UNIVERSITY OF LINCOLN

SCHOOL OF MATHEMATICS AND PHYSICS

Classification of Biological Organisms from Images Using Advanced Mathematical Techniques

Timothy Smith

25796944

Supervised by Dr Danilo Roccatano

March 11, 2024

Scientific Report in the 3rd Year Module MTH3009 Mathematics Project

Abstract

Contents

1 Introduction

2 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.

2.1 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 $\sim$ symbol indicates that the series doesn't strictly converge in all cases.

Firstly we have a set of numbers, A_0, A_n and B_n:

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

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.

2.2 The Fourier Transform

The transform function ($S(t)$) for frequency f is given by (3).

$$S(t) = \int_{-\infty}^{\infty} s(t) \cdot \mathrm{e}^{-i2\pi f t}\,dt \tag{3}$$

2.3 Fourier Analysis

2.4 Elliptical Fourier Analysis

The seminal paper on this topic is [1].

For:

$$\begin{aligned} x(t) &= A_0 + \sum_{n=1}^{\infty} \left[a_n \cos \frac{2n\pi t}{T} + b_n \sin \frac{2n\pi t}{T} \right] \\ y(t) &= C_0 + \sum_{n=1}^{\infty} \left[c_n \cos \frac{2n\pi t}{T} + d_n \sin \frac{2n\pi t}{T} \right] \end{aligned} \tag{4}$$

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

$$\begin{aligned} a_n &= \frac{T}{2n^2\pi^2} \sum_{p=1}^{K} \left[\cos \frac{2n\pi t_p}{T} - \cos \frac{2n\pi t_{p-1}}{T} \right] \\ b_n &= \frac{T}{2n^2\pi^2} \sum_{p=1}^{K} \left[\sin \frac{2n\pi t_p}{T} - \sin \frac{2n\pi t_{p-1}}{T} \right] \\ c_n &= \frac{T}{2n^2\pi^2} \sum_{p=1}^{K} \left[\cos \frac{2n\pi t_p}{T} - \cos \frac{2n\pi t_{p-1}}{T} \right] \\ d_n &= \frac{T}{2n^2\pi^2} \sum_{p=1}^{K} \left[\sin \frac{2n\pi t_p}{T} - \sin \frac{2n\pi t_{p-1}}{T} \right] \end{aligned} \tag{5}$$

It also gives the centres:

$$\begin{aligned} A_0 &= \frac{1}{T} \sum_{p=1}^{K} \left[\frac{\Delta x_p}{2\Delta t_p} \left(t_p^2 - t_{p-1}^2 \right) + \xi_p \left(t_p - t_{p-1} \right) \right] \\ C_0 &= \frac{1}{T} \sum_{p=1}^{K} \left[\frac{\Delta y_p}{2\Delta t_p} \left(t_p^2 - t_{p-1}^2 \right) + \delta_p \left(t_p - t_{p-1} \right) \right] \end{aligned} \tag{6}$$

Where:

$$\begin{aligned} \xi_p &= \sum_{j=1}^{p-1} \Delta x_j - \frac{\Delta x_p}{\Delta t_p} \sum_{j=1}^{p-1} \Delta t_j \\ \delta_p &= \sum_{j=1}^{p-1} \Delta y_j - \frac{\Delta y_p}{\Delta t_p} \sum_{j=1}^{p-1} \Delta t_j \end{aligned} \tag{7}$$

You can see this implemented in graph_area.cpp.

3 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.

3.1 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:

$$g(x,y) = \sum_{i=-n}^{n} \sum_{j=-n}^{n} \omega(i,j) f(x-i, y-j) \tag{8}$$

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, (9).

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \tag{9}$$

More examples include edge detection kernels like (10) and (11).

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix} \tag{10}$$

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix} \tag{11}$$

3.2 Canny Edge Detection

4 Conclusion

5 Acknowledgements

6 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>
```

```cpp
#include "graph_area.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;

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 it's width and height
void GraphArea::draw(
    const Cairo::RefPtr<Cairo::Context>& cr,
    const int width,
    const int height
) {
    /*
    cr->set_source_rgb(0, 0, 0.5); // blue

    int i = 0;
    cr->move_to(i, height/2 + waveform(i/20.0) * height/4);
    for(i = 1; i <= width; i++)
    {
        cr->line_to(i, height/2 + waveform(i/20.0) * height/4);
    }
    //cr->fill();
```

```
cr->begin_new_path();
cr->set_source_rgb(0, 0, 0); // black
double theta = 0;
double main_radius = min(width, height) / 2;
double radius;
radius = main_radius/2 + (main_radius/2) * waveform(theta);
cr->move_to(width/2 + radius*sin(theta), height/2 + radius*cos(theta));
for(;theta <= 2 * pi; theta += 0.01)
{
    radius = main_radius/2 + (main_radius/2) * waveform(theta);
    cr->line_to(width/2 + radius*sin(theta), height/2 + radius*cos(theta));
}
//cr->fill();
*/
double x[] = {-0.5, -0.5, 0.5, 0.5,  0.375, 0.375, 0.125,  0.125,    0};
double y[] = {   0,  0.5, 0.5,   0, -0.125,  -0.5,  -0.5, -0.375, -0.5};
int32_t K = sizeof(x) / sizeof(x[0]);
draw_discrete(cr, width, height, K, x, y);
double t[K];
double delta_x[K];
double delta_y[K];
double delta_t[K];
double T;
generate_functions(x, y, K, t, delta_x, delta_y, delta_t, T);
double centre_y = centre(T, K, y, t, delta_y, delta_t);
//centre_y = -0.05;
double centre_x = centre(T, K, x, t, delta_x, delta_t);
int32_t N = 10;
double a[N];
double b[N];
double c[N];
double d[N];
for(int32_t n = 0; n < N; n++)
{
    a[n] = coefficient(n + 1, T, K, x, t, delta_x, delta_t, cos);
    b[n] = coefficient(n + 1, T, K, x, t, delta_x, delta_t, sin);
    c[n] = coefficient(n + 1, T, K, y, t, delta_y, delta_t, cos);
    d[n] = coefficient(n + 1, T, K, y, t, delta_y, delta_t, sin);
```

```
    }
    int32_t resolution = 1000;
    double transform_x[resolution];
    double transform_y[resolution];
    for(int32_t max_n = 1; max_n <= N; max_n++)
    {
        for(int32_t i = 0; i < resolution; i++)
        {
            double t_smooth = T * i / resolution;
            transform_x[i] = centre_x;
            transform_y[i] = centre_y;
            for(int32_t n = 1; n <= max_n; n++)
            {
                transform_x[i] +=
                    a[n - 1] * cos(2 * n * pi * t_smooth);
                transform_x[i] +=
                    b[n - 1] * sin(2 * n * pi * t_smooth);
                transform_y[i] +=
                    c[n - 1] * cos(2 * n * pi * t_smooth);
                transform_y[i] +=
                    d[n - 1] * sin(2 * n * pi * t_smooth);
            }
        }
        // blue
        cr->set_source_rgb(1 - (double)max_n/N, 1 - (double)max_n/N, 1);
        draw_continuous(cr, width, height, resolution, transform_x, transform_y);
    }
}

void GraphArea::generate_functions(
    double x[],
    double y[],
    int32_t K,
    double t[],
    double delta_x[],
    double delta_y[],
    double delta_t[],
    double &T
```

```
) {
    for(int32_t p = 0; p < K; p++)
    {
        int32_t p_minus_one = p == 0 ? K - 1 : p - 1;
        delta_x[p] = x[p] - x[p_minus_one];
        delta_y[p] = y[p] - y[p_minus_one];
        delta_t[p] = sqrt(delta_x[p] * delta_x[p] + delta_y[p] * delta_y[p]);
        t[p] = delta_t[p];
        if(p != 0)
        {
            t[p] += t[p-1];
        }
    }
    T = t[K - 1];
}

void GraphArea::draw_discrete(
    const Cairo::RefPtr<Cairo::Context>& cr,
    const int width,
    const int height,
    int32_t K,
    double x[],
    double y[]
) {
    cr->set_source_rgb(0, 0, 0); // black
    int centre_x = width / 2;
    int centre_y = height / 2;
    int radius = min(width, height) / 2;
    cr->move_to(centre_x + x[0] * radius, centre_y + y[0] * radius);
    for(int32_t i = 0; i < K; i++) {
        cr->line_to(centre_x + x[i] * radius, centre_y + y[i] * radius);
    }
    cr->line_to(centre_x + x[0] * radius, centre_y + y[0] * radius);
    cr->stroke();
    cr->set_source_rgb(1, 0.5, 0); // orange
    for(int32_t i = 0; i < K; i++) {
        cr->arc(centre_x + x[i] * radius, centre_y + y[i] * radius, 10, 0,
                2 * pi);
```

```
        cr->fill();
    }
}

void GraphArea::draw_continuous(
    const Cairo::RefPtr<Cairo::Context>& cr,
    const int width,
    const int height,
    int32_t K,
    double x[],
    double y[]
) {
    int centre_x = width / 2;
    int centre_y = height / 2;
    int radius = min(width, height) / 2;
    /*
    cr->set_source_rgb(1, 1, 0); // yellow
    for(int32_t i = 0; i < K; i++) {
        cr->arc(centre_x + x[i] * radius, centre_y + y[i] * radius, 3, 0, 2 * pi);
        cr->fill();
    }
    */
    cr->move_to(centre_x + x[0] * radius, centre_y + y[0] * radius);
    for(int32_t i = 0; i < K-1; i++) {
        if(isnan(x[i]) || isnan(y[i])) continue;
        if(isnan(x[i + 1]) || isnan(y[i + 1])) continue;
        cr->move_to(centre_x + x[i] * radius, centre_y + y[i] * radius);
        cr->line_to(centre_x + x[i + 1] * radius, centre_y + y[i + 1] * radius);
        cr->stroke();
    }
}

// This is a waveform method that we can use generally to return a value based on
// an input
double GraphArea::waveform(double x)
{
    double y = 0;
    double freqs[] = {10, 4};
```

```
    double mags[] = {0.1, 0.14};

    for(int i = 0; i < sizeof(freqs)/sizeof(freqs[0]); i++)
    {
        y += sin(x * freqs[i]) * mags[i];
    }

    return y;
}

double GraphArea::coefficient(
    int32_t n,
    double T,
    int32_t K,
    double xy[],
    double t[],
    double delta_xy[],
    double delta_t[],
    double (*trig_function)(double)
) {
    double sum = 0;
    for(int32_t p = 0; p < K; p++)
    {
        int32_t p_minus_one = p == 0 ? K - 1 : p - 1;
        double t_p_minus_one = p == 0 ? 0 : t[p - 1];
        sum += (delta_xy[p] / delta_t[p]) *
               (trig_function(2 * n * pi * t[p] / T) -
                trig_function(2 * n * pi * t_p_minus_one / T));
    }
    sum *= T / (2 * n * n * pi * pi);
    return sum;
}

double GraphArea::centre(
    double T,
    int32_t K,
    double xy[],
    double t[],
```

```
    double delta_xy[],
    double delta_t[]
) {
    double sum = 0;
    double sum_xi_delta = 0;
    for(int32_t p = 0; p < K; p++)
    {
        int32_t p_minus_one = p == 0 ? K - 1 : p - 1;
        //double t_p_minus_one = t[p_minus_one];
        double t_p_minus_one = p == 0 ? 0 : t[p - 1];
        sum += ((delta_xy[p] / (2 * delta_t[p])) *
               (t[p] * t[p] - t_p_minus_one * t_p_minus_one);
        double xi_delta = 0;
        /*
        for(int32_t j = 0; j < p; j++)
        {
            xi_delta += delta_xy[j];
            xi_delta -= (delta_xy[p] / delta_t[p]) * delta_t[j];
        }
        */
        xi_delta = xy[p_minus_one];
        xi_delta -= (delta_xy[p] / delta_t[p]) * t_p_minus_one;
        sum_xi_delta += xi_delta * (t[p] - t_p_minus_one);
    }
    return (sum + sum_xi_delta) / T;
}
```

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.