**Fourier series project**

Layan Jethwa

**Introduction**

Fourier series are a way to express a function as a sum of sine and cosine functions. They are particularly interesting to me as they have a wide range of applications, because they break down quite complex (in the descriptive sense) functions into a sum of simple oscillating functions. They therefore have a wide range of applications, from Joseph Fourier’s work on heat theory to sound waves and signal processing. They are used in many practical applications in daily life, such as noise cancellation, image and audio compression, speech recognition, and medical imaging.

The series is named after Joseph Fourier, who, through his work on heat transfer, showed that representing a function as a sum of sines and cosines could greatly simplify the problem. His discovery birthed the field of Fourier analysis, which is the study of how general functions can be represented as simpler trigonometric ones. The method used to do this is called the Fourier transform, which is an integral transform to describe the extent to which various frequencies are present in the inputted function.

Before Fourier, early ideas of decomposing a periodic function in this way date back to the 3rd century BC, where astronomers used a geometric model called the epicycle to explain the variations in planetary motion. The planets are modelled as moving in a small circle, the epicycle, which in turn moved in a larger circle, the deferent. Epicycles fundamentally follow the same idea as the Fourier series - a complicated periodic function is broken down into circles (which can be represented parametrically as . The Fourier series itself can also be represented as epicycles - each sine and cosine pair can be represented as a rotating circle, and the sum of the motions of these circles will trace out the curve.

The goals of this project are to be able to explain the Fourier series and the transform, to compute the Fourier coefficients manually for a few simple examples, to show how these correspond to epicycles, and to demonstrate with code how epicycles can approximate curves.

**Mathematical background**

The Fourier series is built off the orthogonality of sine and cosine, which means that the dot product of the two functions is equal to 0. This orthogonality leads to 3 results (these are derived from the addition formulae):

We can also see two results from the periodic nature of the functions - as they both oscillate equally around the x-axis, the areas cancel out over a full period:

For cos we had to consider the case where m=0, as the integrand is then equal to 1; as , we do not have to worry about any special cases here.

For the complex valued series, it is useful to have a brief look into Euler’s formula:

This is quite a famous result, especially the special case where . We can prove it by using the Maclaurin series formula (a special case of the Taylor series where a=0, which expresses a function as an infinite sum of terms in terms of the derivative of the function at a specific point):

The here means the nth derivative of . If we apply the formula to , as the derivative is self-preserving, we can see that for all n. This gives us:

We also need to know the Maclaurin series for and . By computing the derivatives at , we can see that the derivatives for each are periodic:

We can see that the series for sine will only be the odd values of n, and the one for cosine will be the even values of n - this also ties in with Fourier’s theorem that every function can be expressed as a sum of sines and cosines.

The Maclaurin series for sine thus follows, which we can write in two ways:

And similarly for cosine:

Now, we can expand using the Maclaurin series that we just derived:

We can then split it into odd and even terms:

This is clearly the sum of the two Maclaurin series we just calculated. This therefore gives us the result of Euler’s formula:

We can derive further expressions for and from this formula. Firstly, we replace with , effectively creating the complex conjugate of the original formula:

Using the symmetries of sine and cosine, we get:

Summing the two equations gives:

Subtracting the second equation from the first gives:

There is also a version of the orthogonality results from above for exponentials, which, due to Euler’s formula, essentially contains the orthogonality results of both sine and cosine. We start with the integral of the product of two exponentials over 1 period:

Now we can consider two cases. Firstly, we will consider the one where :

Secondly, we will consider the one where , keeping in mind our earlier definition of as the angular frequency:

Therefore, we have the orthogonality result:

This is the continuous result, and we can also derive the discrete result from it using a Riemann sum. This is the same calculation as I will demonstrate later in the section about the discrete series, so I will just give the result below:

**Real Fourier series**

**Definition and derivation of coefficients**

The real valued Fourier series, also known as the trigonometric Fourier series, is defined as:

The period of the function here is . It is helpful to simplify the equations slightly by introducing a new variable for the angular frequency, which is equal to . We can therefore rewrite the series as:

If we assume that all periodic functions can be written as an infinite sum of sines and cosines, then , and are the Fourier coefficients for this trigonometric series, and they can be derived from the formula for the series (which just formalises this assumption). This assumption is valid due to the orthogonality mentioned above.

The Fourier coefficients (for ) can be derived using the results from the orthogonality. We will firstly look at the cosine coefficients, and start by multiplying both sides by :

Integrate both sides over one period:

Apply results from orthogonality (because of the sum, n is always greater than 1, so the integral with double cos will equal 0 in every case apart from that in which it is equal to half the period, and all the cases are then summed), and from the periodicity:

This then gives the result of:

We can derive the sine coefficients similarly:

Finally, we can derive the constant term from the original series. We start by integrating both sides (again over one period):

Now we apply the results from periodicity:

Evaluate the RHS integral:

And we get the result of:

We can see that this result is just the mean of the function across one period - it is the whole area divided by the period length.

**Worked example**

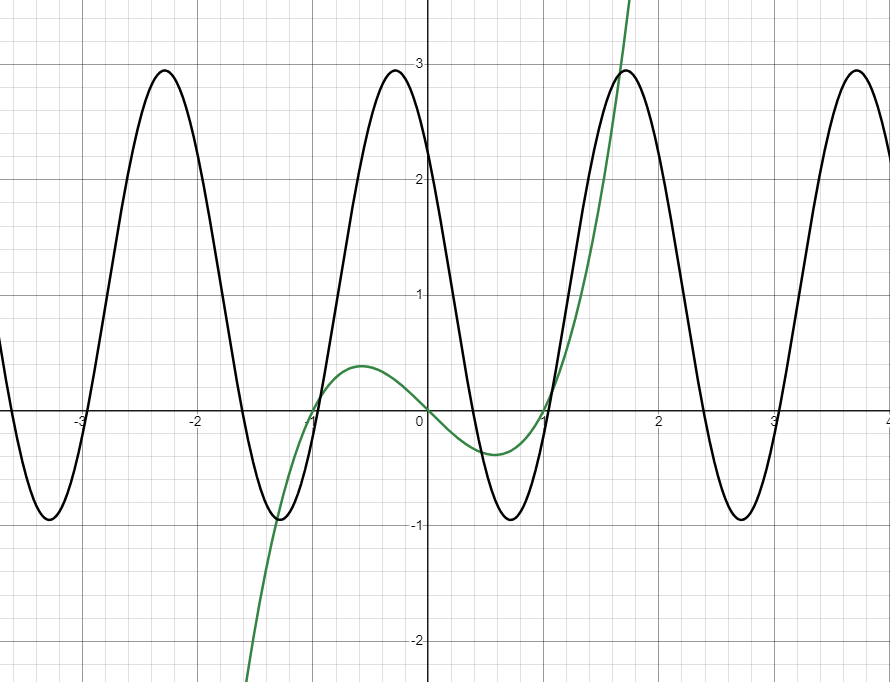
We can use the example of with a period with interval and thus a period of 2, which is not too difficult to integrate manually (just some by parts). Firstly, we calculate the constant coefficient:

Next, we will calculate the cosine coefficient, keeping in mind that integer multiples of will always equal 0 when inputted to sine, and 1 with cosine:

And similarly, the coefficient for sine:

Now that we have computed all the coefficients, we can graph the approximation of the curve. Obviously in the case of this example, the curve itself graphs perfectly well without needing the series approximation, but this is merely the groundwork for the discrete series and transform, which is useful in applications such as signal processing.

Below, we can see the curve in green, and the approximation in black, initially with 1 term of the sum. I have then showed the approximation with 3, 8, 15, 50 and 100 terms. We can notice the Gibbs phenomenon just to the right of the origin - this is where the approximation oscillates rapidly at a jump discontinuity. The series converges to the section of the curve that we have approximated.



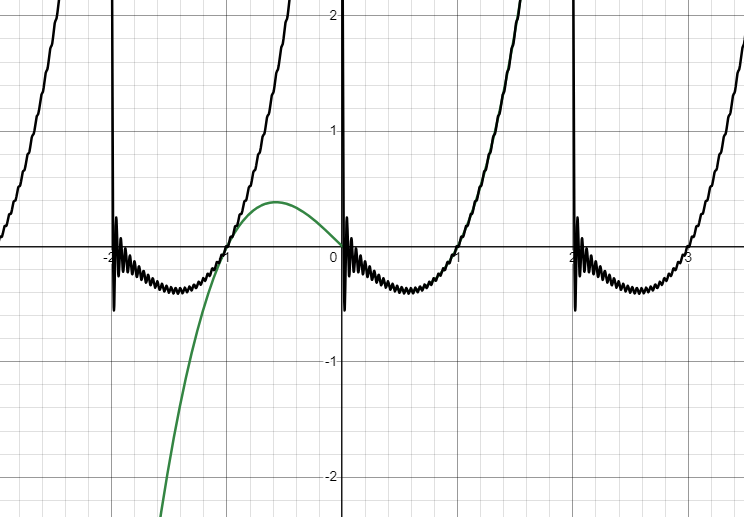
A graph with lines and a green line

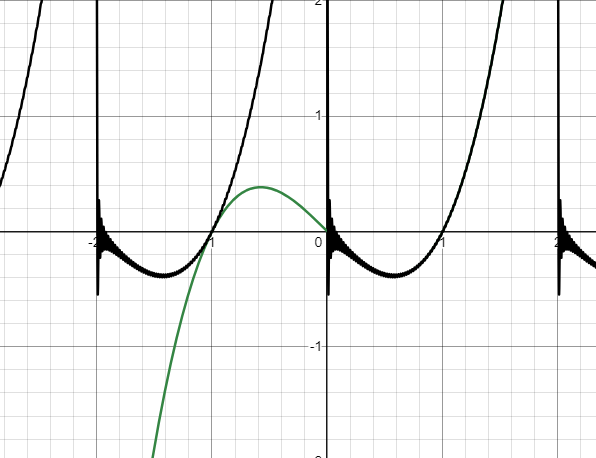
AI-generated content may be incorrect.

A graph on a grid

AI-generated content may be incorrect.







**Complex Fourier series**

**Definition and derivation of coefficients**

Now that we have analysed the real valued series, the next logical step is the complex valued one, which is also known as the exponential Fourier series. The idea of the complex valued series is that we combine the sine and cosine terms through Euler’s identity, which makes the formula more compact, easier to manipulate algebraically, and forms the basis for the continuous Fourier transform.

We can start by rewriting the real valued series in terms of exponentials (from the results we derived from Euler’s formula previously):

Next, we can collect the exponential terms:

We can introduce new variables for the complex coefficients to simplify the equation, and then collapse the series into a single summation (All values of n above 0 are the first exponential, all values below 0 are the second exponential, and when n = 0, the exponential = 1, so the constant coefficient can be defined as ):

This gives us the general formula for the complex-valued continuous Fourier series. We can derive the coefficient using the orthogonal results for exponentials that we derived earlier. We start by multiplying by and then integrating over one period to get the orthogonality results:

Apply the orthogonality results to get the expression for the coefficient :

**Worked example**

We will again use the example of with a period with interval and thus a period of 2. Remember that, due to Euler’s formula, the complex exponential is periodic across , and thus is equal to 1 for all integer multiples of the period. Firstly, we will calculate the Fourier coefficient, taking the special case where n = 0:

Next, we will compute the general coefficient for all other values of n:

Now that we have computed the coefficients, we can graph the function. The sum is symmetric about n = 0, with higher absolute values of n making the approximation more accurate. As we combined the 2 conjugate terms to form the complex series, it converges exactly half as quickly as the real series. We can see from the graph that the calculations are exactly the same, just represented differently. Due to the special case where n = 0, we must first take this term (the same as the constant term in the real series), and then add both conjugates separately (so that the iterations can be increased), so the inputted graph looks like this:

Below are the results for m = 1, 5, 15 and 30 (Desmos gets a bit slower to compute after this, but it is unsurprisingly the same output as the real series):

A graph of a function

AI-generated content may be incorrect.

A graph of a function

AI-generated content may be incorrect.

A graph with a line graph

AI-generated content may be incorrect.

A graph of a function

AI-generated content may be incorrect.

**Discrete Fourier series**

**Definition and derivation of coefficients**

The discrete Fourier series (DFS), also, confusingly, known as the discrete Fourier transform (DFT), is a series that takes an input of evenly spaced samples of a function, and reconstructs the function in the form of Fourier coefficients. As the input is discrete, reconstructing the function will give the exact same output, so we can think of the DFS as having two stages - the analysis stage and the synthesis stage. The analysis stage gives us the coefficients, each of which is a weight (complex number), encoding both amplitude and phase argument. This means that it tells us the amplitude of the frequency component for each frequency, as well as the starting angle (phase shift) of that frequency component. The synthesis stage just reconstructs the initial set of points from the coefficients. At first glance, the synthesis seems pointless, but we can use it for things such as audio compression - samples where the coefficients have very low values can be ignored when reconstructing the waveform. The coefficients are helpful for analysis of the waveform, for example, frequencies can be modified directly to filter the wave and then be reconstructed back into a waveform.

We can derive the DFS from the CTFS by sampling the signal uniformly, calling each sample , where n is the sample number, N is the total number of samples, and L is the period as previously. It is a helpful simplification to define for the Riemann sum approximation. We start from the CTFS coefficient formula to get the analysis stage of the DFS, approximating the integral as a Riemann sum to convert the continuous function into a discrete one (in the last step, because we have discretized the formula, it becomes exact):

The synthesis formula is shown below, and we can prove that it works by simplifying it down using the discrete version of the orthogonality result given earlier and our analysis formula:

This proves that the synthesis formula does in fact work. There is a faster algorithm to compute the DFS, called the FFT (Fast Fourier Transform), but it gives exactly the same output and is just a way of optimizing the number of calculations, so it is unnecessary to talk about.

**Epicycles**

Now that we have derived the DFT, we can use it to visualize the Fourier series through epicycles. The aim is to take an input of discrete points and then visualize the interpolation of the points using epicycles (a series of rotating circles as previously mentioned). Each coefficient for each sample will give an epicycle, with the radius being the modulus, the angular speed being radians per step, and the initial phase being the argument of the coefficient (the coefficients will be complex). To draw the points, we can take each epicycle as a vector and draw the next one at the tip of the previous and then draw a circle around them and rotate them to draw the points at the tip of the final vector/circle.

I made a simple visualisation of this in Python, which consists of computing the coefficients with the analysis formula that we derived above, then interpreting these as vector quantities, and then drawing the vectors and circles around them to show the epicycles. When the angular speed is applied to the vectors, they form the inputted set of points, interpolating between them. I have created a rudimentary system whereby the user can draw an image (as the series is periodic this works best for closed curves so that there are no discontinuity jumps in the interpolation), and the image is then reconstructed with the DFS and visualised as epicycles.

I have also added a slider to change the speed of the rendering, as well as one to change the resolution - this effectively changes the number of coefficients that are used, or the number of samples taken. It is quite interesting to play around with this, as we can see that there is only a definitive drop in quality as the percentage resolution gets quite low - this demonstrates the applications of the Fourier transform in image and audio compression. The visualisation also demonstrates its use in signal processing - we have effectively converted a set of discrete points into a waveform. The quality slider works by sorting the vectors in terms of magnitude, and prioritizing those with the highest magnitude - this is similar to audio processing, where frequencies can be isolated and removed. For example, a frequency spike could be an unwanted high-pitched noise in the sample - by converting a set of points to Fourier coefficients, we move them from the time domain to the frequency domain, making audio analysis and filtering in this way possible.

Below are some screenshots of the program running, with different drawings and different resolutions.

A screenshot of a video

AI-generated content may be incorrect.

A screenshot of a video game

AI-generated content may be incorrect.

A screenshot of a computer screen

AI-generated content may be incorrect.

A screenshot of a computer screen

AI-generated content may be incorrect.

A screenshot of a video game

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a video game

AI-generated content may be incorrect.

A screenshot of a video game

AI-generated content may be incorrect.

A screenshot of a video game

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a video

AI-generated content may be incorrect.

A screenshot of a video game

AI-generated content may be incorrect.

A screenshot of a video game

AI-generated content may be incorrect.

A screenshot of a video game

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

**Reflection**

This project overall has been extremely interesting, and it was really cool to put theory into practice and see the series in action. I love the beauty of the series - it is fairly simple, but can effectively encode any curve, equation or set of points in a set of coefficients.

It was amazing to add sliders to the program and see real-world applications of the series, especially with regard to compression. This project also ties into neural networks and AI, having fundamentally the same building blocks - all the data can be encoded by the coefficients, which are just complex numbers. Neural networks work the same way, using weights, except they are higher dimensional. I also saw first-hand how the Fourier series can be applied to signal processing and audio analysis and filtering.

I now have a much greater understanding and appreciation of this field of mathematics and have learnt a lot from this project. I have tried to explain most of the concepts and equations presented; however, I have left out one major part - the proof of Fourier’s theorem. This is because I don’t fully understand it as of now - it is to do with looking at it in a vector space, and proving properties based on this. It has close links with linear algebra and is something that I would like to explore further in the future.

I hope you find this as fascinating as I have, and maybe even that it inspires you to learn more about this incredible branch of maths. The YouTube channel 3Blue1Brown has some very interesting explainer videos on the topic, and is where I got the idea of rotating epicycles drawing an image from.