Final Project - MTH 361

James Bonnell, Ellen Hamel, TejSai Kakumanu, and Adam Naze

December 12, 2024

1 Introduction and Purpose

There are many currencies that exist in the world, each with their own values. This means that if you want to exchange one currency it will not necessarily be one-to-one. To keep track of the value of a currency, they are compared to other currencies that exist which are from exchange rates. Exchange rates can change due to many factors in economics such as major events like wars, government needs, or trends in society. To keep the global economy flowing, a wealth of data is kept on these exchange rates. Through this record, we as well as any bank or credit union who needs it can check what one currency is worth compared to another. For example, if you walk into a bank in England to exchange United States dollars for English pounds, they will know exactly how many pounds to exchange with you.

For the purposes of this project, we are interested in finding those spots in the data when the conversion was one-to-one. For this we used interpolation since it is a good way to be able to take data points and make a polynomial function with them. This can be applied to things in the real world. In our case, exchange rates of three currencies which are the Pound, Euro, and Swiss Franc were used. To make our functions, the methods we will be using are Lagrange interpolation, Newton divided differences, and linear algebra techniques for finding a polynomial. With these polynomials we can use them to find a point when they were equal. By using zero finding methods including false position, bisection, and secant method, we are able to find when two of the currencies are equal to one another meaning their exchange rate is one-to-one. All data will be taken from Macrotrends which are in Figures 7-9 in the Appendix.

2 Polynomial By Lagrange Interpolation - Pound Dollar Exchange Rate

For the pound dollar exchange rate, the values from the year 2000 to 2016 were taken at four year intervals for a total of five points of data. If the x-axis represents the year (0 representing 2000 to 16 representing 2016) and the y-axis represents the average closing price, the following points can be obtained: (0, 1.52), (4, 1.83), (8, 1.85), (12, 1.59), (16, 1.35). For any given set of nodes with distinct x values (meaning $x_j \neq x_i$ or $j \neq i$) that correlate to y values $(x_i, f(x_i))$, the n^{th} order polynomial can be approximated as:

$$\sum_{j=0}^{n} \ell_j(x) f(x_j) = P_n(x).$$

Here, the Lagrange interpolants (where $k \neq j$) are defined as:

$$\prod_{k=0}^{n} \frac{x - x_k}{x_j - x_k} = \ell_j(x).$$

This is for $k \neq j$, $\ell_j(x_k) = 0$ and $\ell_j(x_j) = 1$. From the set of points listed, there are no two x-values that are the same so the polynomial can be generated by Lagrange interpolation. It should be noted that if values of f(x) are zero then the interpolant associated does not need to be found.

First, the interpolants will be found. Here, $\ell_0(x) = \frac{(x-x_1)(x-x_2)(x-x_3)(x-x_4)}{(x_0-x_1)(x_0-x_2)(x_0-x_3)(x_0-x_4)} = \frac{(x-4)(x-8)(x-12)(x-16)}{(0-4)(0-8)(0-12)(0-16)} = \frac{(x^2-12x+32)(x^2-28x+192)}{6144} =$

$\frac{x^4-40x^3+560x^2-3200x+6144}{6144}$. For the other interpolants, the same process is followed with the following formulas: $\ell_1(x) = \frac{(x-x_0)(x-x_2)(x-x_3)(x-x_4)}{(x_1-x_0)(x_1-x_2)(x_1-x_3)(x_1-x_4)}$, $\ell_2(x) = \frac{(x-x_0)(x-x_1)(x-x_3)(x-x_4)}{(x_2-x_0)(x_2-x_1)(x_2-x_3)(x_2-x_4)}$, $\ell_3(x) = \frac{(x-x_0)(x-x_1)(x-x_2)(x-x_4)}{(x_3-x_0)(x_3-x_1)(x_3-x_2)(x_3-x_4)}$, and $\ell_4(x) = \frac{(x-x_0)(x-x_1)(x-x_2)(x-x_3)}{(x_4-x_0)(x_4-x_1)(x_4-x_2)(x_4-x_3)}$.

Setting up the polynomial, $P_n(x) = (\ell_0)(f(x_0))+(\ell_1)(f(x_1))+(\ell_2)(f(x_2))+(\ell_3)(f(x_3))+(\ell_4)(f(x_4))$. The values for f(x) are the following: 1.52, 1.83, 1.85, 1.59, and 1.35. Following with this formula, the final polynomial can be represented as $P_n(x) = \frac{29x^4}{614400} - \frac{17x^3}{15360} - \frac{41x^2}{38400} + \frac{463x}{4800} + \frac{38}{25}$.

For the calculations for the polynomial, they were found using MATLAB as shown below:

```
clear all
syms x

% Define the Lagrange polynomials (same as before)
L0 = expand((x-4)*(x-8)*(x-12)*(x-16))/((0-4)*(0-8)*(0-12)*(0-16));
L1 = expand((x-0)*(x-8)*(x-12)*(x-16))/((4-0)*(4-8)*(4-12)*(4-16));
L2 = expand((x-0)*(x-4)*(x-12)*(x-16))/((8-0)*(8-4)*(8-12)*(8-16));
L3 = expand((x-0)*(x-4)*(x-8)*(x-16))/((12-0)*(12-4)*(12-8)*(12-16));
L4 = expand((x-0)*(x-4)*(x-8)*(x-12))/((16-0)*(16-4)*(16-8)*(16-12));

% Define the y values
y0 = 1.52;
y1 = 1.83;
y2 = 1.85;
y3 = 1.59;
y4 = 1.35;

% Construct the polynomial P
P = L0*y0 + L1*y1 + L2*y2 + L3*y3 + L4*y4
```

When graphing the data and plotting against the cubic spline, the following MATLAB code can be used:

```
% Lagrange Interpolation
clear all;
close all;

% Define x and fx (iteration nodes and their function values)
x = [0 4 8 12 16];
fx = [1.52 1.83 1.85 1.59 1.35];
n = length(x);  % Number of points
xx = linspace(0, 16, 100);  % Range of points for interpolation

% Initialize the Lagrange basis polynomials
l = zeros(n, length(xx));  % Allocate space for basis polynomials

for j = 1:n
    l(j,:) = ones(1, length(xx));  % Initialize the j-th Lagrange
       polynomial as ones
    for i = 1:n
        if i ~= j
            l(j,:) = l(j,:) .* (xx - x(i)) / (x(j) - x(i));
        end
    end
end
```

```
% Interpolation polynomial P
P = zeros(1, length(xx));  % Initialize the polynomial
for j = 1:n
    P = P + fx(j) * l(j,:);  % Add weighted basis polynomials
end

% Plot the results
xlim([0 16]);
ylim([0 2]);
plot(xx, P, 'b', 'DisplayName', 'Lagrange Interpolation - 5 Points');
hold on;
Q = spline(x, fx, xx);  % Cubic spline interpolation
plot(xx, Q, 'g', 'DisplayName', 'Spline Interpolation - 5 Points');
plot(x, fx, 'rp', 'MarkerFaceColor', 'r', 'DisplayName', 'Data Points'
    );
legend('show');
```

With this code, it plots both the polynomial from the Lagrange interpolation and the cubic spline function embedded within MATLAB in Figure 1 of the appendix. When comparing these two functions, it can be seen that the Lagrange interpolation closely follows the cubic spline. It can also be seen both lines pass through the given points, so they are both accurate to some degree. As the cubic spline is a good interpolation technique, it makes the Lagrange polynomial very promising to use since the two match very well and it will be used for further analysis.

3 Polynomial By Newton Divided Differences - Euro Dollar Exchange Rate

For the euro dollar exchange rate, the points of data will be the following: (0, 0.92), (4, 1.24), (8, 1.47), (12, 1.29), and (16, 1.11). For any given set of nodes with distinct x values (meaning $x_j \neq x_i$ or $j \neq i$) that correlate to y values $(x_i, f(x_i))$, the n^{th} order polynomial can be approximated as:

$$P_n(x) = f[x_o]+f[x_0,x_1](x-x_0)+f[x_0,x_1,x_2](x-x_0)(x-x_1)+...+f[x_0,x_1,...,x_k](x-x_0)(x-x_1)...(x-x_{k-1})$$

Here, the first divided difference is when there is $\frac{f[x_{j+1}]-f[x_j]}{x_{j+1}-x_j} = f[x_j, x_{j+1}]$. Future divided differences then span between other divided differences where the range is between the two closest divided differences. For example, the second divided difference is $\frac{f[x_{j+1},x_{j+2}]-f[x_j,x_{j+1}]}{x_{j+2}-x_j} = f[x_j, x_{j+1}, x_{j+2}]$. After taking the sum of the indicated divided differences with the corresponding products (x-xi), the polynomial is generated. With this method, if the two closest differences are zero it eliminates the need to continue the path of doing divided differences that includes these.

For the given problem, $P_n(x) = f[x_o]+f[x_0,x_1](x-x_0)+f[x_0,x_1,x_2](x-x_0)(x-x_1)+f[x_0,x_1,x_2,x_3](x-x_0)(x-x_1)(x-x_2)+f[x_0,x_1,x_2,x_3,x_4](x-x_0)(x-x_1)(x-x_2)(x-x_3)$.

$f[x_0] = f(x_0) = 0.92$. $f[x_0,x_1] = \frac{f[x_1]-f[x_0]}{x_1-x_0} = \frac{1.24-0.92}{4-0} = 0.08$. $f[x_1,x_2] = \frac{f[x_2]-f[x_1]}{x_2-x_1} = \frac{1.47-1.24}{8-4} = 0.0575$. $f[x_2,x_3] = \frac{f[x_3]-f[x_2]}{x_3-x_2} = \frac{12-8}{1.29-1.47} = -0.045$. $f[x_3,x_4] = \frac{f[x_4]-f[x_3]}{x_4-x_3} = \frac{1.11-1.29}{16-12} = -0.045$. $f[x_0,x_1,x_2] = \frac{f[x_1,x_2]-f[x_0,x_1]}{x_2-x_0} = \frac{0.0575-0.08}{8-0} = -0.0028125$. $f[x_1,x_2,x_3] = \frac{f[x_2,x_3]-f[x_1,x_2]}{x_3-x_1} = \frac{-0.045-0.0575}{12-4} = -0.0128125$. $f[x_2,x_3,x_4] = \frac{f[x_3,x_4]-f[x_2,x_3]}{x_4-x_2} = \frac{-0.045+0.045}{16-8} = 0$. $f[x_0,x_1,x_2,x_3] = \frac{f[x_1,x_2,x_3]-f[x_0,x_1,x_2]}{x_3-x_0} = \frac{-0.0128125+0.0028125}{12-0} = -8.33333{*}10^{-4}$. $f[x_1,x_2,x_3,x_4] = \frac{f[x_2,x_3,x_4]-f[x_1,x_2,x_3]}{x_4-x_1} = \frac{0+0.0128125}{16-4} = 0.0010677083$. $f[x_0,x_1,x_2,x_3,x_4] = \frac{f[x_1,x_2,x_3,x_4]-f[x_0-x_1-x_2-x_3]}{x_4-x_0} = \frac{0.0010677083+8.33333{*}10^{-4}}{16-0} = 1.188151021 * 10^{-4}$.

Plugging these into the polynomial equation, $P_n(x) = 0.92 + (x-0)(0.08) - 0.0028125(x-0)(x-4) - 8.33333 * 10^{-4}(x-0)(x-4)(x-8) + 1.188151021 * 10^{-4}(x-0)(x-4)(x-8)(x-12)$. When simplifying this polynomial, $P_n(x) = \frac{73x^4}{614400} - \frac{283x^3}{76800} + \frac{1079x^2}{38400} + \frac{91x}{4800} + \frac{23}{25}$. The following MATLAB code can be used to generate this polynomial:

```
%Newton Divided Differences - 5 Points (Euro-Dollar)
clear all
```

```
syms x

%Define y-values
y0=0.92;
y1=1.24;
y2=1.47;
y3=1.29;
y4=1.11;

%Define x-values
x0=0;
x1=4;
x2=8;
x3=12;
x4=16;

%Carry out Newton Divided Differences
y01=((y1-y0)/(x1-x0));
y12=((y2-y1)/(x2-x1));
y23=((y3-y2)/(x3-x2));
y34=((y4-y3)/(x4-x3));
y012=((y12-y01)/(x2-x0));
y123=((y23-y12)/(x3-x1));
y234=((y34-y23)/(x4-x2));
y0123=((y123-y012)/(x3-x0));
y1234=((y234-y123)/(x4-x1));
y01234=((y1234-y0123)/(x4-x0));

%Construct the polynomial
P = y0 + y01*(x-x0) + y012*(x-x0)*(x-x1) + y0123*(x-x0)*(x-x1)*(x-x2)
   + y01234*(x-x0)*(x-x1)*(x-x2)*(x-x3)
```

The following code can also be used to generate the polynomial, graph the polynomial, and plot it against cubic splines to check for fit:

```
%Newton Divided Differences - 5 Points (Euro-Dollar)
clear all
close all

x = [0 4 8 12 16];
fx = [0.92 1.24 1.47 1.29 1.11];
n = length(x);  % Number of points
xx = linspace(0, 16, 100);  % Range of points for interpolation

% Zero-th divided differences (initialization)
dd = zeros(n, n);  % Create an n x n matrix for divided differences

for j = 1:n
    dd(1, j) = fx(j);  % First row (0th divided difference) is just fx
        values
end

% All divided differences
for i = 2:n  % Loop over levels of divided differences
    for j = 1:n-i+1  % Loop for the jth value of the ith divided
       difference
        dd(i, j) = (dd(i-1, j+1) - dd(i-1, j)) / (x(j+i-1) - x(j));
    end
end


% Prepare the product terms for the Newton Polynomial
prod = ones(n, length(xx));  % Matrix to hold the product terms

for i = 2:n
    for j = 1:length(xx)
        prod(i, j) = prod(i-1, j) .* (xx(j) - x(i-1));  % Product of
            terms (x - xi)
    end
end

% Calculate the Newton Interpolation Polynomial
P = dd(1, 1) + 0 * xx;  % Start with the 0th term (f(x0))
for i = 2:n
    P = P + dd(i, 1) .* prod(i, :);  % Add higher-order terms
end

% Plot the result
xlim([0 16]);
ylim([0 2]);
plot(xx, P, 'b', 'DisplayName', 'Newton Divided Differences');
hold on;
Q = spline(x, fx, xx);  % Cubic spline interpolation
plot(xx, Q, 'g', 'DisplayName', 'Spline Interpolation');
plot(x, fx, 'rp', 'MarkerFaceColor', 'r', 'DisplayName', 'Data Points'
   );
legend('show');
```

After running this code, the plot generated shows that the cubic spline follows very closely to the polynomial generated by the Newton Divided Differences in Figure 2 of the appendix. As the two match very well, it can be said that the polynomial is sufficient to use that was generated. Something to note is that if this was run through the Lagrange interpolation code from the pound-dollar exchange rate the resulting polynomial is the same. This indicates that either method should be sufficient to use and it doesn't matter which method should be used.

4 Polynomial By Linear Algebra - Swiss Franc Dollar Exchange Rate

The final method that will be used to generate a polynomial will be through the use of linear algebra techniques. The points provided are (0,1.69), (4,1.24), (8,1.08), (12,0.94), and (16,0.99). To turn these into a polynomial by linear algebra, each point will be placed into an equation so for five points there will be five equations. Each equation will have a highest order of one less than the number of points and number of coefficients equal to the number of points. This will generate a polynomial in the end where:

$$P_n(x) = a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0$$

.

In this case,
$y_0 = a_4{x_0}^4 + a_3{x_0}^3 + a_2{x_0}^2 + a_1x_0 + a_0 = a_4 1.69^4 + a_3 1.69^3 + a_2 1.69^2 + a_1 1.69 + a_0 = 0.$
$y_1 = a_4{x_1}^4 + a_3{x_1}^3 + a_2{x_1}^2 + a_1x_1 + a_0 = a_4 1.24^4 + a_3 1.24^3 + a_2 1.24^2 + a_1 1.24 + a_0 = 4.$
$y_2 = a_4{x_2}^4 + a_3{x_2}^3 + a_2{x_2}^2 + a_1x_2 + a_0 = a_4 1.08^4 + a_3 1.08^3 + a_2 1.08^2 + a_1 1.08 + a_0 = 8.$
$y_3 = a_4{x_3}^4 + a_3{x_3}^3 + a_2{x_3}^2 + a_1x_3 + a_0 = a_4 0.94^4 + a_3 0.94^3 + a_2 0.94^2 + a_1 0.94 + a_0 = 12.$
$y_4 = a_4{x_4}^4 + a_3{x_4}^3 + a_2{x_4}^2 + a_1x_4 + a_0 = a_4 0.99^4 + a_3 0.99^3 + a_2 0.99^2 + a_1 0.99 + a_0 = 16.$

To solve these equations, the process is extremely long so this will instead be set as a matrix to help with calculations. To set this as a matrix, the following can be used:

$$A = \begin{bmatrix} {x_0}^4 & {x_0}^3 & {x_0}^2 & x_0 & 1 \\ {x_1}^4 & {x_1}^3 & {x_1}^2 & x_1 & 1 \\ {x_2}^4 & {x_2}^3 & {x_2}^2 & x_2 & 1 \\ {x_3}^4 & {x_3}^3 & {x_3}^2 & x_3 & 1 \\ {x_4}^4 & {x_4}^3 & {x_4}^2 & x_4 & 1 \end{bmatrix}, X = \left[a_0, a_1, a_2, a_3, a_4\right]^T, Y = \left[y_0, y_1, y_2, y_3, y_4\right]^T, AX = Y.$$

Following this equation, MATLAB is able to solve for the coefficients of the polynomial:

```
%Linear Algebra - 5 Points (Swiss Franc-Dollar)
clear all
format long

A=[(0)^4 (0)^3 (0)^2 0 1;
   (4)^4 (4)^3 (4)^2 4 1;
   (8)^4 (8)^3 (8)^2 8 1;
   (12)^4 (12)^3 (12)^2 12 1;
   (16)^4 (16)^3 (16)^2 16 1]

Y=[1.69; 1.24; 1.08; 0.94; 0.99]

X=A\Y
```

From the output of this code, the polynomial comes out to be $P_n(x) = \frac{11x^4}{153600} - \frac{31x^3}{12800} + \frac{289x^2}{9600} - \frac{159x}{800} + \frac{169}{100}$. As this polynomial is the same as if using Newton Divided Differences or Lagrange interpolation, it can be plotted from either technique. The following code was used to plot this polynomial using Lagrange interpolation with a cubic spline as well:

```
% Lagrange Interpolation - 5 Points (Pound-Dollar)
clear all;
close all;

% Define x and fx (iteration nodes and their function values)
x = [0 4 8 12 16];
fx = [1.69 1.24 1.08 0.94 0.99];
n = length(x);  % Number of points
xx = linspace(0, 16, 100);  % Range of points for interpolation

% Initialize the Lagrange basis polynomials
l = zeros(n, length(xx));  % Allocate space for basis polynomials

for j = 1:n
    l(j,:) = ones(1, length(xx));  % Initialize the j-th Lagrange
       polynomial as ones
    for i = 1:n
        if i ~= j
            l(j,:) = l(j,:) .* (xx - x(i)) / (x(j) - x(i));
        end
    end
    % Optionally, you can plot each individual Lagrange basis
       polynomial
    % plot(xx, l(j,:))
    % hold on
end

% Interpolation polynomial P
P = zeros(1, length(xx));  % Initialize the polynomial
```

```
for j = 1:n
    P = P + fx(j) * l(j,:);  % Add weighted basis polynomials
end

% Plot the results
xlim([0 16]);
ylim([0 2]);
plot(xx, P, 'b', 'DisplayName', 'Lagrange Interpolation - 5 Points');
    % Blue line for Lagrange
hold on;
Q = spline(x, fx, xx);  % Cubic spline interpolation
plot(xx, Q, 'g', 'DisplayName', 'Spline Interpolation - 5 Points');  %
    Green line for Spline
plot(x, fx, 'rp', 'MarkerFaceColor', 'r', 'DisplayName', 'Data Points'
   );  % Red points for data
legend('show');
```

With the Lagrange plot for this polynomial with the MATLAB spline at the given points in Figure 3 of the appendix, it can be seen that the two are extremely close to one another. This indicates the linear algebra polynomial is sufficiently close to what it should look like so this polynomial will be used.

5 Possible Error In Polynomials

There are some things to note with the polynomials in terms of error. For any polynomial, given the points $x_0, x_1, ..., x_n$ and corresponding function values $f_0, f_1, ...f_n$, the polynomial $P_n(x)$ of degree n can be interpolated. The error that exists for this is the following:

$$f(x) - P_n(x) = \frac{f^{n+1}(\xi(x))}{(n+1)!}(x - x_0)(x - x_1)...(x - x_n)$$

The reason why this is being brought up is because when plotting using more points, the error has been seen to be very high and the polynomials turned out to be very off from the cubic splines. For instance, originally nine points were plotted at equal intervals from 0 to 16 (or 2000 to 2016) but it was decided to change to just five points. The main reason for this is when plotting this data, it was seen to have extremely high error which may do to several factors. First, when more points are added, the space between points is reduced and the error increases from more terms in the product of multiple differences in x. The (n+1)! does however slightly decrease the error with more points as more points provides the actual shape of the function, but not by enough. Another section to look at is the derivative $f^{n+1}(\xi(x))$ which is critical. This is because the original data fluctuates constantly which can create very high derivative values, especially if more points are used which can have even higher derivative values. This can also cause the polynomial to be very off from what is should be. Something to also note is that the ends of the polynomials were typically the most off. This can be explained by the Range phenomenon where these sections are most frequently off and the more points that are added, the higher the error becomes.

The following code uses the perviously used codes for polynomial interpolation but for ten points:

```
% Lagrange Interpolation - 9 points (Pound-Dollar) For Error Analysis
clear all;
close all;

% Define x and fx (iteration nodes and their function values)
x = [0 2 4 6 8 10 12 14 16];
fx = [1.52 1.50 1.83 1.84 1.85 1.55 1.59 1.65 1.35];
n = length(x);  % Number of points
xx = linspace(0, 16, 100);  % Range of points for interpolation
```

```
% Initialize the Lagrange basis polynomials
l = zeros(n, length(xx));  % Allocate space for basis polynomials

for j = 1:n
    l(j,:) = ones(1, length(xx));  % Initialize the j-th Lagrange
       polynomial as ones
    for i = 1:n
        if i ~= j
            l(j,:) = l(j,:) .* (xx - x(i)) / (x(j) - x(i));
        end
    end
    % Optionally, you can plot each individual Lagrange basis
       polynomial
    % plot(xx, l(j,:))
    % hold on
end

% Interpolation polynomial P
P = zeros(1, length(xx));  % Initialize the polynomial
for j = 1:n
    P = P + fx(j) * l(j,:);  % Add weighted basis polynomials
end

% Plot the results
xlim([0 16]);
ylim([0 2]);
plot(xx, P, 'b', 'DisplayName', 'Lagrange Interpolation - 9 Points');
    % Blue line for Lagrange
hold on;
Q = spline(x, fx, xx);  % Cubic spline interpolation
plot(xx, Q, 'g', 'DisplayName', 'Spline Interpolation - 9 Points');  %
    Green line for Spline
plot(x, fx, 'rp', 'MarkerFaceColor', 'r', 'DisplayName', 'Data Points'
   );  % Red points for data
legend('show');

%_________________________________________________________________
%%
%Lagrange Interpolation - 9 points (Pound-Dollar) For Error Analysis
clear all
syms x

% Define the Lagrange polynomials (same as before)
L0 = expand((x-2)*(x-4)*(x-6)*(x-8)*(x-10)*(x-12)*(x-14)*(x-16))
   /((0-2)*(0-4)*(0-6)*(0-8)*(0-10)*(0-12)*(0-14)*(0-16));
L1 = expand((x-0)*(x-4)*(x-6)*(x-8)*(x-10)*(x-12)*(x-14)*(x-16))
   /((2-0)*(2-4)*(2-6)*(2-8)*(2-10)*(2-12)*(2-14)*(2-16));
L2 = expand((x-0)*(x-2)*(x-6)*(x-8)*(x-10)*(x-12)*(x-14)*(x-16))
   /((4-0)*(4-2)*(4-6)*(4-8)*(4-10)*(4-12)*(4-14)*(4-16));
L3 = expand((x-0)*(x-2)*(x-4)*(x-8)*(x-10)*(x-12)*(x-14)*(x-16))
   /((6-0)*(6-2)*(6-4)*(6-8)*(6-10)*(6-12)*(6-14)*(6-16));
L4 = expand((x-0)*(x-2)*(x-4)*(x-6)*(x-10)*(x-12)*(x-14)*(x-16))
   /((8-0)*(8-2)*(8-4)*(8-6)*(8-10)*(8-12)*(8-14)*(8-16));
L5 = expand((x-0)*(x-2)*(x-4)*(x-6)*(x-8)*(x-12)*(x-14)*(x-16))
   /((10-0)*(10-2)*(10-4)*(10-6)*(10-8)*(10-12)*(10-14)*(10-16));
L6 = expand((x-0)*(x-2)*(x-4)*(x-6)*(x-8)*(x-10)*(x-14)*(x-16))
```

```
   /((12-0)*(12-2)*(12-4)*(12-6)*(12-8)*(12-10)*(12-14)*(12-16));
L7 = expand((x-0)*(x-2)*(x-4)*(x-6)*(x-8)*(x-10)*(x-12)*(x-16))
   /((14-0)*(14-2)*(14-4)*(14-6)*(14-8)*(14-10)*(14-12)*(14-16));
L8 = expand((x-0)*(x-2)*(x-4)*(x-6)*(x-8)*(x-10)*(x-12)*(x-14))
   /((16-0)*(16-2)*(16-4)*(16-6)*(16-8)*(16-10)*(16-12)*(16-14));

% Define the y values
y0 = 1.52;
y1 = 1.50;
y2 = 1.83;
y3 = 1.84;
y4 = 1.85;
y5 = 1.55;
y6 = 1.59;
y7 = 1.65;
y8 = 1.35;

% Construct the polynomial P
P = L0*y0 + L1*y1 + L2*y2 + L3*y3 + L4*y4 + L5*y5 + L6*y6 + L7*y7 + L8
   *y8

%_________________________________________________________________
%%
%Newton Divided Differences - 9 Points (Euro-Dollar)
clear all
close all

x = [0 2 4 6 8 10 12 14 16];
fx = [0.92 0.95 1.24 1.26 1.47 1.33 1.29 1.33 1.11];
n = length(x);  % Number of points
xx = linspace(0, 16, 100);  % Range of points for interpolation

% Zero-th divided differences (initialization)
dd = zeros(n, n);  % Create an n x n matrix for divided differences

for j = 1:n
    dd(1, j) = fx(j);  % First row (0th divided difference) is just fx
        values
end

% All divided differences
for i = 2:n  % Loop over levels of divided differences
    for j = 1:n-i+1  % Loop for the jth value of the ith divided
       difference
        dd(i, j) = (dd(i-1, j+1) - dd(i-1, j)) / (x(j+i-1) - x(j));
    end
end


% Prepare the product terms for the Newton Polynomial
prod = ones(n, length(xx));  % Matrix to hold the product terms

for i = 2:n
    for j = 1:length(xx)
        prod(i, j) = prod(i-1, j) .* (xx(j) - x(i-1));  % Product of
           terms (x - xi)
    end
end

% Calculate the Newton Interpolation Polynomial
P = dd(1, 1) + 0 * xx;  % Start with the 0th term (f(x0))
for i = 2:n
    P = P + dd(i, 1) .* prod(i, :);  % Add higher-order terms
end

% Plot the result
xlim([0 16]);
ylim([0 2]);
plot(xx, P, 'b', 'DisplayName', 'Newton Divided Differences - 9 Points
   ');
hold on;
Q = spline(x, fx, xx);  % Cubic spline interpolation
plot(xx, Q, 'g', 'DisplayName', 'Spline Interpolation - 9 Points');
plot(x, fx, 'rp', 'MarkerFaceColor', 'r', 'DisplayName', 'Data Points'
   );
legend('show');

%_________________________________________________________________
%%
%Newton Divided Differences - 9 Points (Euro-Dollar)
clear all
syms x

%y-values
y0=0.92;
y1=0.95;
y2=1.24;
y3=1.26;
y4=1.47;
y5=1.33;
y6=1.29;
y7=1.33;
y8=1.11;

%x-values
x0=0;
x1=2;
x2=4;
x3=6;
x4=8;
x5=10;
x6=12;
x7=14;
x8=16;

%Divided Differences
y01=((y1-y0)/(x1-x0));
y12=((y2-y1)/(x2-x1));
y23=((y3-y2)/(x3-x2));
y34=((y4-y3)/(x4-x3));
y45=((y5-y4)/(x5-x4));
y56=((y6-y5)/(x6-x5));
```

```
y67=((y7-y6)/(x7-x6));
y78=((y8-y7)/(x8-x7));
y012=((y12-y01)/(x2-x0));
y123=((y23-y12)/(x3-x1));
y234=((y34-y23)/(x4-x2));
y345=((y45-y34)/(x5-x3));
y456=((y56-y45)/(x6-x4));
y567=((y67-y56)/(x7-x5));
y678=((y78-y67)/(x8-x6));
y0123=((y123-y012)/(x3-x0));
y1234=((y234-y123)/(x4-x1));
y2345=((y345-y234)/(x5-x2));
y3456=((y456-y345)/(x6-x3));
y4567=((y567-y456)/(x7-x4));
y5678=((y678-y567)/(x8-x5));
y01234=((y1234-y0123)/(x4-x0));
y12345=((y2345-y1234)/(x5-x1));
y23456=((y3456-y2345)/(x6-x2));
y34567=((y4567-y3456)/(x7-x3));
y45678=((y5678-y4567)/(x8-x4));
y012345=((y12345-y01234)/(x5-x0));
y123456=((y23456-y12345)/(x6-x1));
y234567=((y34567-y23456)/(x7-x2));
y345678=((y45678-y34567)/(x8-x3));
y0123456=((y123456-y012345)/(x6-x0));
y1234567=((y234567-y123456)/(x7-x1));
y2345678=((y345678-y234567)/(x8-x2));
y01234567=((y1234567-y0123456)/(x7-x0));
y12345678=((y2345678-y1234567)/(x8-x1));
y012345678=((y12345678-y01234567)/(x8-x0));

%Forming the polynomial
P1 = y0 + y01*(x-x0) + y012*(x-x0)*(x-x1) + y0123*(x-x0)*(x-x1)*(x-x2)
     + y01234*(x-x0)*(x-x1)*(x-x2)*(x-x3) + y012345*(x-x0)*(x-x1)*(x-x2
    )*(x-x3)*(x-x4) + y0123456*(x-x0)*(x-x1)*(x-x2)*(x-x3)*(x-x4)*(x-x5
    ) + y01234567*(x-x0)*(x-x1)*(x-x2)*(x-x3)*(x-x4)*(x-x5)*(x-x6) +
    y012345678*(x-x0)*(x-x1)*(x-x2)*(x-x3)*(x-x4)*(x-x5)*(x-x6)*(x-x7)
P=simplify(P1)

%_________________________________________________________________
%%
% Lagrange Interpolation - 9 Points (Pound-Dollar)
clear all;
close all;

% Define x and fx (iteration nodes and their function values)
x = [0 2 4 6 8 10 12 14 16];
fx = [1.69 1.55 1.24 1.25 1.08 1.04 0.94 0.92 0.99];
n = length(x);  % Number of points
xx = linspace(0, 16, 100);  % Range of points for interpolation

% Initialize the Lagrange basis polynomials
l = zeros(n, length(xx));  % Allocate space for basis polynomials

for j = 1:n
    l(j,:) = ones(1, length(xx));  % Initialize the j-th Lagrange
```

```
        polynomial as ones
    for i = 1:n
        if i ~= j
            l(j,:) = l(j,:) .* (xx - x(i)) / (x(j) - x(i));
        end
    end
    % Optionally, you can plot each individual Lagrange basis
        polynomial
    % plot(xx, l(j,:))
    % hold on
end

% Interpolation polynomial P
P = zeros(1, length(xx));  % Initialize the polynomial
for j = 1:n
    P = P + fx(j) * l(j,:);  % Add weighted basis polynomials
end

% Plot the results
xlim([0 16]);
ylim([0 2]);
plot(xx, P, 'b', 'DisplayName', 'Lagrange Interpolation - 9 Points');
    % Blue line for Lagrange
hold on;
Q = spline(x, fx, xx);  % Cubic spline interpolation
plot(xx, Q, 'g', 'DisplayName', 'Spline Interpolation - 9 Points');  %
    Green line for Spline
plot(x, fx, 'rp', 'MarkerFaceColor', 'r', 'DisplayName', 'Data Points'
   );  % Red points for data
legend('show');

%___________________________________________________________________
%%
%Linear Algebra Techniques - 9 Points (Swiss Franc-Dollar)
clear all
format long

%Equations Broken Down to Matrix
A=[(0)^8 (0)^7 (0)^6 (0)^5 (0)^4 (0)^3 (0)^2 0 1;
   (2)^8 (2)^7 (2)^6 (2)^5 (2)^4 (2)^3 (2)^2 2 1;
   (4)^8 (4)^7 (4)^6 (4)^5 (4)^4 (4)^3 (4)^2 4 1;
   (6)^8 (6)^7 (6)^6 (6)^5 (6)^4 (6)^3 (6)^2 6 1;
   (8)^8 (8)^7 (8)^6 (8)^5 (8)^4 (8)^3 (8)^2 8 1;
   (10)^8 (10)^7 (10)^6 (10)^5 (10)^4 (10)^3 (10)^2 10 1;
   (12)^8 (12)^7 (12)^6 (12)^5 (12)^4 (12)^3 (12)^2 12 1;
   (14)^8 (14)^7 (14)^6 (14)^5 (14)^4 (14)^3 (14)^2 14 1;
   (16)^8 (16)^7 (16)^6 (16)^5 (16)^4 (16)^3 (16)^2 16 1];

Y=[1.69; 1.55; 1.24; 1.25; 1.08; 1.04; 0.94; 0.92; 0.99];

%Solving For Coefficients of Equations
X=A\Y
```

The graphs generated for these polynomials compared to the cubic splines are in Figures 4-6 in the Appendix. What can be seen is that the ends of each polynomial is off as the Range phenomenon is taking place. This goes back to the reason why the number of points was reduced to 5 since this error is almost not existent when comparing the polynomials to the cubic splines. Something to also mention is with more points, the coefficients would end up extremely high or low which made them more difficult to use. Also, originally the years did not go from 0-16 but from 2000-2016 which also made these coefficients difficult to work with. The graphs even ended up being impossible to read as it turned into vertical lines and it was hard to compare polynomials.

6 Situations For Lagrange Interpolation and Newton's Divided Differences

Typically, Lagrange interpolation is not as useful as Newton divided differences. However, if the function needs to change it is easier to be done with Lagrange interpolation since x-values aren't being added or removed. This allows the formula to easily be changed by only altering the f(x) values since the Lagrange interpolants stay the same. The data set also is not really a determining factor for which method to use since what mostly matters is what you do for future iterations such as adding points or changing the function.

Newton divided differences should be used when adding an additional point as only one term is added to the polynomial and doesn't require doing a long series of calculations like with Lagrange interpolation. For more points added, the entire polynomial doesn't need to be redone and previous terms aren't impacted. When something looks like it will be a polynomial with a small degree, it is usually easier to calculate with Newton divided differences. This is also true if the points look like a small polynomial and there are many points given since terms can cancel out more easily at higher orders.

Either way, both methods do work to some degree but the time it takes to actually calculate out the polynomial can alter based on what is done to the function. For this project, it was easier to work with Lagrange interpolation as the x-values never changed but the y-values did change in the three comparisons made.

7 Exchange Rate Polynomials For Comparison

The following are the polynomials for each exchange rate:

Pound-Dollar - $P_1(x) = \frac{29x^4}{614400} - \frac{17x^3}{15360} - \frac{41x^2}{38400} + \frac{463x}{4800} + \frac{38}{25}$

Euro-Dollar - $P_2(x) = \frac{73x^4}{614400} - \frac{283x^3}{76800} + \frac{1079x^2}{38400} + \frac{91x}{4800} + \frac{23}{25}$

Swiss Franc-Dollar - $P_3(x) = \frac{11x^4}{153600} - \frac{31x^3}{12800} + \frac{289x^2}{9600} - \frac{159x}{800} + \frac{169}{100}$

To determine if these polynomials are really good enough to use for further analysis, they can be compared to the original data the points were taken from. The original data is presented in Figures 7-9 and these are compared to Figures 1-3 which graphs these derivatives. Looking at the general shape of these polynomials, they are all definitely captured. While more points would fit the shape much better, it would increase error so the polynomials will still be used for the zero finding techniques.

8 Find the Equations of the Compared Exchange Rates

To find the intersection points, we equate the three equations to each other and apply the root finding methods on them.

Firstly, we can equate pound-dollar and euro-dollar to find its intersection point:

$$P_1 = P_2 \Longrightarrow P_1 - P_2 = 0$$

By solving this expression using the Matlab code, we get the following equation for the Pound-Euro exchange rate:

$$-\frac{11 * x^4}{153600} + \frac{33 * x^3}{12800} - \frac{7 * x^2}{240} + \frac{31 * x}{400} + \frac{3}{5} = 0$$

Now, we can do the same for euro-dollar and Swiss Franc-dollar to get the Euro-Swiss Franc exchange rate:

$$P_2 = P_3 \Longrightarrow P_2 - P_3 = 0$$

$$\frac{29 * x^4}{614400} - \frac{97 * x^3}{76800} - \frac{77 * x^2}{38400} + \frac{209 * x}{960} - \frac{77}{100} = 0$$

Lastly, the same is done for the Swiss Franc-Dollar and Pound-Dollar to get the Swiss Franc-Pound exchange rate:

$$P_3 = P_1 \Longrightarrow P_3 - P_1 = 0$$

$$\frac{x^4}{40960} - \frac{101 * x^3}{76800} + \frac{399 * x^2}{12800} - \frac{1417 * x}{4800} + \frac{17}{100} = 0$$

Now that we have all the difference equations, we can apply the zero finding techniques on these. This will include bisection method, secant method, and the method of false position. For the error, it will be set to 10^-6 which is relatively large because we are just trying to roughly estimate what year would be used. A smaller error could be used, however this could take more time to compute and isn't necessary as long as the year and month can be established since the actual data only provides these for dates.

9 Bisection Method

The following is the general algorithm form for the bisection method:

1. $f(a) * f(b) < 0$
2. $x_0 = \frac{b+a}{2}$
3. $|f(x_0)| < Tol_{RES}$
4. If $f(x_0) * f(a) < 0$, $b = x_0$, else $a = x_0$
5. $|x_n - x^*| < \frac{b-a}{2^n} < Tol_{ERR}$

Starting with **Point 1**, the Intermediate Value Theorem (IVT) is being used. This theorem states that for a continuous function, in the domain from a to b, all values for y will be hit at least one time. Since the approach of the Bisection Method is to find where the function approaches zero, then along with the (IVT) function at one end must be negative and the function at the other must be positive. The main idea of point 1 is to say that f(a) and f(b) are of opposite signs so that f(c)=0 by IVT. If this is not the case, the Bisection method doesn't work.

Point 2's purpose is to select a point. This is more of a guess, however it is better than random as it tries to predict that function will approach zero in the middle of the domain from a to b. While this is almost never the case, this will help in future guesses which will be explained under point 4.

Point 3 is what determines when we would like to say we are close enough to reaching zero. With the Bisection Method, in every iteration we are trying to get closer and closer to zero. It is very likely however that we never reach a perfect zero, so to save time a residual tolerance is determined to say when to stop. This depends on the problem or goal needed, but generally 10^{-8} or less is good. It should be noted that this is a conditional thing and is used so that the program doesn't continue to run for a long period of time.

Point 4 is the heart of the Bisection Method. With the original guess from x_0, $f(x_0)$ (if it didn't come out to be 0) will either be positive or negative. If multiplying this value by $f(a)$ provides a negative value then x_0 becomes the new b. If not, then x_0 becomes the new a. This is mainly stating what was done in point 1, but now the domain has changed. Since x_0 is the midpoint between a and b, the domain has actually been halved. After going through point 4, this now updates point 2. At point 2 now, there would be a new x_0, then this would carry into point 3 which would check if the algorithm should stop, and finally go to point 4 where a new domain may be set. This cycle would run continuously until point 3 or point 5 (which is discussed next) is no longer met.

Finally, in **Point 5** is another way to determine when the algorithm is finished. The values of b and a refer to the initial domain before any iterations are performed. Starting with $\frac{b-a}{2^n} < Tol_{ERR}$, this can be rearranged to be $n = log_2(\frac{b-a}{Tol_{ERR}})$. If at any point the number of iterations is too large, then we can stop the algorithm. The error tolerance is determined for the problem, similar to the residual tolerance. In $|x_n - x^*| < \frac{b-a}{2^n}$, the $|x_n - x^*|$ represents the error which cannot be measured. This would require the actual value x^* to be known which typically isn't. What this says is if the error is too small, the algorithm should stop.

For finding out if the bisection method will work, the function must be continuous and the function domain must satisfy $f(a) * f(b) < 0$ which means obtaining two y-values such that one is positive and the other is negative. As all the polynomials satisfy these conditions, this method can be utilized

10 Secant Method

The secant method is evolved from Newton's Iteration. For this method to work, the function must be continuous and differentiable in its domain which all the polynomials are. Starting with the slope formula, it can be approximated that $f'(x_k) \approx \frac{f(x_k)-f(x_{k-1})}{x_k - x_{k-1}}$. Plugging this into Newton's iteration of $x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$ provides $x_{k+1} \approx x_k - \frac{f(x_k)*(x_k - x_{k-1})}{f(x_k)-f(x_{k-1}} = \frac{x_k*(f(x_k)-f(x_{k-1})-f(x_k)*(x_k - x_{k-1})}{f(x_k)-f(x_{k-1}}$. Finally, $x_{k+1} \approx \frac{x_{k-1}*f(x_k)-x_k*f(x_{k-1})}{f(x_k)-f(x_{k-1}}$. The main benefit that the secant method has compared to the bisection method is that this converges much more quickly and doesn't require the ends of the domain to have opposite signs.

11 Method of False Position

The method of false position is a method that combines both the secant method and bisection method. If the function is continuous and differentiable on the interval (a,b), it checks to see if they provide y-values of opposite sign. This is $f(a) * f(b) < 0$. Now, the midpoint is selected which is $x_0 = \frac{a+b}{2}$. This acts as the initial guess. If $f(x_o) * f(a) < 0$, the new bound becomes $[a, x_0]$, otherwise it becomes $[x_0, b]$. Everything up until now has been the bisection method, however now the secant method will be implemented. The secant method gives the formula $x_{k+1} \approx \frac{x_{k-1}*(x_k)^2 - x_k*(x_{k-1})^2}{(x_k)^2-(x_{k-1})^2}$. Using this concept, if the previous bound was $[a, x_0]$ the x_{k-1} is replaced by a and x_k is replaced by x_0. Otherwise, x_{k-1} is replaced by x_0 and x_k is replaced by b. Now, this process repeats back to the start where x_1 now replaces x_0 and the previous x_0 is now the new bound (either a or b). This continues until a value is hopefully converged towards. The reason why this method is adopted compared to the secant method and bisection method is that it takes the speed of the secant method while ensuring that the function stays within smaller and smaller bounds which is due to the bisection method. This is so the function cannot diverge outside the domain.

12 Methods Not Used

The first method not used in the analysis is the fixed point iteration. With this method, it requires that the function is mapped to its domain. While this is true for the Pound-Euro and it would work, this is not true for the other methods. Since this method cannot be verified by all the polynomials, it was decided to not use this method. The other method not used was Newton's method. This is a very good technique, however all the polynomials used do have a derivative of zero within their domain. Since this is the case, the method was not used.

13 Finding the Pound-Euro 1:1 Rate

For finding the year at which the Pound-Euro exchange rate is a 1:1 ratio, the bisection method will be used. The polynomial for this is the following: $-\frac{11*x^4}{153600} + \frac{33*x^3}{12800} - \frac{7*x^2}{240} + \frac{31*x}{400} + \frac{3}{5} = 0$. Looking at the left end point, $f(a) = f(0) = \frac{3}{5}$. This means that the left end is positive, so for the method to work the right end needs to be negative. However, from 0 to 16 this is never true. Only in the future was the right end seen to be negative, so it was decided to go from 0 to 20 in the domain with $f(b) = -0.35$. This means that this polynomial would be predicting the future from the year 2016. Using the polynomial that was previously obtained, the following code can be used to solve for this point when the function hits zero:

```
%Bisection Method - Pound-Euro
clc
clear all
a = 0; %Left end point
b = 20; %Right end point
pound_euro = @(x) - (11*x.^4)/153600 + (33*x.^3)/12800 - (7*x.^2)/240
    + (31*x)/400 + 3/5;
f = pound_euro;
ResTol = 1.e-6; % Residual Tolerance
ErrTol = 1.e-6; % Error Tolerance
Nsteps = ceil(log2((b - a)/ErrTol)); % Time Steps that will GUARENTEE
    we reach error tolerance.
x = (b+a)/2;
if (f(a)*f(b) < 0)  % Check that function has an opposite points at
    the left and right endpoints.
        for i = 1:Nsteps  % Take up to Nsteps
             x = (b+a)/2   % Find the midpoint
             xguess(i) = x; % Array that stores all the midpoints
             if (abs(f(x)) < ResTol) % Check that my residual is not
                 too small
                  break; % Break if it is small enough
             end
             if(f(x) * f(a) < 0) % Check which half has the value that
                 we are looking for
                  b = x;
             else
                  a = x;
             end
        end
end
xstar = x %Final x value
size(xguess) %Number of iterations
```

What is found with this code is that the time at which the exchange rate of the Pound-Euro is 1:1 is the year 2018.419 which is roughly the month of May in 2018.

To make a comparison with the other methods, the bisection method has also been tested with the method of false position and the secant method. The secant method can be used as the function is continuous and differentiable in its domain and the method of false position can be used since both the bisection and secant criteria were fulfilled. The following codes were developed for these two other methods:

```
%Secant Method - Pound-Euro
clc
clear all
a = 0; %Left end point
b = 20; %Right end point
pound_euro = @(x) - (11*x.^4)/153600 + (33*x.^3)/12800 - (7*x.^2)/240
    + (31*x)/400 + 3/5;
f = pound_euro; %Function
ResTol = 1.e-6; % Residual Tolerance
Nsteps = 100; %Number of iterations
x_old = a; %Leftbound is the guess before midpoint guess
x = (b+a) / 2; %Midpoint is first guess after x_old
for i=1:Nsteps %Loops for number of iterations
    xtemp = x; %Stores x-value
    x = x - f(x)*(x-x_old)/(f(x)-f(x_old)); % Secant Iteration
    x_old = xtemp; %Replaces initial guess with xtemp
    xguess(i) = x; %Latest guess
    if(abs(f(x)) < ResTol) %If error tolerance is larger, the code
        will stop
         break
    end
end

xstar = x %Final value of x
size(xguess) %Number of iterations

%%
%Method of False Position - Pound-Euro
clc
clear all
a = 0; %Left end point
b = 20; %Right end point
pound_euro = @(x) - (11*x.^4)/153600 + (33*x.^3)/12800 - (7*x.^2)/240
    + (31*x)/400 + 3/5;
f = pound_euro; %Function
ResTol = 1.e-6; % Residual Tolerance
Nsteps = 20; %Number of iterations

% Check that function has an opposite points at the left and right
    endpoints.
 if (f(a)*f(b) < 0)
        x = (b+a)/2; % Find the midpoint
        xguess(1) = x
        % Check which half has the value that we are looking for
        if(f(x) * f(a) < 0)
            xold = a;
        else
            xold = b;
        end
        for i=1:Nsteps
            xnew = (xold * f(x) - x*f(xold))/(f(x)-f(xold));
            xguess(i) = x;
            % Check which side has the value that we are looking for
            if(f(xnew) * f(x) < 0)
                xold = x;
            end
            x = xnew;

            if (abs(f(x)) < ResTol)  % Check that my residual is not
                too small
                 break; % Break if it is small enough
            end

        end
  end
xstar = x %Final value of x
size(xguess) %Number of iterations
```

With the secant method and method of false position, they both arrive at the same answer as the bisection method of year 2018.419 or May 2018. What this says is that all methods do work and that they are all sufficient in being able to find a zero for a function. This justifies these methods being used and they will be used for the other two comparisons.

Something interesting is that the number of iterations in each method is different and the convergence is different. When plotting the xguess from each method, Figure 10 was developed to compare each method. It can be seen that the secant method surprisingly had difficulty converging initially but quickly converged soon after. The number of iterations for bisection method was 18, for secant method was 15, and for method of false position was 12. This goes to show that the method of false position is very helpful in controlling the secant method to converge a little more quickly.

14 Finding the Euro-Swiss 1:1 Rate

For finding the Euro-Swiss exchange rate, the secant method was selected to be used. Since the polynomial is continuous and differentiable on its domain, this method can be used. It should also be noted that the domain went back to going from 0 to 16 as a zero was found in this domain. The following code provides the year for the 1:1 rate between Euros and dollars:

```
%Secant Method - Euro-Swiss
clc
clear all
a = 0; %Left end point
b = 16; %Right end point
pound_euro = @(x) (29*x.^4)/614400 - (97*x.^3)/76800 - (77*x.^2)/38400
    + (209*x)/960 - 77/100;
f = pound_euro; %Function
ResTol = 1.e-6; % Residual Tolerance
Nsteps = 100; %Number of iterations
x_old = a; %Leftbound is the guess before midpoint guess
x = (b+a) / 2; %Midpoint is first guess after x_old
for i=1:Nsteps %Loops for number of iterations
    xtemp = x; %Stores x-value
    x = x - f(x)*(x-x_old)/(f(x)-f(x_old)); % Secant Iteration
    x_old = xtemp; %Replaces initial guess with xtemp
    xguess(i) = x; %Latest guess
    if(abs(f(x)) < ResTol) %If error tolerance is larger, the code
       will stop
        break
    end
end

xstar = x %Final value of x
size(xguess) %Number of iterations
```

From this code, it can be seen that at the year 2004 (in January) the exchange rate between the Euro and Swiss Franc will be 1:1. The number of iterations it took to achieve this was 6 iterations.

15 Finding the Swiss Franc-Dollar 1:1 Rate

The final comparison is with the Swiss Franc to Dollar exchange rate where the method of false position will be utilized. The following code uses this method:

```
%Method of False Position - Swiss Pound
clc
clear all
a = 0; %Left end point
b = 16; %Right end point
pound_euro = @(x) (x.^4)/40960 - (101*x.^3)/76800 + (399*x.^2)/12800 -
    (1417*x)/4800 + 17/100;
f = pound_euro; %Function
ResTol = 1.e-6; % Residual Tolerance
```

```
Nsteps = 20; %Number of iterations

% Check that function has an opposite points at the left and right
   endpoints.
 if (f(a)*f(b) < 0)
        x = (b+a)/2; % Find the midpoint
        xguess(1) = x
        % Check which half has the value that we are looking for
        if(f(x) * f(a) < 0)
            xold = a;
        else
            xold = b;
        end
        for i=1:Nsteps
            xnew = (xold * f(x) - x*f(xold))/(f(x)-f(xold));
            xguess(i) = x;
            % Check which side has the value that we are looking for
            if(f(xnew) * f(x) < 0)
                xold = x;
            end
            x = xnew;

            if (abs(f(x)) < ResTol)  % Check that my residual is not
               too small
                break; % Break if it is small enough
            end
        end
  end
xstar = x %Final value of x
size(xguess) %Number of iterations
```

The output from this code gives the year 2000.614 for the Swiss-Pound exchange rate which converts to roughly July, 2000. This was obtained in 6 iterations.

16 Summary of Zero Findng Results

The following are the results acquired (correct to three decimal places) by performing all the zero finding techniques:

Function	Method Used	Iterations	Year
Pound-Euro	Bisection Method	18	2018.419
Euro-Swiss	Secant Method	6	2004.000
Swiss-Pound	Method of False Position	6	2000.614

To further observe if these results are correct, Figure 11 can be used which displays the polynomials of the exchange rates from 2000 to 2020. As seen, where these polynomials intersect indicates an exchange rate of 1:1. This is also verified in Figure 12 where the polynomials reach 0.

With the number of iterations, it can be seen that the bisection method is by far the slowest method. While secant method and method of false position were both much faster, the method of false position can at least ensure the secant lines are constrained which can make the method faster as seen when doing the pound-euro exchange rate.

17 Comparison to Actual Data

Based on data from Macrotrends, for the Pound-Euro exchange rate at the year 2018 in May the actual exchange rate is 1:0.89. For the Euro-Swiss exchange rate at the year 2004 in January, the actual rate is roughly 1.50:1. Finally, in July 2000 the actual exchange rate for the Swiss Franc to Pound was 1:1.52. When looking at these values compared to the ideal 1:1 ratios, these are all extremely close. Especially the Pound-Euro turned out the best as it was closest to this ratio. While they are all off, they can still be considered good enough for the purposes of this project. With more data points these ratios may become better, however the polynomials can also become more off and polynomial generation methods are needed to allow the polynomial to not have extreme errors such as from the Range phenomenon.

18 Conclusion

In general, this process of zero finding can be used to approximate and estimate the changes in exchange rates. To get better results, a bigger domain and higher number of points would be required. However, more error could occur so methods would be needed to reduce this as much as possible. This process can also be a first step to begin a business where one could buy a currency when the exchange rates are low and sell them when the exchange rates hike up if the methods were used to predict the future. This could result in big profit margins. However, since this polynomial approximation doesn't take natural events like elections, natural disasters, the economy in the country, etc. into consideration, it is very unreliable to depend on these results.

Something to note about future predictions is that the polynomials all ended up approaching infinity when going too far away from the initial domain. That is why these methods are most accurate when being close to the initial domain. The further the year is from the initial domain, the more uncertainty arises and the predictions are much less accurate. Luckily for this project, the data was able to predict a 1:0.89 ratio for the Pound-Euro exchange rate which is very close to the 1:1 and this is very surprising.

Overall, the polynomial techniques can be used to approximate given data to some degree which can be controlled based on the number of points, spacing of points, and the domain selected. By changing these types of parameters, the error can increase or decrease. With a function or polynomial, a zero can be found utilizing multiple techniques. Some techniques are much faster than others, but techniques can also have specific requirements such as not having a zero derivative within the domain or requiring the function to be of opposite signs at the left and right ends of the domain.

19 Appendix

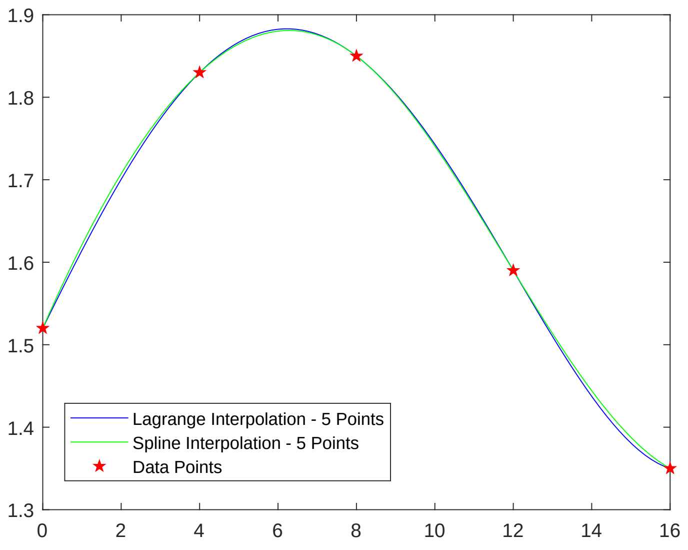

Figure 1: Lagrange Interpolation (Pound-Dollar) With 5 Points

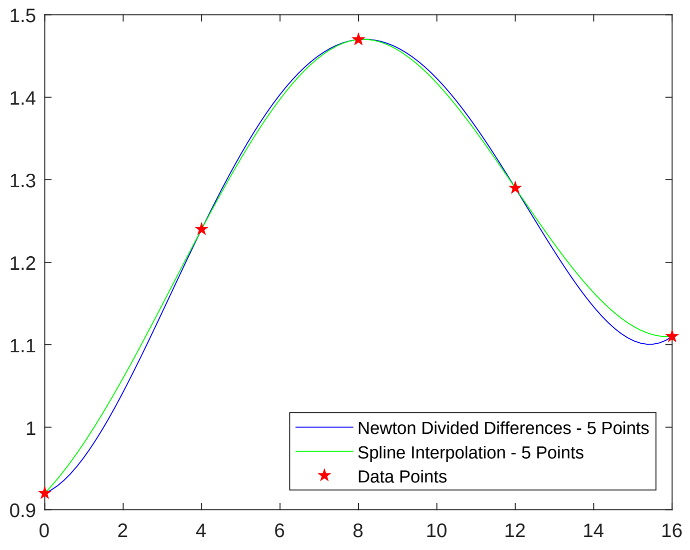

Figure 2: Newton Divided Difference (Euro-Dollar) With 5 Points

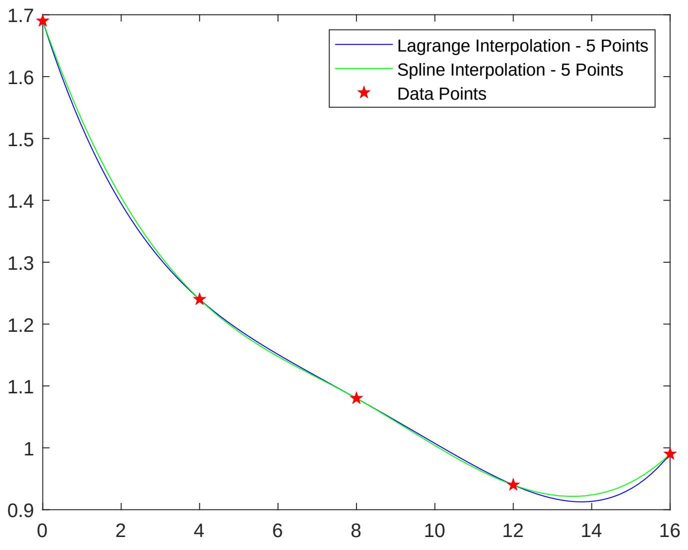

Figure 3: Lagrange Interpolation (Swiss Franc-Dollar) With 5 Points

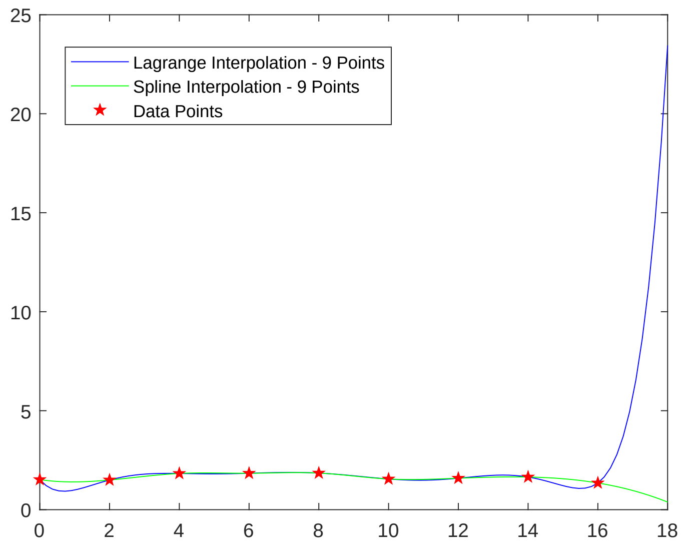

Figure 4: Lagrange Interpolation (Pound-Dollar) With 9 Points

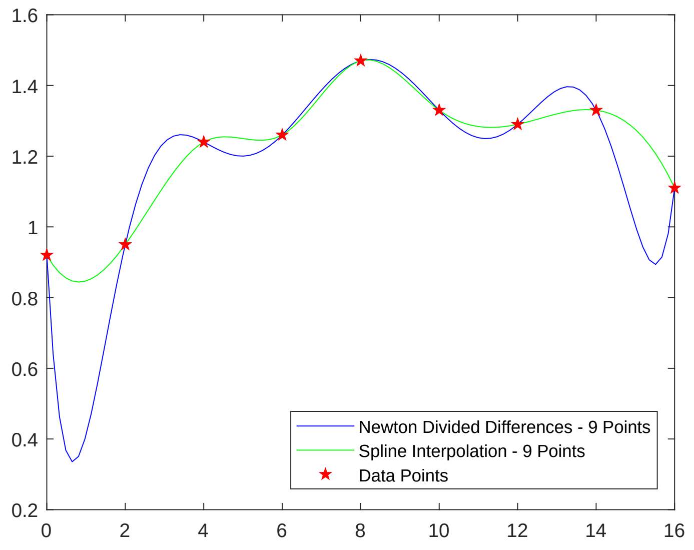

Figure 5: Newton Divided Difference (Euro-Dollar) With 9 Points

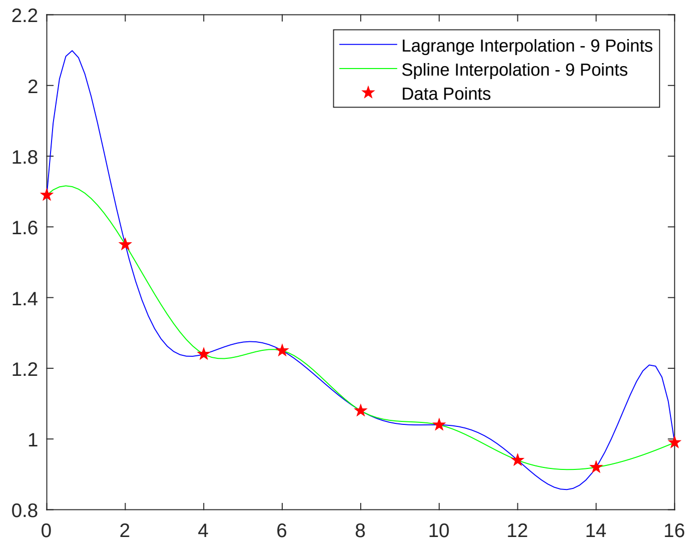

Figure 6: Lagrange Interpolation (Swiss Franc-Dollar) With 5 Points

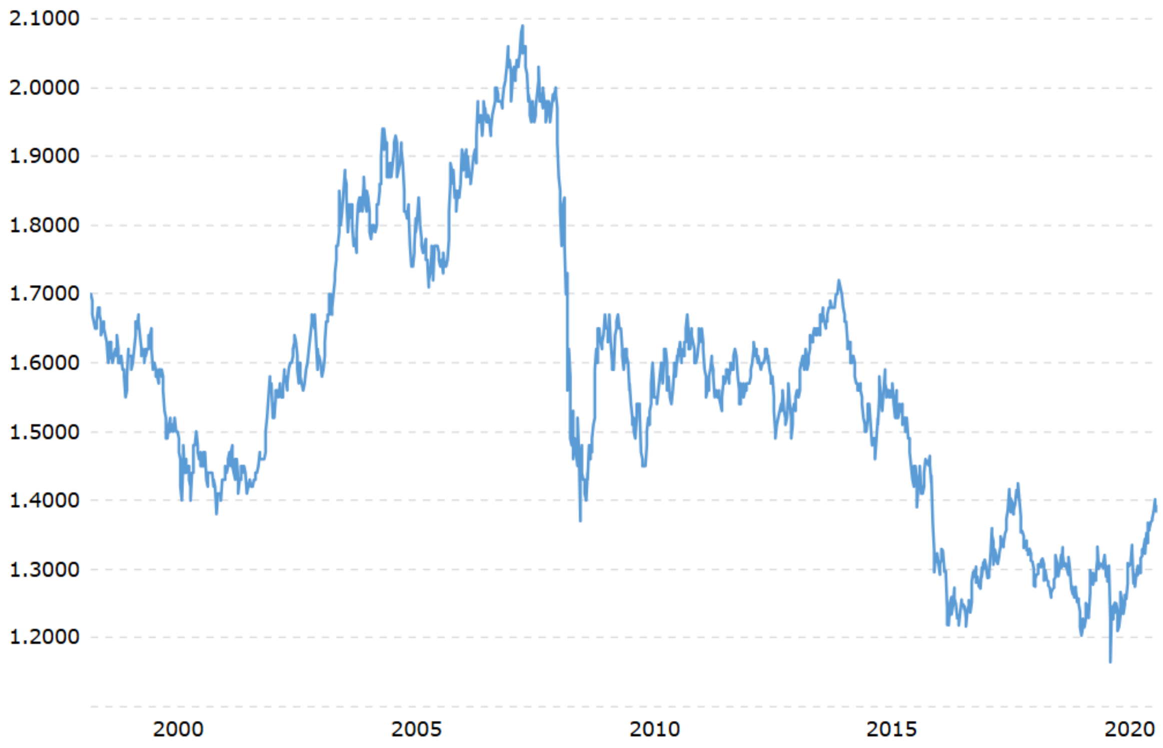

Figure 7: Pound-Dollar Exchange Rate (Macrotrends)

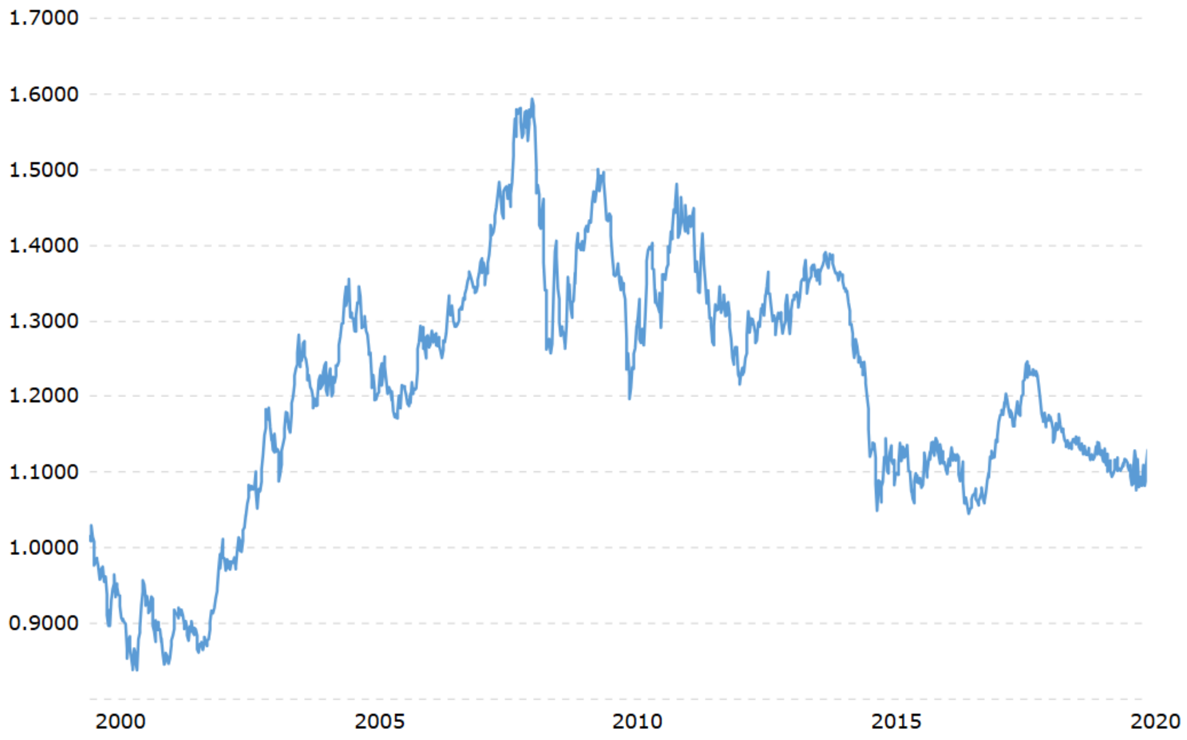

Figure 8: Euro-Dollar Exchange Rate (Macrotrends)

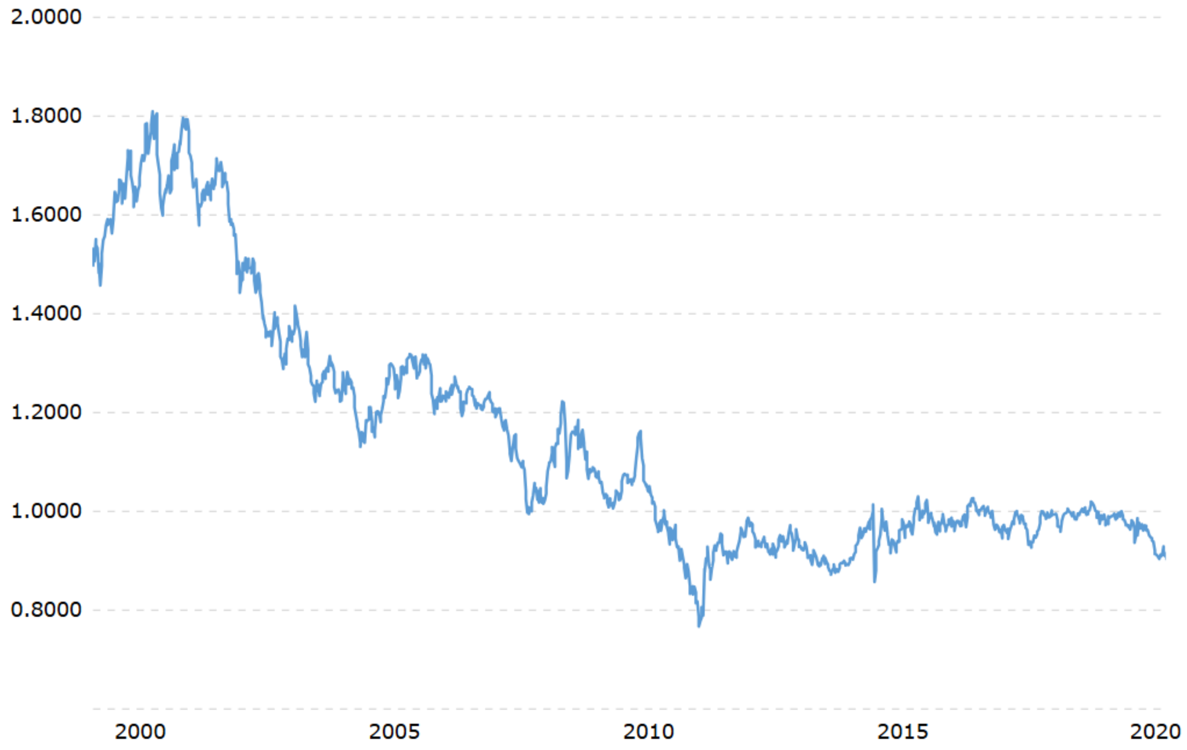

Figure 9: Swiss Franc-Dollar Exchange Rate (Macrotrends)

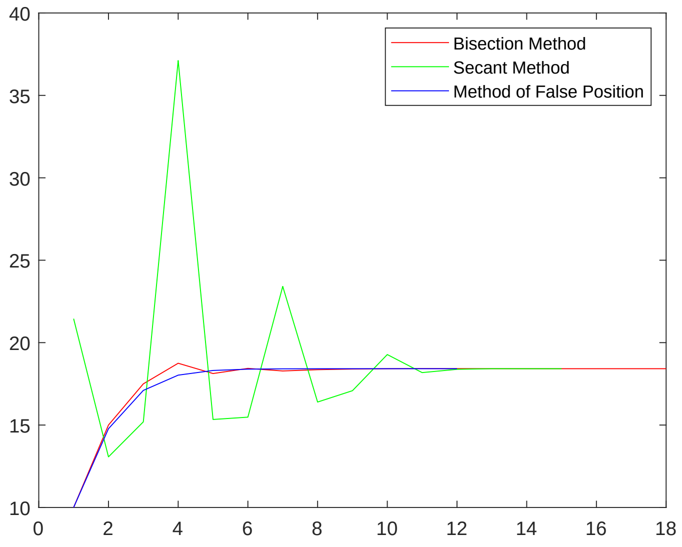

Figure 10: Convergence Plot of Pound-Euro Exchange Rate

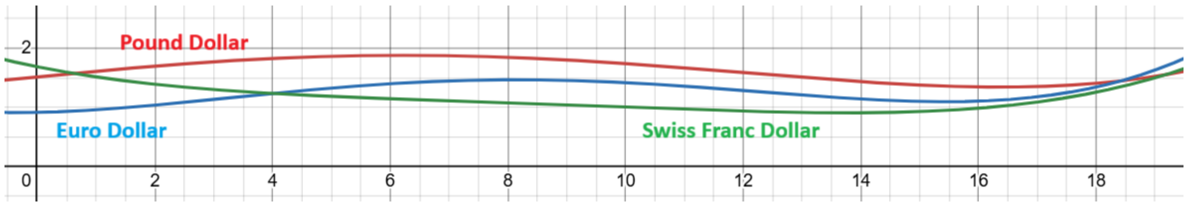

Figure 11: Exchange Rate Plot of Polynomials

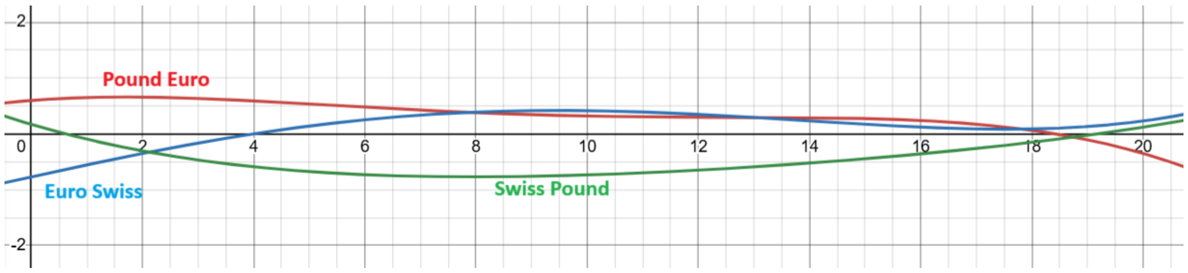

Figure 12: Zero Finding Plot of Polynomials