CMPS 251 - Assignment 4

Kinan Dak Al Bab - 201205052

October 2, 2014

1 Secant Method

1.1 Question

- Solve the univariate equation $x^3 - 2x - 5 = 0$ in the interval $2 \leq x \leq 3$ to an accuracy of 10^{-6} using the secant method.
- Compare your solution with the one produced by the built-in $fzero()$ function.
- Show the sequence of iterates produced and the corresponding function values. Do you observe slower convergence than with Newton's method of the last assignment?
- What is the theoretical rate of convergence of the secant method? Check if your results converge at this rate.

1.2 Code

```

format long

x = [2.5]; % start from the middle of the range
f_x = (x(1)^3 - 2*x(1) - 5);

x = [x x(1) - (f_x / (3*x(1)^2 - 2))]; % using newton's method ...
    to bootstrap the secant method.
new_f_x = (x(2)^3 - 2*x(2) - 5)

f_x = [f_x new_f_x];

i = 2;
while abs(f_x(i)) > 10^-6
    secant_slope = (f_x(i) - f_x(i-1)) / (x(i) - x(i-1)); % ...
        calculate the secant slope
    new_x = x(i) - (f_x(i) / secant_slope); % use the secant ...
        slope as an approximation to the value of the first ...
        derivative
    x = [x new_x];

    new_f_x = (new_x^3 - 2*new_x - 5) % calculate new value of ...
        function
    f_x = [f_x new_f_x]; % save history

    i = i + 1;
end

fzero(@fx, [2 3]) - x(i) % @fx: function handle for x^3 - 2x - 5
```

1.3 Output

```
new_f_x =

   0.807945126228955


new_f_x =

   0.149580733496334


new_f_x =

   0.005632247121143


new_f_x =

    4.193179948686066e-005


new_f_x =

    1.190649001614474e-008


ans =

   1.066752020051354e-009 % Absolute Difference between x of fzero
and x of secant.
```

1.4 Comment

- When comparing the result of the secant to that of $fzero()$, the differance was smaller than 10^{-6}, which means that we reached the desired accuracy. i suspect that by increasing the required accuracy (the termination condition of the while loop) we get closer and closer to the result of $fzero()$.
- We observe slower convergance than in newton's method, in newton's method it took me 4 iterations to reach the designated accuracy, while it

took me 5 iterations here (I didnt count the iteration for $x = 2.5$ in both methods). Also, it is noticable that the last result of newton's method was accurate to 10^{-11}, while in the secant and after 5 iterations we only got to an accuracy of 10^{-8}.

- If we take the last three values for new_f_x we can construct a set of two equations with two unknowns:

$$E^k = C \times (E^{k-1})^\alpha$$

$$E^{k-1} = C \times (E^{k-2})^\alpha$$

  By solving for C and α, we can get the convergance rate (nearly $\alpha = 1.6 The golden ratio$). We can solve the system by dividing the two equations, then taking the log of each side.

2 System of two nonlinear equations

2.1 Questions

- Use Newton's method to solve the system of equations in problem 1(b) on p 287.

2.2 Code

```
x = [0.2; 0.2]; % inital guess is x1 = 1, x2 = 1

f1_x = x(1) + x(2) - 2 * x(1) * x(2);
f2_x = x(1)^2 + x(2)^2 - 2 * x(1) + 2 * x(2) + 1;

F_X = [f1_x; f2_x];
n = norm(F_X);
while n > 10^-6
    J = [1 - 2 * x(2) 1 - 2 * x(1); 2 * x(1) - 2 2 * x(2) + 2]; ...
        %Jacobian

    x = x - (J \ F_X); % get new x by solving a set of linear ...
        equations

    f1_x = x(1) + x(2) - 2 * x(1) * x(2);
    f2_x = x(1)^2 + x(2)^2 - 2 * x(1) + 2 * x(2) + 1;

    F_X = [f1_x; f2_x]; % get new f1(x), f2(x)
    n = norm(F_X) % calculate the norm
end

x % print the solution
```

2.3 Output

```
n =

   0.241007402171042


n =

   0.016343621261882


n =

    1.724036927189438e-004


n =

    9.271449752678968e-009


x =

   0.215760917061070
  -0.379541244437223


F_X =

  1.0e-008 *

   0.654839701907939
   0.656340437554803
```

2.4 Comment

By changing the inital guess, we can reduce or increase the number of iterations required to get the answer, for an inital guess of $[0.1; 0.1]$ we can get the answer using three iterations, for $[1; 1]$ we get it in 17 iterations, and for $[2; 2]$ we get it in 31 iterations.

The answer we get falls within the designated accuracy, the norm of is in the order of 10^{-9} and the values of the two functions are both smaller than 10^{-6}.

3 Behavior of polynomials

3.1 Question

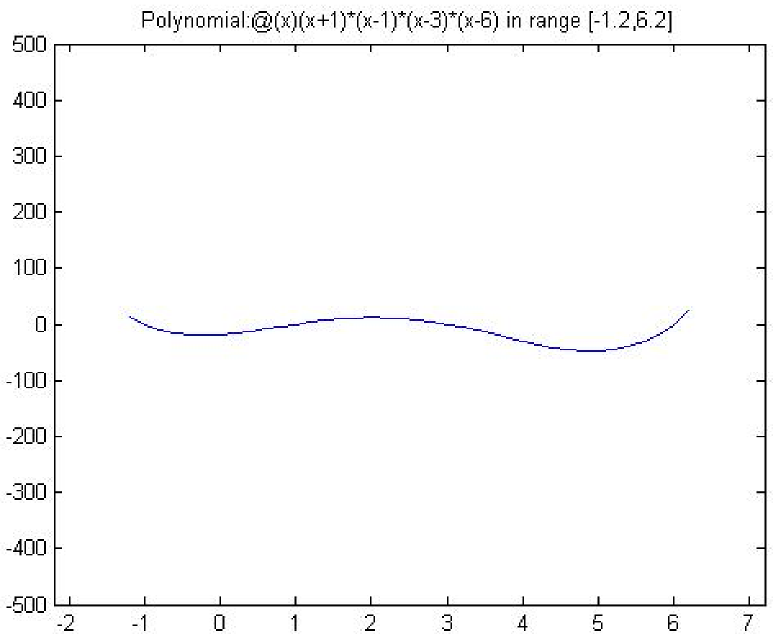
- Consider the following fourth degree (quartic) polynomial: $p(x) = (x+1)(x-1)(x-3)(x-6)$ Plot it in the range $(-1.2, 6.2)$. How many times does the polynomial oscillate? What are the roots of the polynomial? Plot the polynomial in the range $(-2, 7)$, and then in the range $(-3, 8)$ using the same vertical scale. What do you observe? Any explanation?

- Repeat the above for the degree 6 polynomial $p(x) = (x+3)x(x-1)(x-4)(x-5)(x-8)$. Explore its behavior in the range $(-3, 8)$ and then $(-4, 9)$.

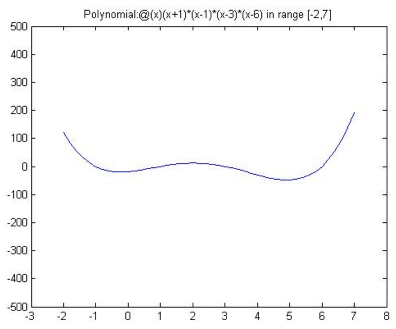
3.2 code

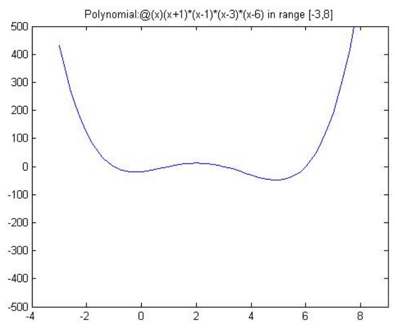
```
%% This function takes a polynomial and plots it within the ...
     range [a,b].
%% The plot y-axis spans between the given ymin, ymax.

function f_x = polynomialBeh(px, a, b, ymin, ymax)

x = [];
f_x = [];
for i = a:0.2:b
    x = [x i];
    f_x = [f_x feval(px, i)];
end

plot(x, f_x)
axis([a-1 b+1 ymin ymax]);
title(strcat('Polynomial: ', func2str(px), ' in range [', ...
     num2str(a), ',', num2str(b), ']'));
end



>> syms x; %% Create the polynomials and plot them.
>> px = @(x)(x+1)*(x-1)*(x-3)*(x-6);
>> polynomialBeh(px, -1.2, 6.2, -500, 500);
>> polynomialBeh(px, -2, 7, -500, 500);
>> polynomialBeh(px, -3, 8, -500, 500);

>> px = @(x)(x+3)*x*(x-1)*(x-4)*(x-5)*(x-8);
>> polynomialBeh(px, -3, 8, -4000, 4000);
>> polynomialBeh(px, -4, 9, -4000, 4000);
```

3.3 First Polynomial



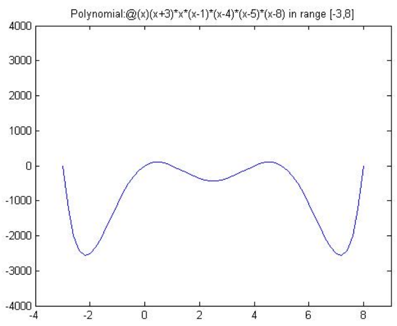
The Polynomial oscillates 3 times. which is expected, the first derivative of the polynomial f‘ would be of degree 3, thus having three roots for f‘$(x) = 0$.
The Roots of the polynomial are: $-1, 1, 3, 6$. The roots can easily be determined from the factorized form of the polynomial (or by looking at the plot).

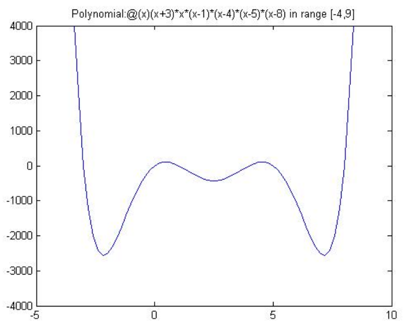




The Polynomial goes to infinity as $|x|$ increases, A polynomial's order of growth is that of its highest term, in this case power of four. as $|x|$ increases the other terms start to participate less in shaping the polynomial (cause the highest the power the highest the acceleration). this causes the oscillations to grow bigger and bigger in magnitude as we approache the sides of the polynomial until we reach a point when it goes to infinity. The function goes to infinity from both sides because the highest power is even (thus negative values of x gets positive values when the power is applied) with a positive coeffiecent (that can be checked by transforming the polynomial to its power form).

3.4 Second Polynomial





This time the polynomial oscillates 5 times, which is expected because its first derivative is of order 5, The roots are also those determined by the factorized form: $-3, 0, 1, 4, 5, 8$. As noticed before the oscilliations get larger in magnitude towards the sides of the polynomial (as $|x|$ increases). the same justification applies here too, the polynomial goes to positive infinity from both sides because the largest power is even with a positive coeffiecent.

4 Power form of the interpolating polynomial

4.1 Question

- Write a matlab function (or script) to compute the coefficients of a polynomial interpolant for a set of points $(x_i, y_i), i = 1 \dots n$. If you are writing a function, its header could be:

$$function\ a = interp(x, y)$$

- Write a function or script that evaluates the polynomial defined by the coecients a at a given set of coordinates u. This should produce a vector v where $v_i = p(u_i), i = 1 \dots m$. If you are writing a function, its header could be:

$$function\ v = peval(a, u)$$

- Test your script or functions using the following data:

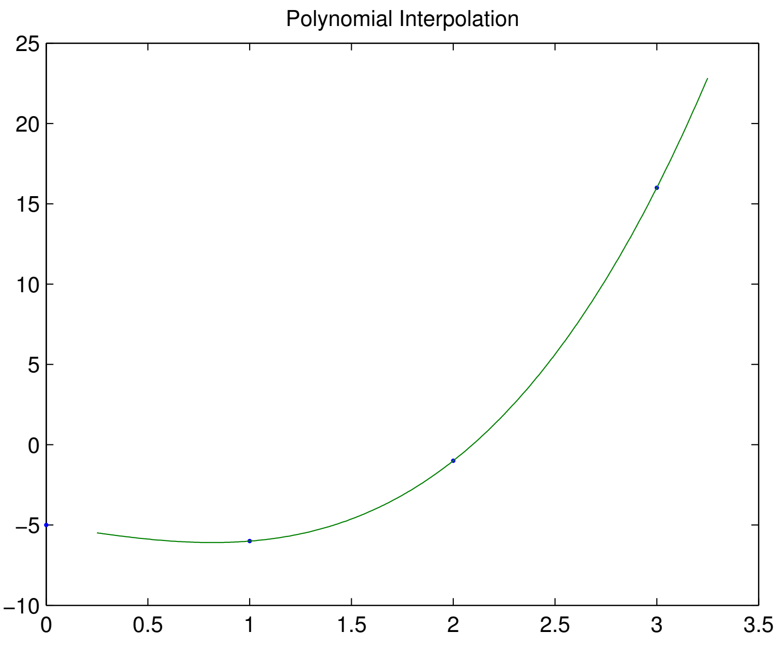
$$x = 0 : 3;\ y = [-5 - 6 - 116];\ u = .25 : .01 : 3.25$$

  by plotting the interpolating polynomial and marking the (x, y) points with little circles on the plot.

4.2 Code

```
%% Polynomial Interpolation
function a = interp(x, y)
j = length(x);

a = [];
cy = [];
for i = 1:j
    xi = x(i);
    yi = y(i);

    v = [1];
    for p = 1:j
        v = [v xi^p];
    end

    a = [a; v];
    cy = [cy; yi];
end

a = a \ cy;
end


%% Polynomial Evaluation
function v = peval(a, u)
v = [];

j = length(a);
i = length(u);

for k = 1:i
    x = u(k);

    vx = [1];
    for p = 1:j-1
        vx = [vx x^p];
    end

    v = [v vx * a];
end

end

%% Testing Script
>> x = 0:3;
>> y = [-5 -6 -1 16];
>> u = 0.25:.01:3.25;
>> a = interp(x, y);
>> v = peval(a, u);
>> plot(x, y, '.', u, v);
>> axis([-0.1 3.5 -10 25]);
>> title('Polynomial Interpolation');
```

4.3 Output



4.4 Comment

Clearly, The interpolated polynomial passes through the 4 points (because of the way we constructed it). We can tell from the plot that this polynomial has one local minima in the range $[0.25, 3]$. We can also tell that the resulting polynomial goes to negative infinity as x gets smaller, so the polynomial should have a local maxima in the range $]-\inf, 0.25[$, so we know that the polynomial osiciliates at least 2 times. so the resulting polynomial is of degree 3 or 4, but if we printed the vector of coeffiecents a, we would find that the last coeffiecent (that of x^4) is zero, therefore the resulting polynomial is of degree 3. This might be due to some dependency between the 4 given points. 3 of them might fall on the same parabola, in which case we are only given 2 pieces of information by these three points and not 3.