```matlab
%% combined code script

%% find the fractal boundary for x in [-2,1] for 10^3 points

% initiate points
N= 1000;
X = linspace(-2, 1, N);
Y = nan(size(X)); %empty list for boundary y values

% loop over every x value to find where the boundary is
for k = 1:N
    x = X(k);                    % updates x value
    fn = indicator_fn_at_x(x);  % tests each point

    % if a point is inside (-1) find where it switches to (+1)
    if fn(0) < 0
        Y(k) = bisection(fn,0,2);
    end
end

% plot the results
plot(X,Y,'.');
xlabel('x');
ylabel('imaginary boundary y');
title('Top boundary of mandelbrot set');
grid on;


%% fit a polynomial to the boundary

% set polynomial order = 15
degree = 15;

% filter only the real values
filterVals = ~isnan(Y) & (X > -2) & (X < 0.25);
xfilter = X(filterVals); % filtered x values
yfilter = Y(filterVals); % filtered y values
p = polyfit(xfilter, yfilter, degree); % fit polynomial

%make a smooth x-range for plotting
xSmooth = linspace(min(xfilter),max(xfilter),1000);
% evaluate the fitted polynomial at the x points
ySmooth = polyval(p, xSmooth); % evaluate the fitted polynomial

% plot the original data and the smooth curve
figure;
plot(xfilter, yfilter, '.'); hold on;
plot(xSmooth, ySmooth, 'r-', 'LineWidth',1.5);
xlabel('x'); ylabel('y');
title('15 Degree Polynomial Approximation of the Fractal Boundary');
grid on;


%% compute the curve length of the fitted polynomial

s = min(xfilter);   % lower bound
e = max(xfilter);   % upper bound

% get the curve length using the poly_len function
curveLength = poly_len(p, s, e);   % only represents half of the boundary

```

```
% Since the curve length represents only half of the boundary, double it for the full length
fullOutline = 2 * curveLength;

% Display the computed full outline length
%answers the final question on the assignment
disp(['Full outline length of the fractal boundary: ', num2str(fullOutline)]);


%%%%%%%%----------------------%%%%%%%%%%%%

% write a function that returns how many
% iterations it takes for a point to diverge (|z|>2.0)

function it = fractal(c)
% fractal returns the # of iterations until divergence for mandelbrot
% input c - complex point (c = 1 + 1.5i)
% output it - # of iterations when |z| > 2.0
%   if divergence is not detected within 100 iterations, return 100

z = 0; % Initialize z as a complex number
maxIt = 100; % set max iterations = 100

for it = 1:maxIt
    z = z^2 + c; % Update z using the Mandelbrot formula
    if abs(z) > 2.0
        return; % Exit the function if divergence is detected
    end
end

it = maxIt; % If no divergence, set it to max iterations
end

%%%%%%%%----------------------%%%%%%%%%%%%


function fn = indicator_fn_at_x(x)
% returns an indicator function along a vertical line at given x
%   fn(y) = -1 if (x + 1i*y) is inside (no divergence within 100 iterations)
%           +1 if (x + 1i*y) is outside (diverges within 100 iterations)

fn = @(y) indicator_value(x, y);
end

function val = indicator_value (x, y)
    c = x + 1i*y;
    it = fractal(c);
    if it == 100
        val = -1; % (inside) Indicates divergence
    else
        val = +1; % (outside) Indicates no divergence
    end
end


%%%%%%%%----------------------%%%%%%%%%%%%


function m = bisection(fn_f, s, e)
% finds boundary where the indicator goes from -1 (inside) to +1 (outside)

    fs = fn_f(s); % indicator at lower bound
```

```
    fe = fn_f(e); % indicator at upper bound

    % check if sign change is correct

    if fs >= 0 || fe <= 0
        error('fn_f(s) must be < 0 and fn_f(e) must be > 0');
    end

    % Initialize the tolerance
    tolerance = 1e-6;
    maxIter = 60; % max iterations

    for iter = 1:maxIter
        m = (s + e) / 2; % midpoint
        fm = fn_f(m); % evaluate function at midpoint

        % break if interval is small enough
        if abs(e - s) < tolerance
            break; % root found within tolerance
        end

        % keep the half that still contains the sign change
        if fm > 0
            e = m; % update upper bound
        else
            s = m; % update lower bound
        end

        % final midpoint estimate

        m = (s+e)/2;
    end
end

%%%%%%%%----------------------%%%%%%%%%%%%

function L = poly_len(p, s, e)
% p fitted polynomial coefficients
% s (left bound on x) e (right bound on x)
% L = curve length [s, e]

    % take the derivative of the polynomial
    dp = polyder(p);                        % new coefficients for the derivative

    % make the function sqrt(1 + (P'(x))^2)
    ds = @(x) sqrt( 1 + (polyval(dp, x)).^2 );
    % integrate that function from s to e to get the length
    L = integral(ds, s, e);                 % built-in integral function
end
```