2\pi kt)}}{2} \tag{1.3}$$

$$\sin(2\pi kt) = \frac{e^{i(2\pi kt)} - e^{-i(2\pi kt)}}{2i} \tag{1.4}$$

And on substituting the above two equations into equation 1.1 we find that for each frequency k, we will have 2 complex exponentials, one corresponding to $+k$ and another to $-k$. We can then rewrite equation 1.1 as -

$$f(t) = \sum_{k=-n}^{n} c_k \cdot e^{2\pi ikt} \tag{1.5}$$

Note the change in the limits from $1 \to n$ to $-n \to n$.

1.5 Finding Fourier Coefficients

We are interested in finding the c_k's so that we can find the expression for the fourier series of the function. Therefore, assuming that f can be written as a series shown in equation 1.5 our problem reduces to finding the coefficients c_k for all k,$-n <= k <= n$
To find the m'th coefficient c_m we isolate it from the series 1.5

$$c_m \cdot e^{2\pi imt} = f(t) - \sum_{k \neq m} c_k \cdot e^{2\pi ikt} \tag{1.6}$$

$$c_m = f(t) \cdot e^{-2\pi imt} - \sum_{k \neq m} c_k \cdot e^{2\pi i(k-m)t} \tag{1.7}$$

Integrating on both sides w.r.t t with limits 0 to 1 we get

$$\int_0^1 c_m dt = \int_0^1 f(t) \cdot e^{-2\pi imt} dt - \sum_{k \neq m} \int_0^1 (c_k \cdot e^{2\pi i(k-m)t}) dt \tag{1.8}$$

The L.H.S is simply c_m. Let us look at the second term of the R.H.S more closely. We will look at a single k and find the total result.

$$\int_0^1 (c_k \cdot e^{2\pi i(k-m)t}) dt = \int_0^1 c_k (\cos 2\pi(k-m)t + i \cdot \sin 2\pi(k-m)t) dt \tag{1.9}$$

As $k \neq m$ $\int_0^1 \cos(2\pi(k-m)t)dt$ and $\int_0^1 \sin(2\pi(k-m)t)dt$ evaluate to 0. This happens for all k on the R.H.S of the equation, and equation 1.8 can be rewritten as -

$$\int_0^1 c_m dt = c_m = \int_0^1 f(t) \cdot e^{-2\pi imt} dt \tag{1.10}$$

Thus, if a function f can be written a series 1.5, equation 1.10 gives us a method to evaluate c_k for all k

These c_k's are special coefficients which we will call fourier coefficients of the function, and will hereon represent it as $\hat{f}(k)$.

Now, if we consider any periodic function with period 1 i.e. $f(t)$ and define $\hat{f}(k)$ as -

$$\hat{f}(k) = c_k = \int_0^1 f(t) \cdot e^{-2\pi ikt} dt \tag{1.11}$$

where $\hat{f}(k)$ is known as the kth fourier coefficient of f

Can we say that -

$$f(t) = \sum_{k=-n}^{n} \hat{f}(k) \cdot e^{2\pi ikt} \quad ? \tag{1.12}$$

In general, the answer is NO. This is because some functions may not be expressible in the form 1.5 for any finite n.

Is this the end of the road? Fortunately, we can yet say something significant about expression 1.12. Although the R.H.S isn't exactly the same as the L.H.S, it is an approximation of it. So we can say -

$$f(t) \approx \sum_{k=-n}^{n} \hat{f}(k) \cdot e^{2\pi ikt} \tag{1.13}$$

For functions f that cannot be written as finite sums of sines and cosines, the R.H.S approximates the L.H.S. The approximation improves as we consider larger n i.e. as $n \to \infty$.

1.6 A Note about the results

The value of the integral in equation 1.11 can be a complex number. In that case, the coefficient will be a complex number.

The results in the previous section were found for a real-valued periodic function $f : \mathbf{R} \to \mathbf{R}$, with period 1 but the same holds for complex-valued periodic functions i.e. $f : \mathbf{R} \to \mathbf{C}$. One trivial example of such a function is $f(t) = \cos(2\pi t) + i \cdot \sin(2\pi t)$. Alternatively we could consider this as a function from one-dimensional reals to 2-dimensional reals, i.e. $f : \mathbf{R} \to \mathbf{R}^2$. The previous example could be written as $f(t) = (\cos(2\pi t), \sin(2\pi t))$ where $\cos(2\pi t)$ is the x-component and $\sin(2\pi t)$ is the y-component.

One important point is to be noted about equation 1.11 in case of real valued functions. It can be proved that for some k, we have -

$$c_{-k} = \overline{c_k} \quad or \quad \hat{f}(-k) = \overline{\hat{f}(k)} \tag{1.14}$$

i.e. c_{-k} is the complex conjugate of c_k. This can be intuitively understood because all the complex portions of the terms must cancel out, and they do so in pairs, to get a final real sum.

This constraint (eqn 1.14) is not present for complex valued functions.

Chapter 2

Implementation in Python

2.1 Introduction

The aim of the implementation program was to calculate the fourier series of any complex-valued function fed in by the user, and to show(trace) the fourier series curve over the original curve so that the user can see to what degree the approximation was made. The fourier series would be calculated for a range of max_frequencies(n) in order to show that the approximation improves with larger n. max_frequency (i.e. n) refers to the limits of the summation used in 1.13

There are 3 main steps in this program.

1. **Get the user input** and **convert it into a function** of the form $f : \mathbf{R} \to \mathbf{C}$. We will be using a complex valued function, because it will be more general and will allow greater freedom to the user for the input.
2. **Perform Fourier Analysis** on the function in order to find the fourier coefficients for the fourier series. In particular, perform equation 1.11 to find $\hat{f}(k)$ for all k.
3. **Trace the curve** on top of the original user-input to show how the fourier series approximates it.

2.2 Getting User Input

The first stage in the process is to get in the user input.

The technique used here was to let the user specify a set of points on the complex plane(or 2D plane) and the user-generated function would be a set of line segments joining the points in the order given by the user. Once the user finishes entering the points, he/she presses "Enter" and a final line segment joining the last point and the first point is added to the curve to make it 'closed loop' and periodic.

```
def register_turtle_movements(points):
        # Makes a list of points on canvas clicked by user

        custom_function_turtle=turtle.Turtle("circle")
        custom_function_turtle.shapesize(0.2, 0.2, 0)
        custom_function_turtle.penup()

                def next_point(x, y):
                custom_function_turtle.goto(x, y)
                points.append(custom_function_turtle.pos())
                custom_function_turtle.pendown()

        turtle.listen()
        turtle.onscreenclick(next_point)
        turtle.onkey(fill_last_and_continue, "Return")

        turtle.done()


def fill_last_and_continue():
        # Utility function used after register_turtle_movements

        global points
        custom_function_turtle = turtle.Turtle("circle")
        custom_function_turtle.shapesize(0.2, 0.2, 0)
        custom_function_turtle.penup()
        custom_function_turtle.goto(points[len(points)-1])
        custom_function_turtle.pendown()
        custom_function_turtle.goto(points[0][0], points[0][1])
        points.append(custom_function_turtle.pos())

        perform_remaining()
```

This series of line segments is then converted into a function $f : \mathbf{R} \to \mathbf{C}$. This is done by parameterising each point on the curve by its distance to the

"starting point" along the curve. This, along with the fact that the curve starts and ends at the same point ensures periodicity of the complex-valued function. In order to make the period=1, we simply normalise by dividing the distance values by the total distance covered by the curve. Hence all 'distance' values are now between 0 and 1.

We then form a function that takes in a real number t between 0 and 1 and finds the point on the curve that has the same normalised distance as t. In the implementation, this is done by finding two user provided points that contain the point of interest using *binary search* and then interpolating the position of the required point. The complex number that represents the point is then returned back.

```
def make_custom_function(points):
        #Making function out of collected points

        total_dis=total_distance(points)
        custom_function=get_function(points, total_dis)
        return(custom_function)


def total_distance(points):
        #Finds total distance of the curve given

        total_dis=0
        prev_point=points[0]
        for point in points[1:]:
                total_dis+=math.sqrt((prev_point[0]-point[0])**2+
                (prev_point[1]-point[1])**2)
                prev_point=point

        return(total_dis)

def get_function(points,total_dis):
        #Converts the list of points into a function

        cur_dis=0
        prev_point=points[0]
        function_reference=[]
        function_reference.append(tuple([cur_dis,prev_point]))

        #Normalising distance values
        for i,point in enumerate(points[1:]):
                cur_dis += math.sqrt((prev_point[0] - point[0]) ** 2 +
                 (prev_point[1] - point[1]) ** 2)
```

```
            function_reference.append(tuple([cur_dis/total_dis,point]))
            prev_point=point

    def function_from_points(t):
    #Function to be returned

            if t<0 or t>1:
                    print("error_t_is",t)
                    return(0)
            else:
                    end=len(function_reference)-1
                    start=0
                    #Performing binary search to find
                    #the appropriate interval
                    while(start+1<end):
                            mid = (end + start) // 2
                            if function_reference[mid][0]>t:
                                    end=mid
                            else:
                                    start=mid


            #Interpolating in the interval to return
            #final function value
            first_point=function_reference[start][1]

            first_normalised_distance=function_reference[start][0]

            next_point=function_reference[end][1]

            next_normalised_distance=function_reference[end][0]

            x_val=first_point[0]+((t-first_normalised_distance)/
            (next_normalised_distance-first_normalised_distance))*
            (next_point[0]-first_point[0])

            y_val=first_point[1]+((t-first_normalised_distance)/
            (next_normalised_distance-first_normalised_distance))*
            (next_point[1]-first_point[1])

            return(complex(x_val,y_val))

    return(function_from_points)
```

The *perform_remaining()* function basically calls *make_custom_function()* and makes sure the proper program flow is maintained.

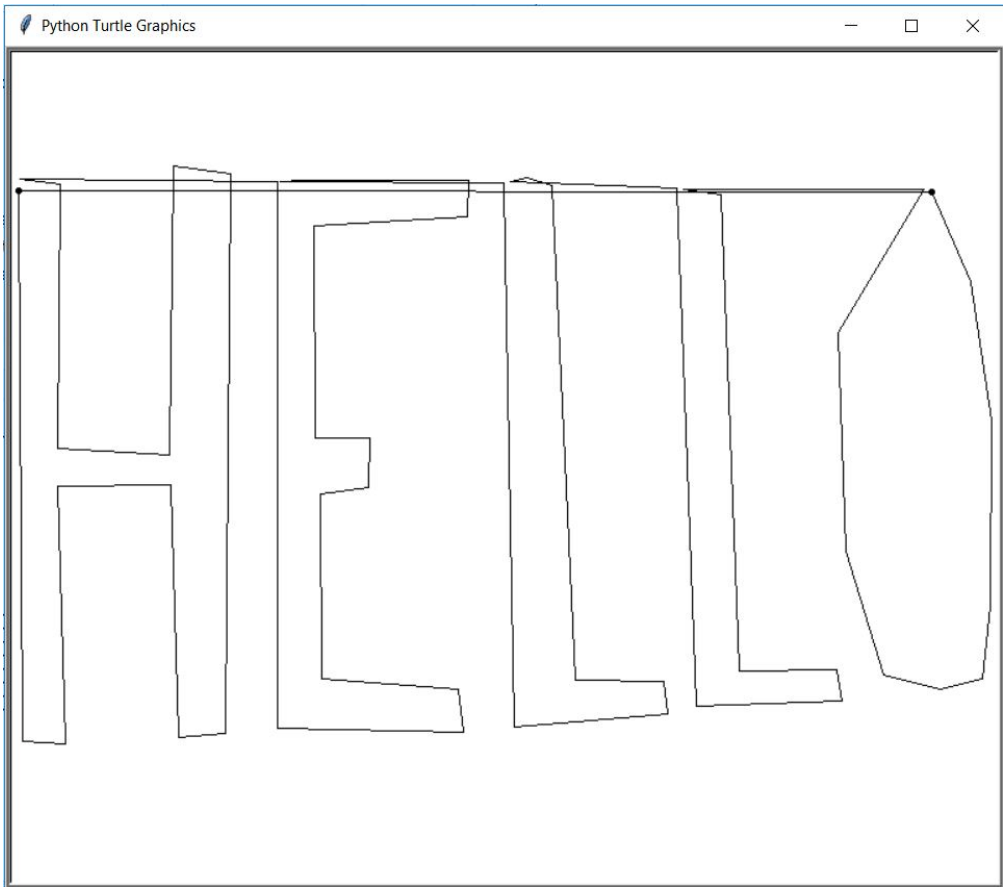


Figure 2.1: Accepting User Input

2.3 Computing Fourier Coefficients

Now that we've got the user input converted into a function, we would like to apply Fourier Analysis on it to get the fourier coefficients of the complex exponentials.

This process is done according to the definition of fourier coefficients (equation 1.11), with the integration being replaced by a finite sum for practicality. The *delta_for_integration* parameter is used to set the value of dt. A dictionary containing *key : value* pairs as *frequency : coefficient* will be returned.

```
def find_fourier_coefficients_complex(function, max_frequency,
    delta_for_integration):
        #Returns a dictionary containing the fourier coefficients

        coefficients={}
        for frequency in range(-max_frequency,max_frequency+1):
                coefficients[frequency]=0
                ck=0
                #Performing integration
                for t in np.arange(0,1,delta_for_integration):
                        ck+=cmath.rect(1,-2*cmath.pi*frequency*t)*
                            function(t)*delta_for_integration

                coefficients[frequency]=ck

        return(coefficients)
```

We could print the coefficients dictionary if we wanted to see how the values tend to look. Here c_k and c_{-k} won't be complex conjugates because the function is complex-valued.

Another important point to note is that the *function* is sampled $\frac{1}{delta\_for\_integration}$ number of times (also known as sampling frequency). The minimum sampling frequency is dependent of the maximum frequency(n) we want to use for approximation. In this project, we will assume that the sampling frequency is high enough to not cause a problem.

2.4 Computing Fourier Series

Finally, now that we have the fourier coefficients, we can find the fourier series of the function by applying equation 1.13. Again, the fourier series is computed according to the definition. The output of the following code is a function that is a series of complex exponentials built from the fourier coefficients.

```
def find_fourier_series_complex(coefficients):
        #Returns the fourier series from the coefficients

        # making the function
        def fourier_series(t):
                return_val=0
                for freq in coefficients:
                        return_val+=coefficients[freq]*cmath.rect
                           (1,2*cmath.pi*freq*t)

                return(return_val)

        return(fourier_series)
```

Now that we've got the fourier series of the custom user function, we can go ahead and draw it.

```
def draw_function_complex(f,end_time,draw_scale=1,time_scale=1.0,
    colour="black"):
        #Draw the function given as a parameter

        function_drawer=turtle.Turtle("circle")
        function_drawer.shapesize(0.1,0.1,0)
        function_drawer.color(colour,colour)
        function_drawer.penup()
        pos0 = f(0) * draw_scale
        function_drawer.goto(pos0.real, pos0.imag)
        function_drawer.pendown()

        start=time.time()

        while((time.time()-start)*time_scale<end_time):
                newpos=f((time.time()-start)*time_scale)*
                   draw_scale
                function_drawer.goto(newpos.real,newpos.imag)

        time.sleep(30)
        function_drawer.clear()
```

```
del(function_drawer)
```

We can calculate several different fourier series with different maximum frequncies(n)to see how the approximation improves with increase in n. An example is shown below.

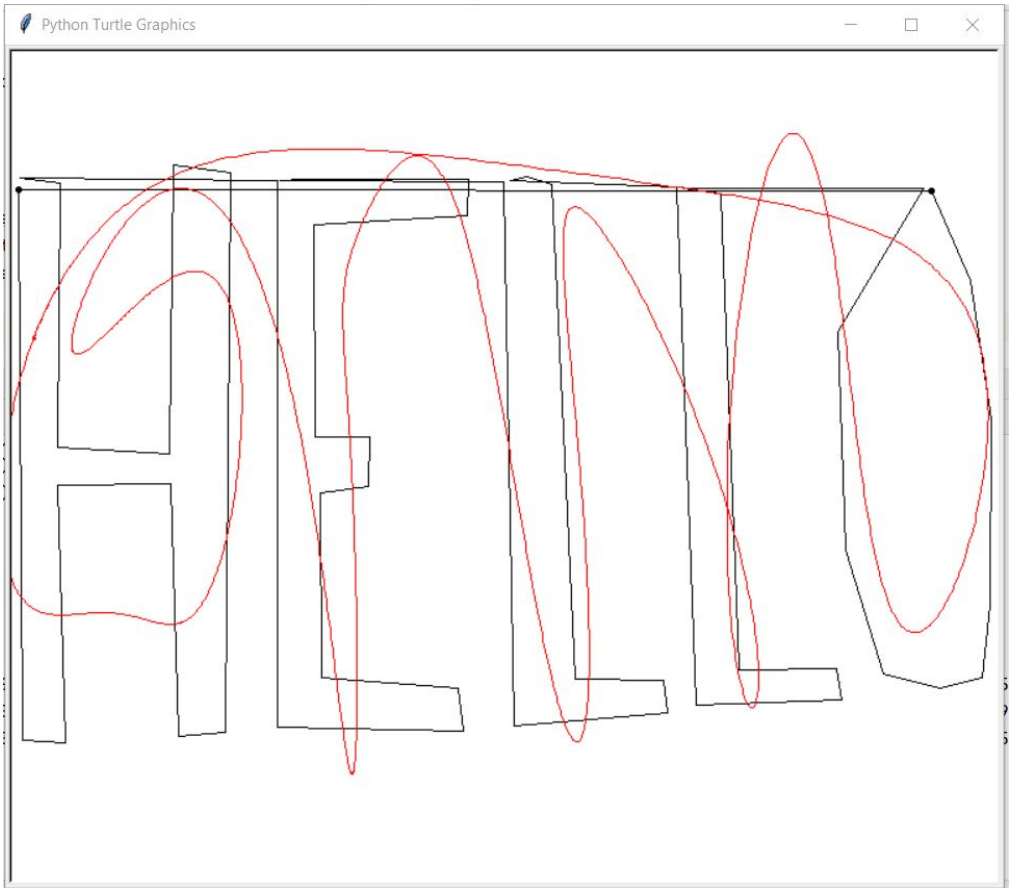


Figure 2.2: Fourier Series Approximation of the "HELLO" curve($n = 10$)

Here we can clearly see that for the given curve, $n = 10$ does a very poor job of approximating the curve.

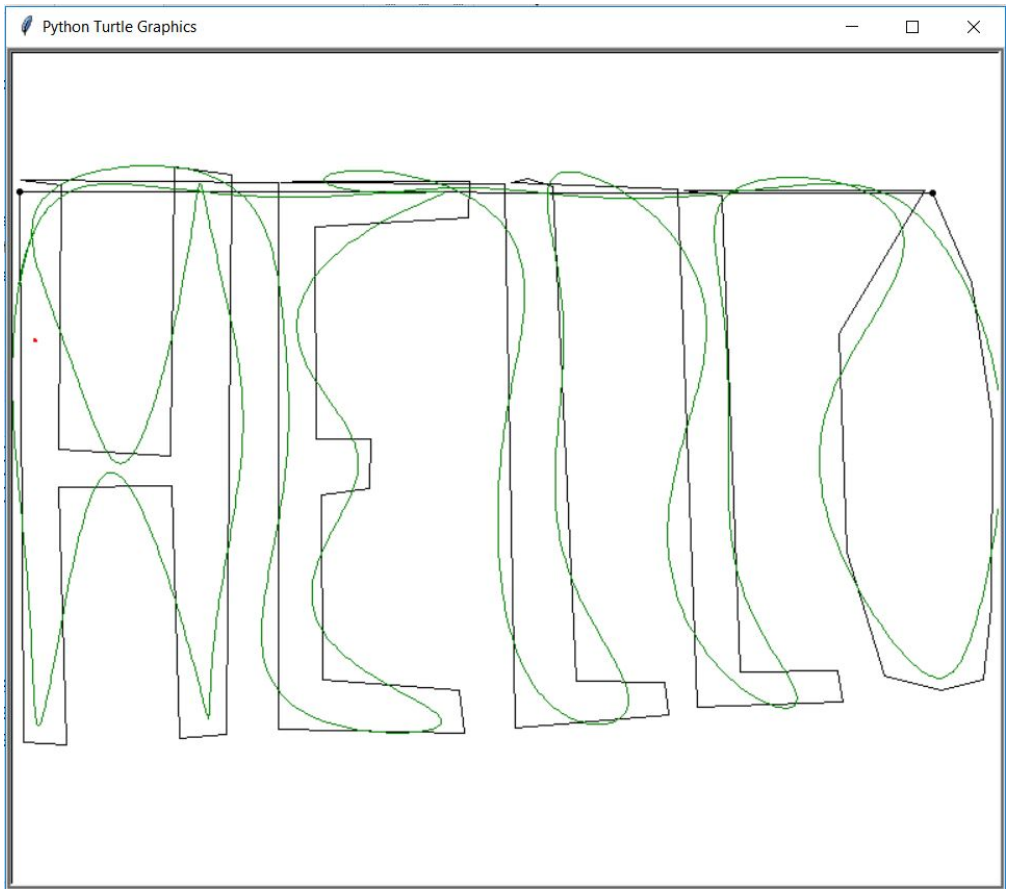


Figure 2.3: Fourier Series Approximation of the "HELLO" curve($n = 25$)

$n = 25$ doesn't do much better, but we can see that it has taken the form of the curve somewhat.

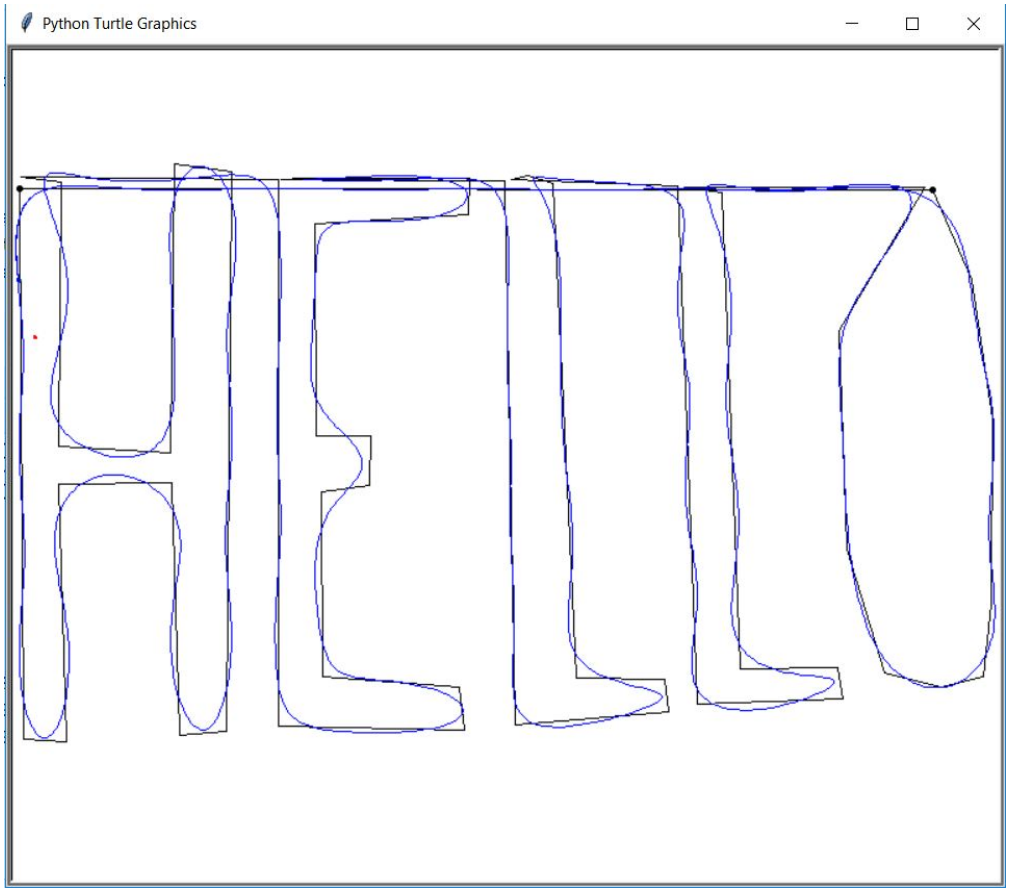


Figure 2.4: Fourier Series Approximation of the "HELLO" curve($n = 50$)

$n = 50$ does a good job of approximating the curve. Even higher values for n can be used leading to even better approximations.

Chapter 3

Higher dimensions

3.1 Extending the method to n dimensions

Now that we have got a working implementation to find fourier series approximations for functions of the form $f : \mathbf{R} \to \mathbf{C}$, we ask the following question -

Is it possible to extend this to extend this to curves in higher dimensions (i.e. functions of the form $f : \mathbf{R} \to \mathbf{R}^m$)? We will explore one technique that allows us to do this.

The main idea is to perform component-wise fourier analysis, and to combine the fourier series we get for each component. We shall show that this method does indeed approximate the actual function f.

Just for a recap, we defined the fourier coefficient $\hat{f}(k)$ as-

$$\hat{f}(k) = c_k = \int_0^1 f(t) \cdot e^{-2\pi ikt} dt \tag{3.1}$$

and the fourier series summation that approximates a funtion f as-

$$f(t) \approx \sum_{k=-n}^{n} \hat{f}(k) \cdot e^{2\pi ikt} \tag{3.2}$$

Now, consider a new function $f : \mathbf{R} \to \mathbf{R}^m$. We will assume f can be written as an ordered tuple -

$$f(t) = (x_1(t), x_2(t), x_3(t), ..., x_m(t)) \tag{3.3}$$

where $x_i : \mathbf{R} \to \mathbf{R}$ for all i from 1 to m.

We also assume that $f(t)$ is periodic with period 1 (i.e. $f(t+1) = f(t)$ for all t).

We define a new function $g : \mathbf{R} \to \mathbf{R}^m$. This will be our approximation curve for f. Let us write g as -

$$g(t) = (y_1(t), y_2(t), y_3(t), ..., y_m(t)) \tag{3.4}$$

such that $y_i : \mathbf{R} \to \mathbf{R}$ for all i from 1 to m and y_i is the fourier series approximation of x_i. Using equations 3.1 and 3.2 -

$$y_i(t) \approx \sum_{k=-n}^{n} (\int_0^1 e^{-2\pi ikt} \cdot x_i(t)dt)e^{2\pi ikt} \tag{3.5}$$

As y_i approximates x_i we can write -

$$\lim_{n\to\infty} y_i(t_0) = x_i(t_0) \tag{3.6}$$

$$\text{for all } t_0$$

and we want to show that g approximates f i.e.

$$\lim_{n\to\infty} g(t) = f(t) \tag{3.7}$$

We will consider our convergence measure as $|g(t) - f(t)|$ that is, g is said to converge to f if -

$$\lim_{n\to\infty} |g(t_0) - f(t_0)| = 0 \tag{3.8}$$

$$\text{for all } t_0$$

Using the definition of norm for $|x|$ at a particular t_0 we have -

$$\begin{aligned} |g(t_0) - f(t_0)| =& ((y_1(t_0) - x_1(t_0))^2 + \\ & (y_2(t_0) - x_2(t_0))^2 + \\ & . \quad . \quad . \quad . \quad . \quad + \\ & (y_m(t_0) - x_m(t_0))^2)^{1/2} \end{aligned} \tag{3.9}$$

Now, from the limit 3.6 and convergence measure we have for all i from 1 to m and an arbitrary $\varepsilon > 0$ -

$$\exists \quad integer \quad q_i > 0 \quad s.t. \quad \forall q >= q_i$$

$$|y_i(t_0) - x_i(t_0)| < \frac{\varepsilon}{\sqrt{m}} \tag{3.10}$$

for all t_0

This can be said for each component of the function f. Now consider $q_0 = max(q_1, q_2, ..., q_m)$. All the inequalities hold for $q >= q_0$ as well.

Thus, $|g(t_0) - f(t_0)|$ on applying equation 3.10 to all components in equation 3.9 -

$$\begin{aligned} |g(t_0) - f(t_0)| &< (\frac{\varepsilon^2}{m} + \frac{\varepsilon^2}{m} + ... + \frac{\varepsilon^2}{m})^{1/2} \\ &< (\varepsilon^2)^{1/2} \\ &< \varepsilon \end{aligned} \tag{3.11}$$

for any arbitrary $\varepsilon > 0$

Hence we can say that -

$$\lim_{n \to \infty} |g(t_0) - f(t_0)| = 0$$

for all t_0

and that function g approximates function f.

This is the method that we will be using further to compute fourier series approximations of curves in higher dimensions.

Chapter 4

A Better Algorithm

4.1 Analysing Time Complexity of the Naïve Algorithm

We will call the algorithm that calculates coefficients according to the definition (3.1) as the Naïve algorithm. This is because there is another algorithm that does the same computation in much lesser time. For practical implementation purposes, we cannot use the integral as in equation (3.1). Rather we will use an approximation of it (replacing dt by Δt and the integral by a summation over appropraite interval). The equation we will use for computational purposes is then -

$$\hat{f}(k) = c_k \approx \sum_{t=0}^{1} f(t) \cdot e^{-2\pi i k t} \cdot \Delta t \tag{4.1}$$

for some finite value of Δt.

The number of times the function is sampled is equal to $\frac{1}{\Delta t}$. This is equal to the sampling rate or sampling frequency. We will assume that we want coefficients for frequencies between $k = -n$ to $k = +n$ where n is the maximum frequency. It turns out that the maximum frequency n determines the sampling rate of the function. In this case, as we have $2n + 1$ unique frequencies, the minimum value of Δt should be $2n + 1$.

For each frequency, we have to sample the function atleast $2n + 1$ times. This is therefore an $\mathcal{O}(n)$ operation. Since we have to do this for the $2n + 1$ frequencies, all the coefficients can be found in $\mathcal{O}(n^2)$ time.

Now we try to compute the time complexity for the reverse process i.e.

finding the fourier series from the coefficients. We refer to the equation -

$$f(t) \approx \sum_{k=-n}^{n} \hat{f}(k) \cdot e^{2\pi ikt} \tag{4.2}$$

If we try to compute $f(t)$ for a particular time instant t_0, we simply have to evaluate the sum once. Hence the time complexity for this process is $\mathcal{O}(n)$. Note that this complexity is only for finding the function value at a single point. For drawing the curve we will want to find the function approximation at several points (say m points). For this the time complexity would be $\mathcal{O}(m \cdot n)$.

It turns out, for large values of n, the first process is very time consuming, and there is a better way to compute the fourier coefficients.

Regarding the reverse process, we will stick to the $\mathcal{O}(m \cdot n)$ method since we want to actually draw the function to show the approximation to the users. The only way to come up with a faster solution would be to come up with a faster than $\mathcal{O}(n)$ way to compute equation (4.2) and currently I do not know if such a method exists.

4.2 The Fast Fourier Transform (FFT)

The better method that we were talking about in the last section is the Fast Fourier Transform algorithm or **FFT** algorithm for short. It has a faster time complexity than the naïve algorithm $\mathcal{O}(n \log n)$ compared to $\mathcal{O}(n^2)$. This section gives some of the intuition behind the algorithm. The analysis of the time complexity is done in the next section.

We will start by discussing the forward process (getting the fourier coefficients from the function samples).

Consider that we have function sampled at n points on the curve. We will call the values f_0, f_1, f_2,...,f_{n-1}. Using these samples we want to get the fourier coefficients $\hat{f}_0$, $\hat{f}_1$, ..., $\hat{f}_{n-1}$. Note that here the number of fourier coefficients is equal to the number of function samples. We discussed this in the last chapter (here we are using n values instead of $2n + 1$ values). Also, here we do not mention positive and negative frequencies. This is just for making notation simpler. We will extract both positive and negative frequencies after the process.

We can say that we want some transformation so that we can convert the f_i's to $\hat{f}_k$'s i.e. -

$$\begin{bmatrix} f_0 \\ f_1 \\ . \\ . \\ . \\ f_{n-1} \end{bmatrix} \xrightarrow{\text{forward fft}} \begin{bmatrix} \hat{f}_0 \\ \hat{f}_1 \\ . \\ . \\ . \\ \hat{f}_{n-1} \end{bmatrix} \tag{4.3}$$

It turns out we can apply a matrix transformation on the f_i's to get $\hat{f}_k$'s. To do this we will first look at a slightly modified form of equation (4.1). We change Δt to $\frac{1}{n}$ (to get n samples) and we write t as $\frac{j}{n}$ for $j = 0, 1, 2, ..., n-1$. We get -

$$\hat{f}(k) = c_k \approx \frac{1}{n} \cdot \sum_{j=0}^{n-1} f(j) \cdot e^{-2\pi i k(\frac{j}{n})} \tag{4.4}$$

For notational simplicity, we define a variable ω_n to be the fundamental

frequency i.e.

$$\omega_n = e^{-2\pi ik(\frac{1}{n})} \tag{4.5}$$

Now we can finally write the matrix multiplication as -

$$\begin{bmatrix} \hat{f}_0 \\ \hat{f}_1 \\ \hat{f}_2 \\ . \\ . \\ \hat{f}_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & ... & 1 \\ 1 & \omega_n & \omega_n^2 & ... & \omega_n^{n-1} \\ 1 & \omega_n^2 & \omega_n^4 & ... & \omega_n^{2(n-1)} \\ . & . & . & . & . \\ . & . & . & . & . \\ 1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & ... & \omega_n^{(n-1)^2} \end{bmatrix} \begin{bmatrix} f_0 \\ f_1 \\ f_2 \\ . \\ . \\ f_{n-1} \end{bmatrix} \tag{4.6}$$

Or as a shorthand we can write -

$$\hat{f} = F_n \cdot f$$

The validity of the matrix equations can be easily seen by using equation (4.5) to equation (4.4) and knowing that the rows correspond to $k = 0, 1, 2, ..., n-1$. Please note that we have not included the $\frac{1}{2n}$ factor here for simplicity. We can adjust this after getting the $\hat{f}(k)$'s as defined in (4.6).

The square matrix in equation (4.6) is called the DFT matrix (Discrete Fourier Transform matrix) and we will refer to it by the name F_n. This equation is simply a restatement of equation (4.4) in matrix form.

The reason for the reduction in time complexity is the good old "divide and conquer" strategy used in so many algorithms. Interestingly, the R.H.S in equation 4.6 can be factorised as follows -

$$\hat{f} = F_{2n} \cdot f = \begin{bmatrix} I_n & +D_n \\ I_n & -D_n \end{bmatrix} \begin{bmatrix} F_n & 0 \\ 0 & F_n \end{bmatrix} \begin{bmatrix} f_{even} \\ f_{odd} \end{bmatrix} \tag{4.7}$$

Where I_n is the $n \times n$ identity matrix. D_n is a diagonal matrix of size $n \times n$

and F_n is the $n \times n$ DFT matrix. D_n, f_{even} and f_{odd} are given as-

$$D_n = \begin{bmatrix} 1 & 0 & 0 & ... & 0 \\ 0 & \omega_{2n} & 0 & ... & 0 \\ 0 & 0 & \omega_{2n}^4 & ... & 0 \\ . & . & . & . & . \\ . & . & . & . & . \\ 0 & 0 & 0 & ... & \omega_{2n}^{(n-1)} \end{bmatrix}$$

and

$$f_{even} = \begin{bmatrix} f_0 \\ f_2 \\ f_4 \\ . \\ . \\ f_{2n-2} \end{bmatrix}, f_{odd} = \begin{bmatrix} f_1 \\ f_3 \\ f_5 \\ . \\ . \\ f_{2n-1} \end{bmatrix}$$

Here we have used a max frequency of $2n$ instead of n for notational simplicity. This shows the "divide" step and can be similarly written for F_n using $\frac{n}{2}$. This also puts a constraint on the possible values of the maximum freqency (it has to be a power of 2).

This can be verified by expanding the R.H.S and comparing it to the L.H.S although the calculations are a bit tedious.

The recursion can be seen in equation (4.7) To see it more clearly, we can rewrite equation (4.7) as -

$$\hat{f} = F_{2n} \cdot f = \begin{bmatrix} I_n & +D_n \\ I_n & -D_n \end{bmatrix} \begin{bmatrix} F_n \cdot f_{even} \\ F_n \cdot f_{odd} \end{bmatrix} \tag{4.8}$$

$F_n \cdot f_{even}$ and $F_n \cdot f_{odd}$ are of the same form as $F_{2n} \cdot f$ and can be broken down further. The base case for frequency = 1 can be handled by simply evaluating the matrix product.

Here we have discussed only the forward process (getting the fourier coefficients). The backward process of finding the sample points (f_i's) from the fourier coefficients ($\hat{f}_k$'s) is similar, but we will not be using it in this project as we want to plot the fourier series.

We will be seeing the time complexity for this algorithm in the next section.

4.3 Analysing the Time Complexity of the FFT algorithm

As mentioned in the previous section, we will only consider the forward FFT algorithm for time complexity analysis. We will write equation (4.8) again to discuss the time taken.

$$\hat{f} = F_{2n} \cdot f = \begin{bmatrix} I_n & +D_n \\ I_n & -D_n \end{bmatrix} \begin{bmatrix} F_n \cdot f_{even} \\ F_n \cdot f_{odd} \end{bmatrix}$$

Here we can see that to compute $F_{2n} \cdot f$, we need to first compute two similar quantities - $F_n \cdot f_{even}$ and $F_n \cdot f_{odd}$ and then do a matrix and vector multiplication i.e. $\left[\begin{smallmatrix} I_n & +D_n \\ I_n & -D_n \end{smallmatrix}\right]$ with $\left[\begin{smallmatrix} F_n \cdot f_{even} \\ F_n \cdot f_{odd} \end{smallmatrix}\right]$.

Let us first discuss the latter matrix multiplication first. We claim that this is an $\mathcal{O}(n)$ operation. To see why this is true, note that the matrix $\left[\begin{smallmatrix} I_n & +D_n \\ I_n & -D_n \end{smallmatrix}\right]$ is sparse. In each row, there are only 2 non-zero entries because the component submatrices are all diagonal matrices. Thus we can simply compute it within one for loop.

If we call the matrix $\left[\begin{smallmatrix} F_n \cdot f_{even} \\ F_n \cdot f_{odd} \end{smallmatrix}\right]$ as A and write it as

$$A = \begin{bmatrix} a_0 \\ a_2 \\ . \\ . \\ a_{2n-2} \\ a_1 \\ a_3 \\ . \\ . \\ a_{2n-1} \end{bmatrix}$$

The final result vector will look like the following

$$\hat{f} = F_{2n} \cdot f = \begin{bmatrix} 1 \cdot a_0 + 1 \cdot a_1 \\ 1 \cdot a_2 + \omega_{2n} \cdot a_3 \\ . \\ . \\ 1 \cdot a_{2n-2} + \omega_{2n}^{(n-1)} \cdot a_{2n-1} \\ 1 \cdot a_0 - 1 \cdot a_1 \\ 1 \cdot a_1 - \omega_{2n} \cdot a_3 \\ . \\ . \\ 1 \cdot a_{2n-2} - \omega_{2n}^{(n-1)} \cdot a_{2n-1} \end{bmatrix} \tag{4.9}$$

which can be calculated in $\mathcal{O}(n)$ time.

Now to compute time complexity for the entire algorithm, we can write the time taken as a recursive function of $\mathcal{T}(n)$ as the following -

$$\mathcal{T}(2n) = \mathcal{O}(n) + 2 \cdot \mathcal{T}(n) \tag{4.10}$$

and using the Master Theorem (or simply checking that the contribution of each level is equal and then multiplying that by the number of levels i.e. $\log n$) we get that the time complexity of the FFT algorithm is $\mathcal{O}(n \cdot \log n)$.

Chapter 5

Implementation in Unity and C#

5.1 Introduction

The 3D implementation of Curve Tracing using Fourier Analysis is a bit different compared to the 2D version done in Python. In the 2D version we could simply use complex numbers to find the Fourier Series of the custom user function.

On the other hand, in the 3D version, we compute the Fourier Seires on the x, y and z components of the function and then stitch them back together to show the results.

To show the curve in 3D, we decided to implement Curve Tracing in Unity because it has natural support for 3 dimensional rendering. We simply have to write the script and Unity will render the drawing for us.

Unity has native support for C# scripting and therefore we will make use of C# programming language.

I have only explained the most important parts of the code here, as the code itself is pretty large and consists of several files. However, if one does want to look at the code, it is present on my GitHub account. I have added a link to the repository in the references.

5.2 Getting User Input

We take in user input through the **UserFunction.cs** script. This script is used by the **Driver.cs** script that runs the entire program.

The functions used in **UserFunction.cs** are -

1. The function *Start()* simply gets the LineRenderer of the GameObject it is attached to, which is used later to draw the user input function.
2. The *CheckForInput()* function checks if the mouse is clicked/ Enter key is pressed and performs the corresponding function (adding a point to the user-function, finishing the loop respectively)
3. The *ConvertListtoFunction()* function is called by *CheckForInput()* once the user is done entering the function. It converts the list of 3D points into a function.
4. The *SetPositionDistances()* function sets the "distances" of all the points along the curve starting from the first point.
5. The *UpdateLineRenderer()* function updates the Line Renderer by adding one point in the line and showing it on the screen.

The end result of inputting the user function is that the user can see the input he/she input as a set of straight lines connecting the given points. The function need not be planar and can be drawn anywhere in 3D. The only constraint is that it has to be a closed loop.

The function that is entered by the user will now be approximated by using Fourier Series.

5.3 Computing Fourier Coefficients and Calculating Fourier Series

Now that we have the function that we want to approximate, we want to find the fourier coefficients and Fourier Series for a given maximum frequency. For this we will use the **FourierSeries.cs** class. We have defined it as a static class as we will be using the functions provided therein as utility functions. The **FourierSeries.cs** script will not be attached to any GameObject and therefore need not inherit from Monobehaviour like other scripts in Unity.

It is important to note that both the naïve and FFT implementations are done in **FourierSeries.cs**, but the other scripts only use the FFT algorithm.

The **ApplyFourierSeries.cs** script will be attached to a GameObject and will be showing the Fourier Series approximations. The **Driver.cs** script will provide instances of this class with the function to approximate and the maximum frequency. **ApplyFourierSeries.cs** will use functions in the **FourierSeries.cs** script for the meat of the work.

The **ApplyFourierSeries.cs** script contains the following methods -

1. *CalculateFourierSeries()* This function computes the fourier coefficients and from them, the Fourier Series given the function and max frequency.
2. *FunctionUpdate()* Updates the drawing of the Fourier Series approximation by some time delta_t defined in the class. It also updates the location of the GameObject to be equal to the approximated value of the function at that time.
3. *UpdateLineRenderer()* Updates the line renderer attached to the GameObject. Used by *FunctionUpdate()*.

The **DrawCoefficients.cs** script is very similar to the **ApplyFourierSeries.cs** script. The difference is that **DrawCoefficients.cs** attempts to show how the individual contributions add up to give the final function approximation.

It does this using "Arrow" GameObjects. The position, scale, and orientation of the different arrows is calculated here.

5.4 The Driver Script

The **Driver.cs** script is the main script that runs all the other scripts. We can change the number of approximators, frequency range and the max_frequency for **DrawCoefficients.cs** from the Driver.

The responsibilities of the Driver are -

1. Create as many approximator GameObjects as required. These GameObjects have the **ApplyFourierSeries.cs** script attached to them.
2. Run **UserFunction.cs** to get the user input. Fourier Series must not be calculated until the function is ready.
3. After the function is ready, to call the necessary functions to get the Fourier Series computed for all approximators.
4. Every frame update all the approximators' location. Baisically to call their *FunctionUpdate()* methods.
5. To listen to keypresses that toggle the visibility of approximators on/off (Numeric keys are used for this and hence only a maximum of 10 approximators can be created).

5.5 Other Scripts Used

Some other scripts used and their descriptions are -

1. **CameraScript.cs** - This script controls the camera and allows the user to move in the 3D environment.
2. **Complex.cs** - A script that defines some functionaliy of complex numbers used in other scripts (eg. adding/ subtracting /multiplying/ dividing Complex numbers, creating rectangular form from Polar form etc.).

5.6 Results

When we run the program, we start with an empty environment. Here we can click multiple times in our environment to make our desired 3D function. The process looks like this -

Figure 5.1: Getting user input

Once the user enters the desired function, he/she presses "Enter". The program completes the "closed loop" and converts the list of points into a function. The approximation of functions then starts -

Here I show how the program runs on one input. The user function is given above.

The first approximation corresponds to using a maximum frequency of 4. It does a very poor job of approximating the function.

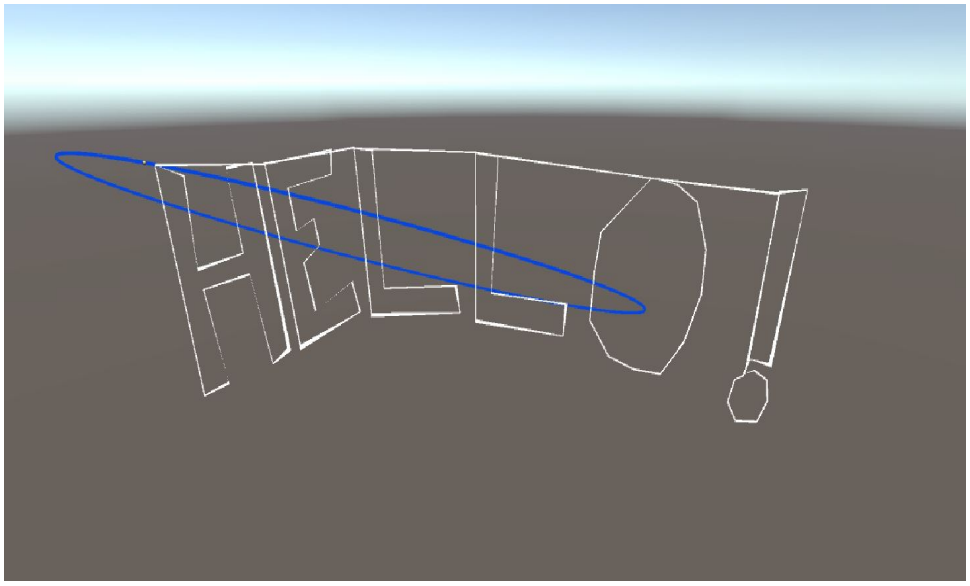


Figure 5.2: Approximation with max frequency = 4(Blue)

The next few approximations still perform poorly, but we can see that they slowly start to take the shape of the curve as the maximum frequency increases.

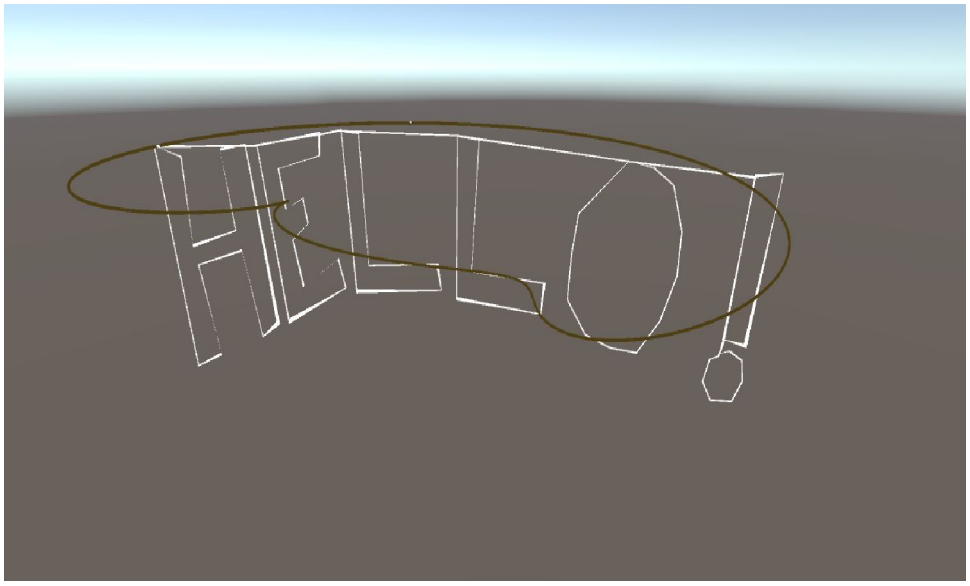


Figure 5.3: Approximation with max frequency = 8(Brown)

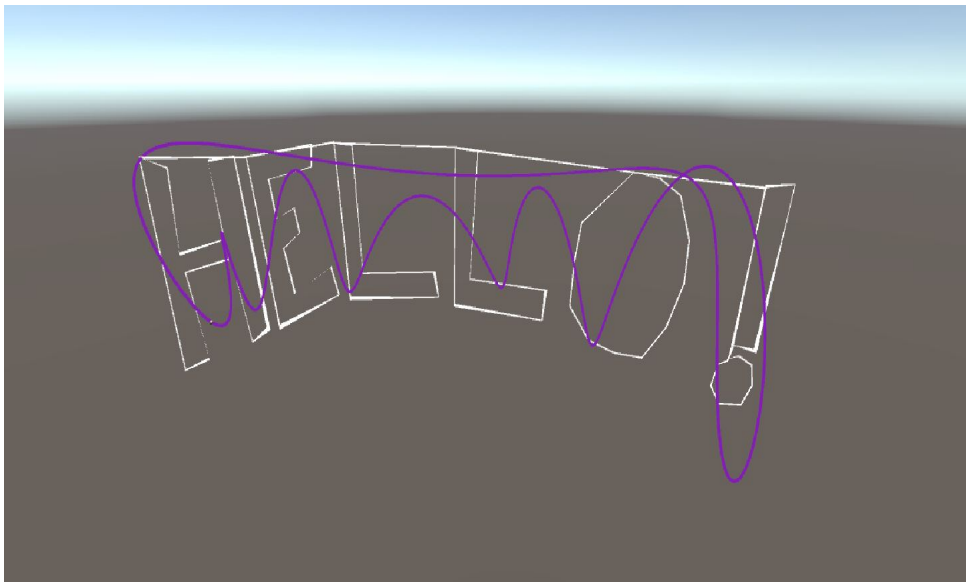


Figure 5.4: Approximation with max frequency = 16(Purple)

Figure 5.5: Approximation with max frequency = 32(Greenish-white)

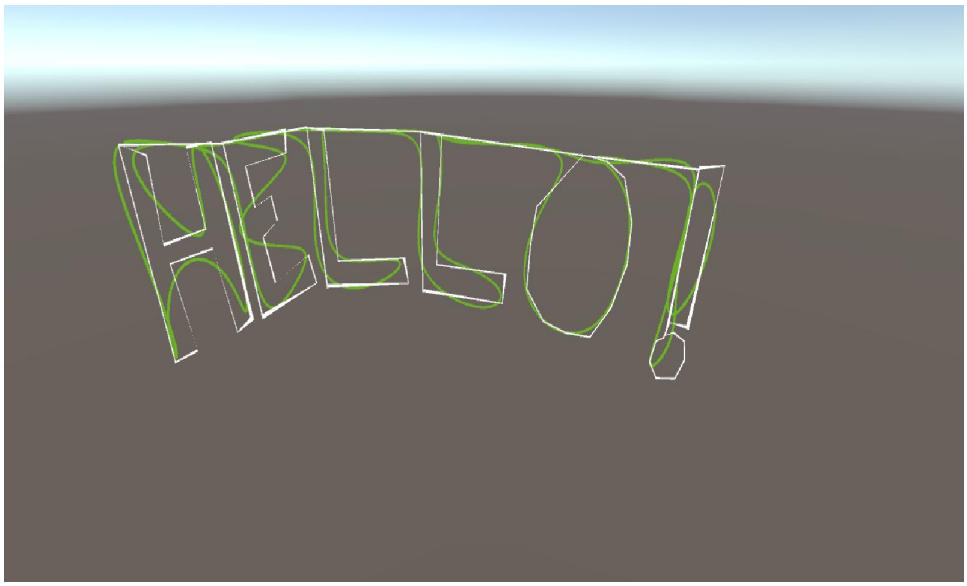


Figure 5.6: Approximation with max frequency = 64(Green)

In the following pictures, we can see that the shape of the approximations have formed very well. The further increase in frequency makes the approximations nearly indistinguishable from the original. One clear difference are the sharp turns in the original, which are not present in the approximation.

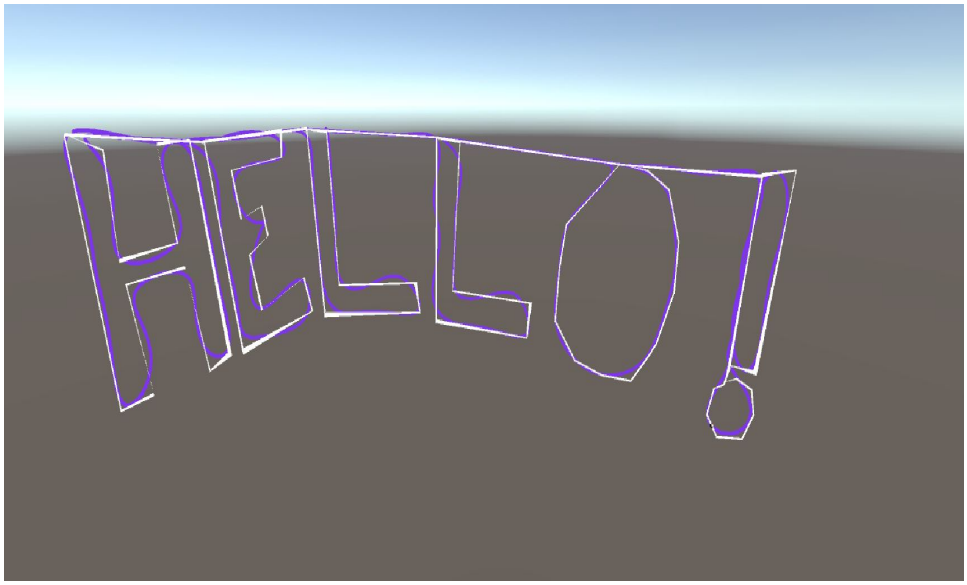


Figure 5.7: Approximation with max frequency = 128(Purple)

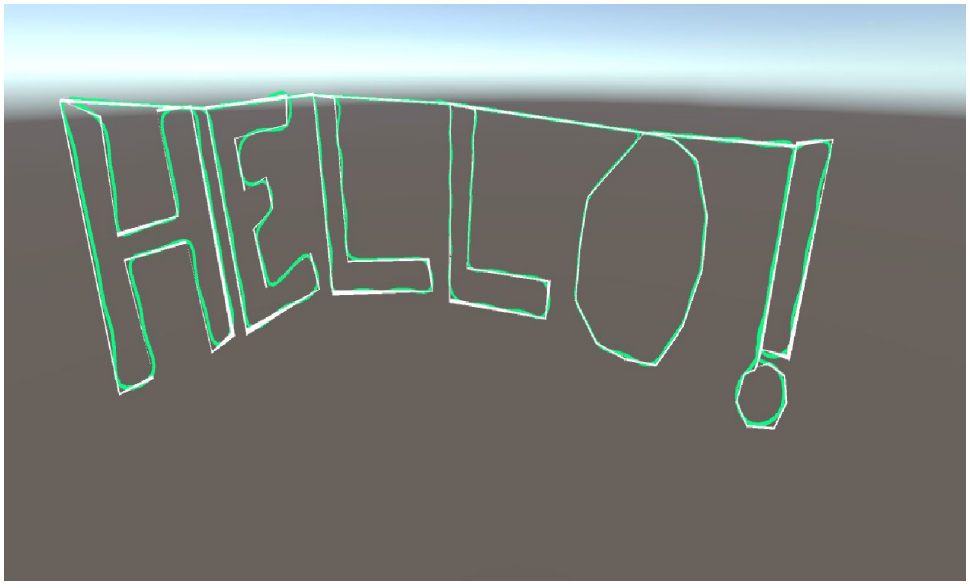


Figure 5.8: Approximation with max frequency = 256(Light Green)

The last frequency we will show both with and without the user function so that the differences can be clearly seen.

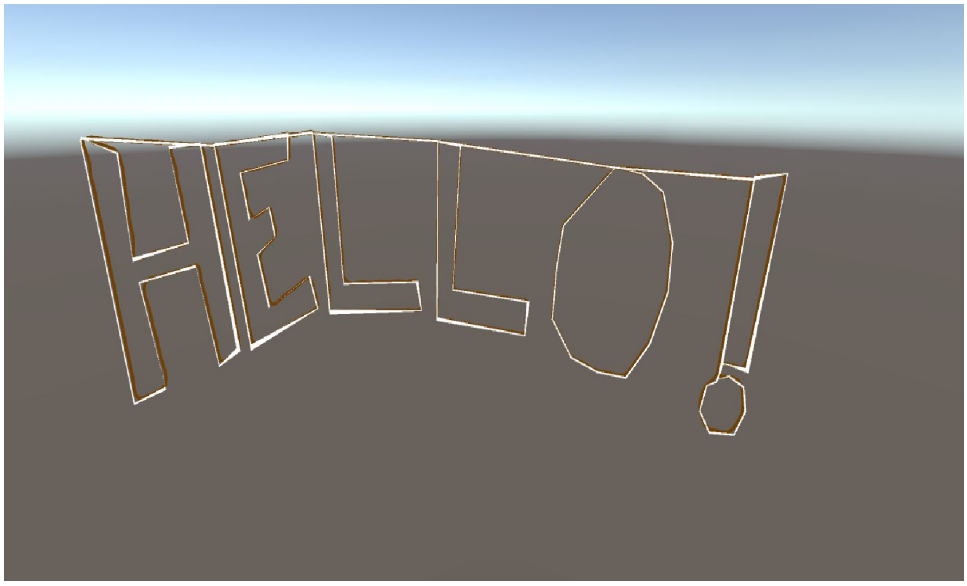


Figure 5.9: Approximation with max frequency = 512(Brown)

Figure 5.10: Approximation with max frequency = 512(Brown)

Chapter 6

Conclusion

Thus, we can conclude that Fourier Series can be effectively used to approximate any custom user-drawn periodic curve in n Dimensions. The results for n = 2 and n = 3 have been shown in this project.

As the maximum frequency used for approximation increases, the approximation improves. For very high frequencies, the approximations can be nearly indistinguishable from the original curves.

In higher dimensions (n >2) we can get approximations for the functions by doing component-wise Fourier Analysis. We break down an n-dimensional closed curve into n one-dimensional closed curves, get each component's Fourier Series and then patch them together before showing the results. Thus, strictly speaking, the time complexity would also depend on the number of dimensions of the curve.

The Fast Fourier Transform Algorithm(FFT) is effective at computing the fourier coefficients and is much faster(/efficient) in doing so. Therefore, we should use the FFT algorithm especially if we are working with high frequencies.

References

1. 3Blue1Brown video that inspired this project-
   https://www.youtube.com/watch?v=r6sGWTCMz2k&vl=en

2. Stanford University Lecture Series by Professor Brad Osgood
   https://www.youtube.com/watch?v=gZNm7L96pfY&list=PLB24BC7956EE040CD

3. Wikipedia articles for information on Joseph Fourier, Fourier Analysis
   https://en.wikipedia.org/wiki/Joseph_Fourier
   https://en.wikipedia.org/wiki/Fourier_analysis

4. The code for fft in C++. This helped me to check and improve my own implementation in C#.
   https://cp-algorithms.com/algebra/fft.html

5. My GitHub repository link for the code.
   https://github.com/Sharanya-Ranka/Fourier-Curve-Tracing