B1 Project Report:
Wireless Communication Channels

Edward Stevinson
Trinity College, Oxford
project @ https://github.com/Edward-Stevinson/B1-Project-Code

January 24, 2014

1 Introduction

This report summarises how I reached a continuous approximation (a density function) in the absence of a physics-based model but instead by using parametrised functions. This can then be used in order to analyse communication system performance and aid the overall project goal of characterising the performance of Wi-Fi receivers operating in indoor environments.

Included in the report are sections of my code and what I learnt about modelling techniques, in particular various aspects of orthogonal function theory and application, particularly as it applies to the Wi-Fi project data.

2 Orthogonality

2.1 Theory

Orthogonal basis functions allow us to describe functions as the sum of defined orders of the orthogonal basis functions with associated coefficients. Orthogonal basis functions are desirable when modelling or fitting to data as more coefficients can be calculated without changing the values of the prevously calculated coefficients.

$$f(x) = a_0 g_0(x) + a_1 g_1(x) + a_2 g_2(x) \ + ... \tag{1}$$

We are given a function `fs_orthog.m` which integrates Fourier basis functions over a period. We are asked to write some code, `fs_check_orthog.m`, which calls this function for $cos(mx) \times cos(nx)$ for $m = 0 \rightarrow 6$ and $n = 0 \rightarrow 6$ and stores the result in a 2D matrix:

```
coscos = zeros(7);
for m = 0:6;
    for n = 0:6
        coscos(m+1, n+1) = fs_orthog( 1, 1000, m, n, 'cc');
    end
end
```

This produces a diagonal matrix, showing them to be orthogonal. The same is then repeated for $sin(mx)sin(nx)$ which again produces a digonal matrix and for $sin(mx)cos(nx)$ which returns a null matrix.

Care must be taken, however, as when `nint = m = n` the command `fs_orthog(T,nint,m,n,'cc')` returns 1 and not 0.5. This occurs beacuse it is using the trapezoidal method to calculate the integral of $cos^2(m\pi)$, which is merely 1. This is similar to the case where `nint = m = n` with the command `fs_orthog(T,nint,m,n,'ss')` which returns 0, as $sin^2(m\pi) = 0$. This can be avoided by making it return `codeok = 0` at these values.

2.2 Fourier Series of a Triangular Function

To compute the Fourier series coefficients, we are given two pieces of code, `fs_Acoeff.m` and `fs_Bcoeff.m` which calculate the A_m and B_m coefficients respectively. These functions are called from a top level script, `fs_triangle.m` to plot

a graph of the Fourier series approximation of the function. The periodic function, a triangle wave, is generated by `fs_periodictriangle.m`.

I modified the above and created `fs_triangle_task.m` which generates the periodic function shown in Fig. 2.

I use the 'least-squares' criterion as an appropriate definition of a 'good description' of data, for this section and the rest of the project. I defined a decent representation to mean one that produces an error of less than 1×10^{-6}. The function `fs_error` calculates the number of terms needed to produce an error less than this value and plots it against a.

The B_m coefficients are zero as they refer to the sine terms and this function is even (technically a value is returned of order of magnitude 10^{-18} and fluctuate around zero due to the nature of trapezoidal numerical integration).

`fs_error.m`:

```
for i = 1:5
    E = 1;
    j = 0;
    A = a(1,i);
        while E > e
        E = fs_triangle_task(T,nint,j,A);
        j = j + 1;
        end
    N(1,i) = j - 1;
end
```

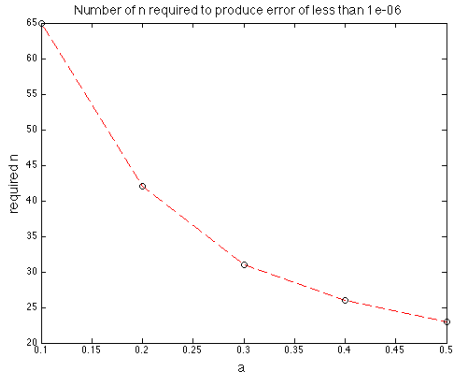

Figure 1: Number of terms required for a least squares error $< 1 \times 10^{-6}$

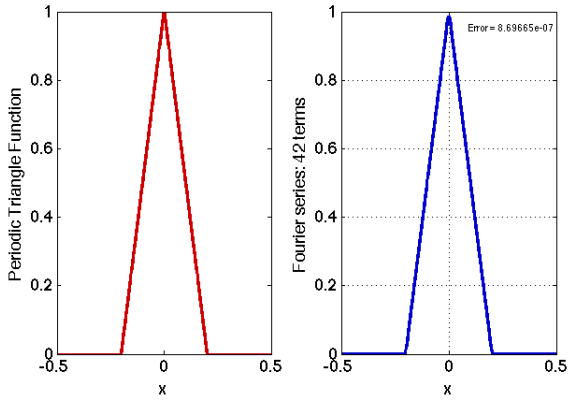

Figure 2: Fourier Representation

3 Orthogonal functions

3.1 The Gram-Schmidt Process

The Gram-Schmidt process is a method of orthogonalising a set of linearly independent functions. To do so, we take the first linearly independent function and orthogonalise the rest of the functions in relation to it by subtracting the projection of the previous functions onto the current function from the current function:

$$g_0(x) = v_0(x) \tag{2a}$$
$$g_1(x) = v_1(x) - e_{10}g_0(x) \tag{2b}$$
$$g_2(x) = v_2(x) - e_{20}g_0(x) - e_{21}g_1(x) \tag{2c}$$
$$\vdots$$

The e values are the coefficients representing the projections of function onto each other:

$$e_{10} = \frac{\langle v_1, g_0 \rangle}{\langle g_0, g_0 \rangle} \tag{3}$$

We are to perform Gram-Schmitt orthogonalisation on the linearly independent set of monomials:

$$v_0(x) = 1 \tag{4a}$$
$$v_1(x) = x \tag{4b}$$
$$v_2(x) = x^2 \tag{4c}$$
$$\vdots$$

with respect to the inner product:

$$\langle g_n, g_m \rangle = \int_0^\infty g_n(x) g_m(x) e^{-x} \, \mathrm{d}x$$

3.2 An Example of Gram-Schmidt

My code is run by calling a top level script, `gs_script.m`. The script calls on a number of functions which, in order, perform the folling tasks:

- `generate_v.m` generates a matrix, V, of the monomials.
- `gram_schmidt.m` performs Gram-Schmidt orthogonalisation upon the matrix V to produce a matrix of orthogonal functions, G, and a matrix of coefficients, E. The nested `for` loops cycle row by row through the matrix E, calculating and storing coefficient values $e_{i,j}$ in the column spaces before subtracting the projections of previous functions from G_j.

```
function [E, G] = gram_schmidt(x, n, V):
```

```
G = zeros(n+1, length(x));
E = zeros(n+1, n);
G(1,:) = V(1,:);      % Set G0(x) = V0(x)
for i = 1:n
    G(i+1,:) = V(i+1,:);
    for j = 1:i
         E(i+1,j) = innerproduct(x, V(i+1,:), G(j,:))...
         / innerproduct(x, G(j,:), G(j,:));        % eqn. 3
         G(i+1,:) = G(i+1,:) - E(i+1,j)*G(j,:);    % equations 2
    end
end
end
```

- `verify_orthog.m` verifies the orthogonality of G by showing that it is a diagonal matrix.

```
function [Z] = verify_orthog(G, n, x)
Z = zeros(n+1);
for i = 1:n+1
    for j = 1:n+1
        Z(i,j) = innerproduct(x, G(i,:),G(j,:));
    end
end
end
```

```
function [result] = innerproduct(x, a, b)
    result = trapz(x, a.*b.*exp(-x));
end
```

- `normalising_C.m` calculates a vector of normalising constants.
- `normalise_G.m` normalises G, to create $\tilde{G}$.
- `verify_orthog` is used again, but this time confirms the orthogonality of $\tilde{G}$.
- `Is_norm_G_orthon.m` then outsputs a yes/no answer depending on whether $\tilde{G}$ is orthonormal.

```
function [check] = verify_orthon(Y, n):
```

```
check = zeros(1);
I = eye(n+1);
P = norm(I-Y);
P(P<0.05) = 0;
if P == 0
    check = 'Yes';
else
    check = 'No';
```

- Finally, the script plots the orthonormal functions the orthonormal functions $\tilde{g}_0,\ \tilde{g}_1, \tilde{g}_2,\ ...$

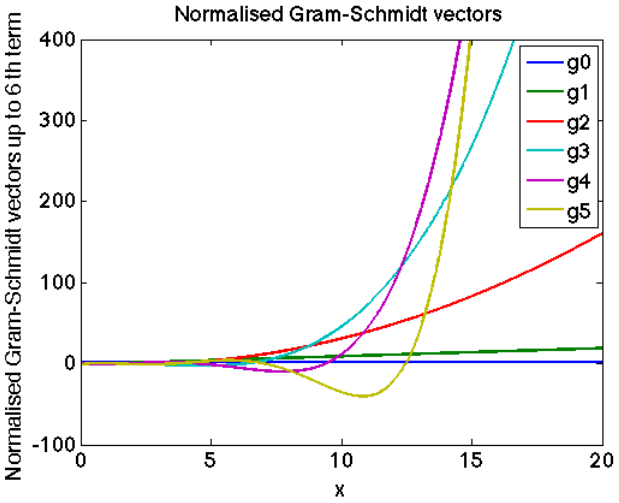

Figure 3: First six normalised functions

4 Laguerre

4.1 Laguerre Polynomials

Explicit forms of orthogonal functions are required to express the data sets as the weighted summation of a subset of orthogonal functions. The Laguerre polynomials are a set of such polynomials that are orthogonal with respect to the exponential weighting function and can be calucualted by the use of Rodrigues formula:

$$L_n^{(\alpha)}(x) = \frac{1}{n!}x^{-\alpha}e^x\frac{d^n}{dx^n}(x^{n+\alpha}e^{-x}),\ n \in \mathbb{N}, \alpha \in \mathbb{R} \tag{5}$$

Inspection of this equation leads to the realisation that the $x^{-\alpha}e^x$ terms cancel and so the Laguerre functions are merely just the coefficients multiplied by $\frac{1}{n!}$. The coefficients can be found by using Leibniz's differentiation rule, which I used in `lag_coeff.m`:

```
C = zeros(1,n);

for i = 0:n
    C(1,i+1) = nchoosek(n,i)*((-1)^(n-i))*(factorial(n + alpha)/factorial(n + alpha - i));
end
```

They can also be generated through the use of a recurrence relationship:

$$nL_n^{(a)}(x) = (2n + \alpha - 1 - x)L_{n-1}^{(\alpha)}(x) - (n + \alpha - 1)L_{n-2}^{(\alpha)}(x) \tag{6}$$

Again I write a top level script for this section, which performs the following tasks:

- `LaguerreGen.m` calculates the generalized Laguerre polynomials recursively using matrix operations. If no alpha is supplied, alpha is set to zero and thus the 'normal' Laguerre polynomial is calculated.

```
function [y, Lcoeff, L] = LaguerreGen(varargin)
if (nargin == 1)                % if only one parameter supplied set alpha = 0 and x
        n = varargin{1};
        alpha = 0;
        x = 0:0.01:150;
elseif (nargin == 2)            % if two parameters then n & alpha provided use given x
        n = varargin{1};
        alpha = varargin{2};
        x = 0:0.01:150;
elseif (nargin == 3)            % if 3 parameters then n, alpha and x provided
        n = varargin{1};
        alpha = varargin{2};
        x = varargin{3};
end;
if (nargin == 0) || (nargin > 3) || (n~=abs(round(n)))
        error('Error');
end;                            % Returns error if conditions not satisfied

Lag=zeros(n+1);                 % Recursive calculation of generalized Laguerre polynomial using eqn. 6
switch n
    case 0
        Lag(1,:)=1;
    otherwise
        Lag(1,:)=[zeros(1,n), 1];
        Lag(2,:)=[zeros(1, n-1), -1, (alpha+1)];  % First two Laguerre
        for i=3:n+1
            A1 = 1/(i-1) * (conv([zeros(1, n-1), -1, (2*(i-1)+alpha-1)], Lag(i-1,:)));
            B1 = 1/(i-1) * (conv([zeros(1, n), ((i-1)+alpha-1)], Lag(i-2,:)));
            A2=A1(length(A1)-n:1:length(A1));
            B2=B1(length(B1)-n:1:length(B1));
            Lag(i,:)=A2-B2;
        end;
end;

y=Lag(n,:);                     % generates a vector of Laguerre polynomial values
Lcoeff = zeros(n,n+1);

for k=1:n
    Lcoeff(k,:) = Lag(k,:);
end                             % generates a matrix of Laguerre polynomial values

L = zeros(n, length(x));        % Creates matrix of values for the x values provided
for i = 1:n
        L(i,:) = polyval(Lcoeff(i,:),x);
end
end
```

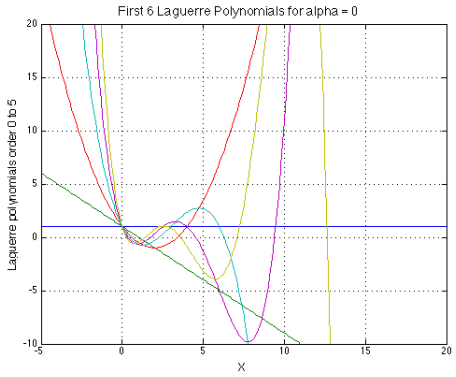

Figure 4: Laguerre Polynomials

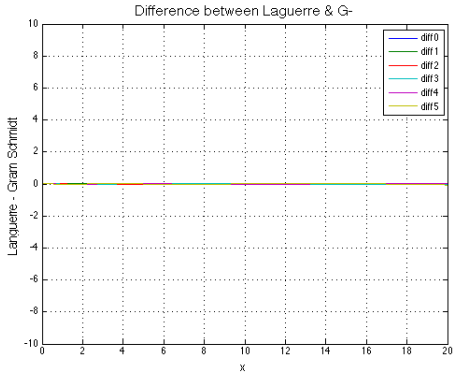

Figure 5: Difference between $\tilde{G_n}$ and $\tilde{L_n}$

- By plotting the difference between $\tilde{G_n}$ and $\tilde{L_n}$ one can see that the odd numbered n terms differ by a nth order polynomial. This is caused by the differentiation of the e^{-x} term switching the signs with each next order. `generate_v`

is adapted to combat this:

```
{V(i,j) = (-x(j))^(i-1);} % Now the difference is zero (Fig.5.)
```

- The gamma function (programmed into MATLAB) can be utilised to form an orthonormal set of assosiated Laguerre functions and `verify_orthog` used again to verify orthogonality.

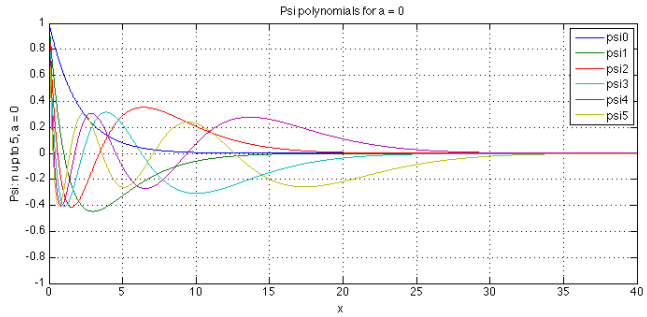

Figure 6: Psi polynomials (alpha = 0)

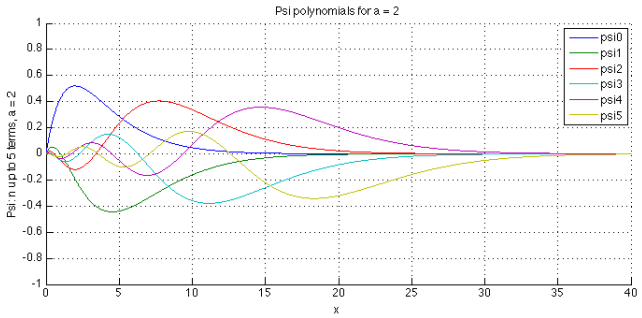

Figure 7: Psi polynomials (alpha = 2)

5 Fitting to Synthesised Data

5.1 Least squares approximation

The exact fitting of high order polynomials can be a very bad idea as small changes to the function values, particularly near the middle of the interpolating region, cause large changes near to the edge of the interpolating region. If you allow an error a polynomial with lower order can be found that almost goes through the points. The least squares approach is just the sum of the absolute errors for the set of points, and for a very large number of points becomes:

$$C = \int_0^\infty (f(x) - f_o(x))^2 dx \tag{7}$$

To solve for the parameters just solve $\frac{\partial C}{\partial a_0} = 0, \frac{\partial C}{\partial a_1} = 0, \frac{\partial C}{\partial a_2} = 0$ and so on. Each equation is independent and gives the result:

$$a_n = \int_0^\infty \psi_n(x) f_o(x) dx \tag{8}$$

This is the same result as given by $\langle \psi_n,\ f_o \rangle\ = \int_0^\infty \psi_n(x)\ f_o(x)\ dx$ so we see that the approximation is a least-squares approach.

5.2 Generating and Fitting to Data

`exp_data.m` generates data by squaring the magnitude of a complex normal distribution with mean, `mu`, and variance, `sigma2`.

For the tasks in this section I created a function that creates a Laguerre fit to a supplied set of input data, called `fitting.m`.

```
function [ffitted] = fitting(n,alpha,fo,x)

[~, ~, L] = fit_LaguerreGen(n,alpha, x);
psi = fit_generate_psi(n, alpha, L, x);             % create the psi vectors

a_n = zeros(1,n);                                   % coefficients for the fitting function
for i = 1:n;
    func = psi(i,:).*fo;
    a_n(1,i) = trapz (x,func);                      % eqn. 8
```

```
end
ffitted = a_n(1,1).*psi(1,:);
 for i = 2:n;
     ffitted = ffitted + a_n(1,i)*psi(i,:);          % eqn. 1
 end
end
```

A chi-square distribution with n degrees of freedom has a mean, n, and a variance of 2n. The distribution function can be calculated from:

$$f(x) = \frac{\frac{e^{-x/2}}{2}(\frac{x}{2})^{\frac{n}{2}-1}}{\Gamma(\frac{n}{2})} \tag{9}$$

As $X = |W|^2 = Y^2 + Z^2$, $n = 2$, and so for `mu = 0; sigma2 = 2;` we would expect $f(x) = \frac{e^{-x/2}}{2}$, which is what I get when I run the script `fitting5_3.m` which plots the data, the laguerre fit (with $\alpha = 0$, order $= 0$) and the parameterised fit.

```
% Parameterised fit
emu = trapz(x,x.*fo);           % estimated sample mean
p_fit = 1/emu.*exp(-x/emu);     % paramaterised fit
```

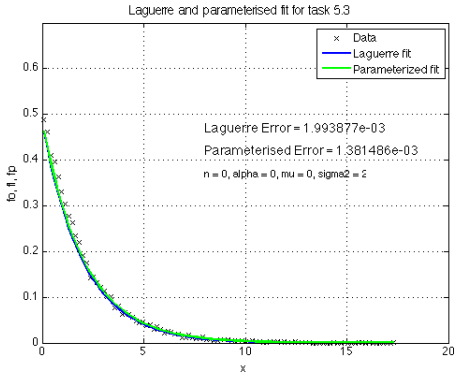

Figure 8: Laguerre and Paramaterised fit for Task 5.3

- The coefficient a_0 is 0.4532 (average of 5 values). This is close to what I expected, 0.5 (expected as it should be $a_0 = \int_0^\infty 0.5e^{-x}\, dx = 0.5$).
- `estimated_mean = trapz(x,x.*fo)` estimates the mean is defined as $E[X] = \int_{-\infty}^{\infty} x\ f(x)dx$ and `fo` has been normalised and so is a probability density function.
- The parameterised fit error is smaller than the Laguerre fit error (values shown on graph).

I then ran `exp_data.m` with the values `sigma2 = 1/3; mu = 1 +sqrt(-1).` The script `fitting5_4` generates a Laguerre fit for expansions of the order of a given n (I used n = 5). Function `minimise_error_given_n.m` finds the value of alpha which provides the smallest error. The script then plots the graphs for this value of alpha, along with the value above and below, as shown in Fig. 9.

```
nth_order = 5;
n = nth_order + 1;
alpha_new = minimise_error_given_n(sigma2, mu, nsamp, nbins, n); % find best alpha for the given n
alpha1 = alpha_new -1;;
alpha2 = alpha_new;
alpha3 = alpha_new + 1;    % Plot for the optimum alpha and the value each side for comparison

[fo, x] = exp_data(sigma2,mu,nsamp,nbins);

[f1] = fitting(n, alpha1, fo, x);
func1 = (f1-fo).^2;
est_error_f1 = trapz(x,func1);
```

```
function alpha_new = minimise_error_given_n(sigma2, mu, nsamp, nbins, n)

no_of_alpha = 10;                          % Number of alphas to test (not the actual alpha value)
[fo, x] = exp_data(sigma2,mu,nsamp,nbins);  % create the random data and corresponding x

E_a = zeros(1,no_of_alpha);
    for j = 0:no_of_alpha-1;
        f = fitting1(n,j,fo,x);
        func = (f-fo).^2;
        est_error_f = trapz(x,func);
        E_a(1,j+1) = est_error_f;
    end

[r]=find(E_a==min(E_a));
alpha_new = r - 1                          % Return alpha that gives smallest error
end
```

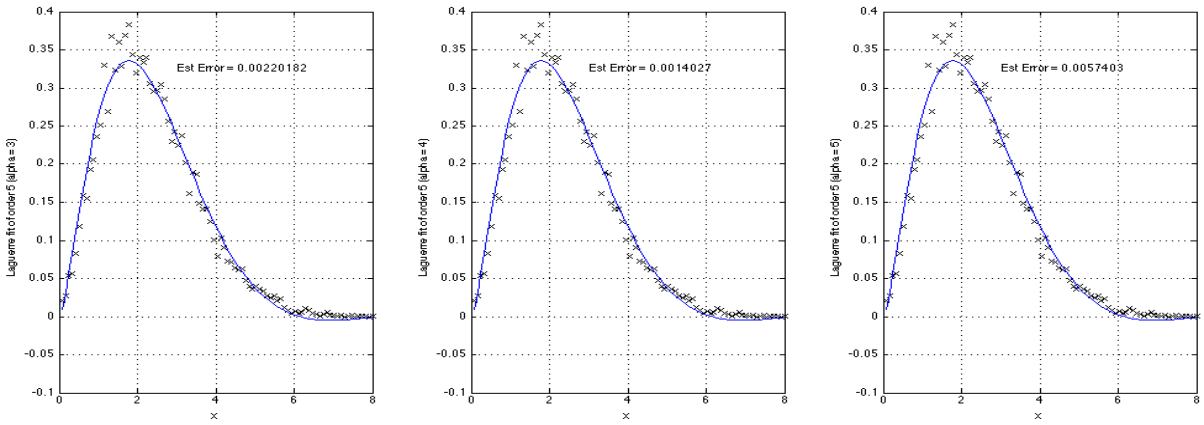

Figure 9: Fitting for Task 5.4 with n = 5 and $\alpha = 3, 4, 5$

Now `exp_data.m` is ran with `sigma2 = 6; mu = 0;`. This leads to the need to redefine the domain of the data by a scaling factor. This need is not to reduce the error as I have previously defined it, but to scale the fit for x terms near to zero. Without it the fit near the origin is poor. I rescale the location data as follows:

```
xscaled = 2*x/sigma2
```

(Notice the factor of two is included to cancel the 2 in the $e^{-x/2}$ term of the equation for $\psi_n^{(\alpha)}(x)$.)

Equation for new 0th order fit with $\alpha = 0$:

$$f(x) = a_0\psi_0 = 0.2206e^{-x/\sigma^2} = 0.2206e^{\frac{-x}{6}}$$

Whereas the equation 5.13 gives the parameterised fit equation as:

$$f(x) = \frac{1}{\xi}e^{-x/\xi} = 0.1598e^{\frac{-x}{6.2563}}$$

NB. The above used `estimated_sample_mean = 6.2563; a_0 = 0.2206` calculated by my code in MATLAB.

Running the script `fitting5_5` returns:

6 Fitting to Real Data Set

Having built up my understanding of function fitting and approximation, as well as how to operate MATLAB, I was ready to apply my new skills to attempt to fit a mathematical model to the real data set provided by the company. Once again I wrote a top level script, `fitting_script.m`, to implement a number of functions which produce the model.

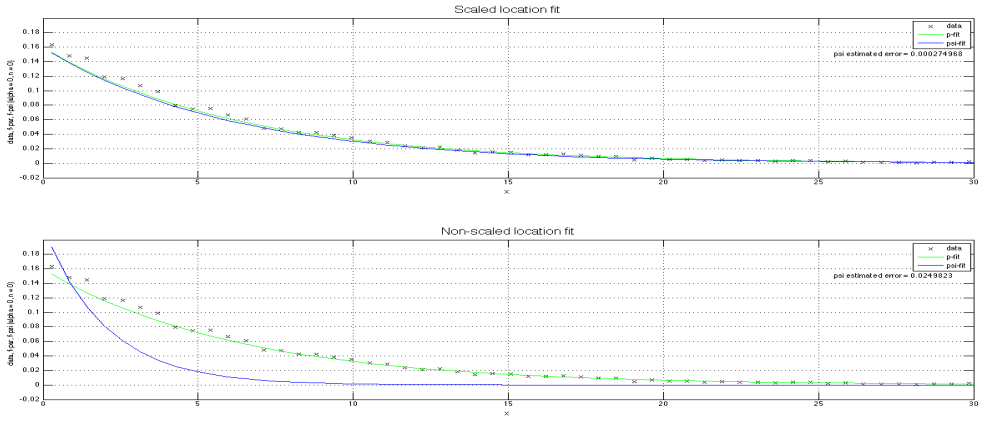

Figure 10: Fit with and without redefined domain

```
load('measured_data.mat')

tic;                                                    % Start timing
no_of_n = 10;                                           % Check orders up to n
no_of_alpha = 10;                                       % check alpha up to a

[n, alpha] = error_6(no_of_n, no_of_alpha);             % Find n and alpha that give smallest error
[est_error_laguerre, f] = fit_measured_data(n, alpha);

t = toc;                                                 % Record elapsed time
plot_data(est_error_laguerre, t, f)
```

- `error_6.m` finds the values of n and α which give the fit with the smallest error. Similar to `minimise_error_given_n.m` used above, but this function steps through a range of both n and α to find the smallest error (Er).

```
function [nth_order, alpha] = error_6(n, alpha)

load('measured_data.mat')
Er = zeros(n+1,alpha+1);
Er_n = zeros(1,alpha+1);

for i = 0:n;
    for j = 0:alpha;
    [E,~] = fit_measured_data(i,j);
    Er_n(1,j+1) = E;
    end
    Er(i+1,:) = Er_n;
end

[n_error,alpha_error]=find(Er==min(min(Er)));
nth_order = n_error - 1
alpha = alpha_error - 1
end
```

- `fit_measured_data.m` generates a Laguerre fit to a set of measured channel data. The fit mathematically describes the probability density of the attenuation (squared) of a measured wireless channel. Both an offset and scale paramters are required. `xi-min(xi)` serves as the offset as it redefines the domain shifts all the x values to the right by the smallest x value. The scaling factor used was sigma, calculated by finding the variance $\sigma^2 = \int x_i f_0 dx - \mu^2$. `fitting.m` is used as previously. (Note that finding the optimised scaling parameter produced a smaller error and a different alpha value)

```
function [est_error_laguerre, f] = fit_measured_data(nth_order, alpha)


load('measured_data.mat')                    % Load measured data.
```

```
n = nth_order + 1;
mu = trapz(xi,xi.*fo);
var = trapz(xi,(xi.^2).*fo) - mu.^2;
sigma = sqrt(var);                          % Scaling parameter to redefine domain

xscaled = (xi-min(xi))/sigma;               % Offset the domain
f = fitting(n,alpha,fo,xscaled);

err = (f-fo).^2;
est_error_laguerre = trapz(xi,err);         % Estimating error
end
```

- Finally, both the data and the fit are plotted by `plot_data.m`. n = 10 and $\alpha = 2$ are used to produce an error of 0.0011977.

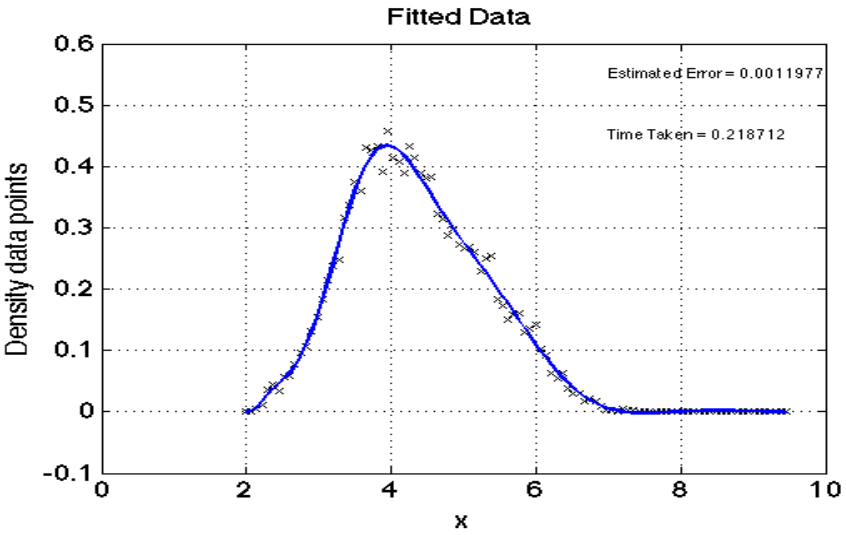

Figure 11: Fitted Data

6.1 Conclusions

The scattering of the radio waves with regards to Wi-Fi is very dependent on the local environment, i.e. what objects are in the vacinity. However, it can be seen from the measured data that for this experiment the signal power has a relatively small standard deviation and the strength drops away quickly around the mean value.

6.2 Possible Further Work

This project could be extended to both explore various other ways of fitting the curve and to improve the fit I came up with. Other orthogonal bases - such as Chebyshev or Hermite polynomials - could be explored as an alternative way to fit the data. We could also explore the use of non-parametric modelling such as kernel regression.

Further functions could be used to ensure a better fit, such as a function that ensures the data is not overfitted. In addition we could explore the possibility of using another error metric and also look at the errors arising from the MATLAB such as that arising from the use of the the trapezoidal rule:

$$E = \frac{a-b}{12} h^2 \frac{d^2 f}{dx^2}(\xi) \tag{10}$$

Other possibilities include exploring methods such as 'economising' - cancelling the highest order term of a polynomial fit - to save on computational time.