UNIVERSITY OF LINCOLN

SCHOOL OF MATHEMATICS AND PHYSICS

Analysis of Shapes in Nature Using Advanced Mathematical and Computing Techniques

Timothy Smith

25796944

Supervised by Dr Danilo Roccatano

April 24, 2024

Scientific Report in the 3rd Year Module MTH3009 Mathematics Project

Abstract

Placeholder abstract text

Contents

1 Introduction

2 Theoretical Background

2.1 The Elliptic Fourier Transform

The Fourier transform is a way of finding constituent frequencies within a function on the real domain. The Fourier transform extends the concept of the Fourier series (which openerates on a bounded interval) to the real domain. That is to say, the Fourier Series operations on a periodic function with period P and decomposes it into a series of sinusoidal waves, while the Fourier transform does the same but for any complex valued function $f(x)$.

[Get references from book from differential equations module]

The elliptic Fourier transform extends this concept to a finite series of points on the (x, y) plane. This is useful for practical applications in computing because one of the most typical ways to model a two-dimensional curve in a computer program is just that, a finite series of points on the (x, y) plane.

The way that the elliptic Fourier transform works is by taking said series of points and assuming that they are connected by straight lines, and modelling x and y coordinates as periodic functions of a parameter t, such that $x = x(t)$ and $y = y(t)$. Importantly, $x(t)$ and $y(t)$ have the same period T.

2.1.1 The Fourier Series

The Fourier series (1) lets you take a single repeating unit of a periodic function and use this to generate an infinite sum of sinusoidal functions that converge to it. This is useful because trigonmetric functions have mathematical properties that not all functions posess, allowing you to work with them nicely in more situations.

$$s(x) \sim A_0 + \sum_{n=1}^{\infty}\left(A_n \cos\left(\frac{2\pi n x}{P}\right) + B_n \sin\left(\frac{2\pi n x}{P}\right)\right) \tag{1}$$

The ~ symbol indicates that the series doesn't strictly converge in all cases.

The coefficients A_0, A_n and B_n are given by (2).

$$\begin{aligned} A_0 &= \frac{1}{P}\int_{-\frac{P}{2}}^{\frac{P}{2}} s(x)\,dx \\ A_n &= \frac{1}{P}\int_{-\frac{P}{2}}^{\frac{P}{2}} s(x)\cos\left(\frac{2\pi n x}{P}\right)dx \\ B_n &= \frac{1}{P}\int_{-\frac{P}{2}}^{\frac{P}{2}} s(x)\sin\left(\frac{2\pi n x}{P}\right)dx \end{aligned} \tag{2}$$

Here A_0 is a constant, and A_n and B_n are functions of n. In fact, since both the denominator of the fraction and the range of the integral for A_0 are the same, P, A_0 is simply the average around which the function oscillates.

A good intuition for why the Fourier series works is that the integral of the product of a function and a sinusoid is a measure of how well that function *matches* the sinusoid. For example, the sin function perfectly matches itself since $\sin^2 x$ has a positive average, while the sin and cos functions are a perfect anti-match since $\sin x \cos x$ simplifies to $\frac{\sin 2x}{2}$ which averages around 0.

2.1.2 Exponential Form

The concept of the Fourier series can actually be extended to the complex numbers $\mathbb{C}$.

This isn't terribly surprising, since there is a nice connection between the complex numbers and trigonometric functions given by Euler's formula (3).

$$e^{ix} = \cos x + i \sin x \tag{3}$$

Thus if we take A_n and B_n from (2) we can let a new variable

$$c_n = A_n + iB_n$$

and we can get

$$s(x) \sim A_0 + \sum_{n=1}^{\infty} c_n e^{i\frac{2\pi n x}{P}} \tag{4}$$

2.1.3 The Fourier Transform

The transform function $(S(t))$ for frequency f is given by (5).

$$S(t) = \int_{-\infty}^{\infty} s(t) \cdot \mathrm{e}^{-i2\pi f t} dt \tag{5}$$

2.1.4 Elliptical Fourier Analysis

The seminal paper on this topic is [1].

For:

$$\begin{aligned} x(t) &= A_0 + \sum_{n=1}^{\infty} \left[a_n \cos \frac{2n\pi t}{T} + b_n \sin \frac{2n\pi t}{T} \right] \\ y(t) &= C_0 + \sum_{n=1}^{\infty} \left[c_n \cos \frac{2n\pi t}{T} + d_n \sin \frac{2n\pi t}{T} \right] \end{aligned} \tag{6}$$

This paper gives the coefficients (for a discrete set of points that form a closed contour):

$$\begin{aligned}
a_n &= \frac{T}{2n^2\pi^2}\sum_{p=1}^{K}\left[\cos\frac{2n\pi t_p}{T} - \cos\frac{2n\pi t_{p-1}}{T}\right]\\
b_n &= \frac{T}{2n^2\pi^2}\sum_{p=1}^{K}\left[\sin\frac{2n\pi t_p}{T} - \sin\frac{2n\pi t_{p-1}}{T}\right]\\
c_n &= \frac{T}{2n^2\pi^2}\sum_{p=1}^{K}\left[\cos\frac{2n\pi t_p}{T} - \cos\frac{2n\pi t_{p-1}}{T}\right]\\
d_n &= \frac{T}{2n^2\pi^2}\sum_{p=1}^{K}\left[\sin\frac{2n\pi t_p}{T} - \sin\frac{2n\pi t_{p-1}}{T}\right]
\end{aligned} \tag{7}$$

It also gives the centres:

$$\begin{aligned}
A_0 &= \frac{1}{T}\sum_{p=1}^{K}\left[\frac{\Delta x_p}{2\Delta t_p}\left(t_p^2 - t_{p-1}^2\right) + \xi_p\left(t_p - t_{p-1}\right)\right]\\
C_0 &= \frac{1}{T}\sum_{p=1}^{K}\left[\frac{\Delta y_p}{2\Delta t_p}\left(t_p^2 - t_{p-1}^2\right) + \delta_p\left(t_p - t_{p-1}\right)\right]
\end{aligned} \tag{8}$$

Where:

$$\begin{aligned}
\xi_p &= \sum_{j=1}^{p-1}\Delta x_j - \frac{\Delta x_p}{\Delta t_p}\sum_{j=1}^{p-1}\Delta t_j\\
\delta_p &= \sum_{j=1}^{p-1}\Delta y_j - \frac{\Delta y_p}{\Delta t_p}\sum_{j=1}^{p-1}\Delta t_j
\end{aligned} \tag{9}$$

You can see this implemented in graph_area.cpp.

2.2 Processing the Images

In order to find shapes in our images for the purposes of analysis, we will need to detect edges. For this we will use the Canny Edge detection method, developed by John F. Canny.

2.2.1 Convolution Kernels

The most basic aspect of edge detection is the convolution kernel. This is an $M \times M : \{M = 2n+1, n \in \mathbb{N}\}$ matrix (an odd sided square matrix). Odd side lengths allow the kernel to be centered at each pixel. If we let the function $f(x,y)$ represent the original image, $g(x,y)$ represent the convolved image and ω represent the kernel, then:

$$g(x,y) = \sum_{i=-n}^{n}\sum_{j=-n}^{n}\omega(i,j)f(x-i,y-j) \tag{10}$$

Notice that the coordinates of the kernel are not 1 to M as with a traditional matrix, but rather $-n$ to n.

The effect of this on an image will be to make each pixel of a convolved image a function of the surrounding pixels. The simplest convolution matrix is the identity convolution matrix, (11).

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \tag{11}$$

More examples include edge detection kernels like (12) and (13).

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix} \tag{12}$$

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix} \tag{13}$$

2.2.2 Active Contours

2.2.3 Canny Edge Detection

3 Proposed Methodology

4 Project Results

5 Conclusion

6 Acknowledgements

7 Bibliography

References

[1] F. P. Kuhl and C. R. Giardina, “Elliptic fourier features of a closed contour,” *Computer Graphics and Image Processing*, vol. 18, no. 3, pp. 236–258, 1982.

8 Appendix

Note: These are unlikely to be in the final report, they are simply there as an example appendix for now.

graph_area.cpp

```
#include <cairomm/context.h>
#include <cmath>
#include <numbers>
#include <algorithm>
#include <iostream>
#include "graph_area.h"
#include "elliptic_fourier_analyser.h"
#include "elliptic_fourier_curve.h"

using std::sin;
using std::cos;
using std::min;
using std::max;
using std::numbers::pi;
using std::cout;
using std::endl;
using std::isnan;
using std::vector;
using std::array;

GraphArea::GraphArea()
{
    set_draw_func(sigc::mem_fun(*this, &GraphArea::draw));
}
```

```
GraphArea::~GraphArea()
{
}

// This is a method of an area and the only information
// we have about said area is its width and height
void GraphArea::draw(
    const Cairo::RefPtr<Cairo::Context>& cr,
    const int width,
    const int height
) {
    vector<array<double, 2>> coords = {
        {-0.5, 0},
        {-0.5, 0.5},
        {0.5, 0.5},
        {0.5, 0},
        {0.375, -0.125},
        {0.375, -0.5},
        {0.125, -0.5},
        {0.125, -0.375},
        {0, -0.5},
    };
    draw_discrete(cr, width, height, coords);
    EllipticFourierAnalyser analyser(coords);
    for(int i = 1; i <= 10; i++)
    {
        auto curve = analyser.analyse(i);
        auto curve_shape = curve.get_shape(1000);
        draw_continuous(cr, width, height, curve_shape);
    }
}

void GraphArea::draw_discrete(
    const Cairo::RefPtr<Cairo::Context>& cr,
    const int width,
    const int height,
    vector<array<double, 2>> coords
) {
    cr->set_source_rgb(0, 0, 0); // black
    int cx = width / 2;
    int cy = height / 2;
    int r = min(width, height) / 2;
    cr->move_to(cx + coords[0][0] * r, cy + coords[0][1] * r);
    double x, y;
    for(int i = 0; i < coords.size(); i++) {
        x = coords[i][0];
        y = coords[i][1];
        cr->line_to(cx + x * r, cy + y * r);
    }
    cr->line_to(cx + coords[0][0] * r, cy + coords[0][1] * r);
    cr->stroke();
    cr->set_source_rgb(1, 0.5, 0); // orange
    for(int i = 0; i < coords.size(); i++) {
        x = coords[i][0];
        y = coords[i][1];
        cr->arc(cx + x * r, cy + y * r, 10,
                0, 2 * pi);
        cr->fill();
    }
}

void GraphArea::draw_continuous(
    const Cairo::RefPtr<Cairo::Context>& cr,
    const int width,
    const int height,
    vector<array<double, 2>> coords
) {
    cr->set_source_rgb(0, 0, 1); // blue
    int cx = width / 2;
    int cy = height / 2;
    int r = min(width, height) / 2;
    cr->move_to(cx + coords[0][0] * r, cy + coords[0][1] * r);
    double x, y;
    for(int i = 1; i < coords.size(); i++) {
        x = coords[i][0];
        y = coords[i][1];
        if(isnan(x) || isnan(y)) continue;
        cr->line_to(cx + x * r, cy + y * r);
    }
    cr->stroke();
}
```