CMPS251 - Assignment 6

Kinan Dak Al Bab, 201205052

October 24, 2014

Note: MATLAB R2010b (the version in the labs) was used to solve this assignment.

1 Lagrange form of the polynomial

1.1 Question

Consider the interpolation of the following $n = 6$ points.

$$x = linspace(-1, 1, 6);$$

$$y = sin(\pi * x) + cos(2 * \pi * x);$$

- Compute the n terms $w_i = \frac{1}{\prod_{j \neq i}(x_i - x_j)}$
- verify that the Lagrange polynomial interpolant can be written as:

$$p(x) = \prod_{j=0}^{n-1}(x - x_j) \sum_{i=0}^{n-1} \frac{w_i y_i}{x - x_i}$$

  and can therefore be evaluated at any point with $O(n)$ floating point operations.

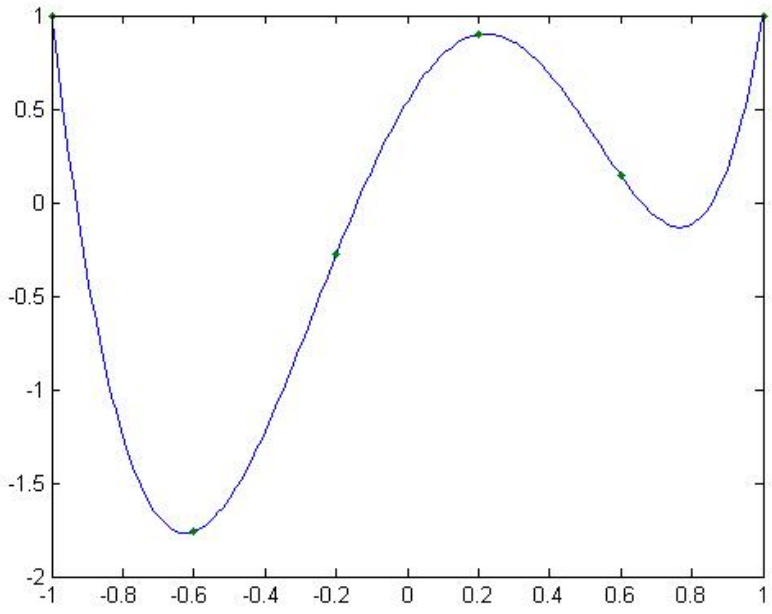
- evaluate the polynomial at the points $u = -1 : 10^{-2} : 1$ and plot it.

1.2 Code

```
x = linspace(-1, 1, 6);
y = sin(pi*x) + cos(2*pi*x);

n = length(x);

w = zeros(1, n);
for i = 1:n
    tmp = x(i) - x;
    w(i) = 1 / prod(tmp(tmp ≠ 0)); % compute w(i)
end

u = -1:10^(-2):1;
f_u = zeros(1, length(u));
f_u_ = zeros(1, length(u));
```

```
15
16
17  for j = 1:length(u) % O(n) lagrange evaluation
18      [mem loc] = ismember(u(j), x); % refer to comment section, ...
            bullet 3
19      if mem
20          f_u(j) = y(loc);
21      else
22          f_u(j) = prod(u(j) - x) * sum((w .* y) ./(u(j) - x));
23      end
24  end
25
26  for j = 1:length(u) %O(n^2) lagrange evaluation
27      f_u_(j) = 0;
28      for i = 1:n
29          p = 1;
30          for k = 1:n
31              if k ≠ i
32                  p = p * ((u(j) - x(k)) / (x(i) - x(k)));
33              end
34          end
35
36          f_u_(j) = f_u_(j) + (p * y(i));
37      end
38  end
39
40  norm(f_u - f_u_) % verify two forms are equivalent
41
42  plot(u, f_u);
```

1.3 Plot



(Plotting the lagrange polynomial in its fast form against given u).

1.4 Output

```
ans =

  3.5196e-015 % the norm
```

1.5 Comment

- Notice that the norm of the difference is very close to zero, in other words, the original lagrange form, and the "faster" lagrange evaluate to the same value for the same x, i,e, the norm verifies that the lagrange polynomial can be written in the new form and evaluated in $O(n)$. Remember that the interpolant polynomial is unique.
- The derivation of the new form from the original form is in the book and the notes, I will not re write it here. but the derivation also shows equivalence.
- However, in the derivation, there is a step in which we group all $x - x_j$ outside of the sum and then divide it by $x - x_i$ to make the two loops (the product and the sum) independent (and thus achieve $O(n)$). In a derivation we cant divide by any quantity unless we show it to be not equal to zero, however in this case we know that this quantity can be zero when x is equal to the x of one of the n interpolated points. this will give us error in the evaluation (divide by zero yields infinity, then we multiply that by zero which is undefined, shown in matlab as NaN). So during evaluation we check if the given $x = u$ is one of the original points, and if it is we evaluate it to be equal to the corresponding y of that point.

2 Newton form of the polynomial

2.1 Question

Consider again the same set of interpolation points above.

- compute the first, second, third, fourth, and fifth divided differences and put them in a triangular array as shown on page 308.
- explain how you can evaluate the Newton interpolant using $O(n)$ floating point operations.

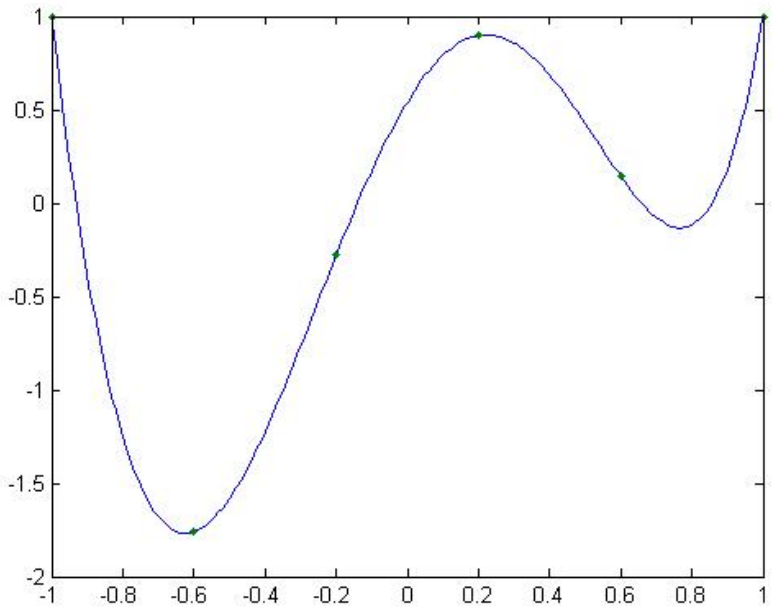
$$p(x) = \sum_{i=0}^{n-1} f[x_o, \ldots, x_i] \prod_{j=0}^{i-1} (x - x_j)$$

- evaluate the polynomial at the points $u = -1 : 10^{-2} : 1$ and plot it. Comment.

2.2 Code

```
x = linspace(-1, 1, 6);
y = sin(pi*x) + cos(2*pi*x);
n = length(x);

D = zeros(n, n); % Divided Differences Matrix
D(1:n, 1) = y(1: n);

for i = 2:n
    D(i:n, i) = (D(i:n, i-1) - D(i-1:n-1, i-1)) ./ (x(i:n) - ...
        x(1:n-i+1))';
end
d = diag(D); % Diagonal of the Divided difference matrix

u = -1:10^(-2):1; % evaluate at u
f_u = zeros(1, length(u));
for i = 1:length(u)
    tmp = 1;
    for k = 1:n
        f_u(i) = f_u(i) + (d(k) * tmp);
        tmp = tmp * (u(i) - x(k));
    end
end

plot(u, f_u, x, y, '.') % plot
```

2.3 Output



(Plotting the Newton polynomial in its fast form against given u).

2.4 Comment

- Explanation of $O(n)$ Evaluation: At the first glance the formula seems to give us $O(n^2)$ flops, because it has two nested loops that depend on one another, but at a closer look we can see that the inner loop (the product) is basically that of the previous iteration plus one extra term.

$$p(x) = \sum_{i=0}^{n-1} f[x_o, \ldots, x_i] \prod_{j=0}^{i-1} (x - x_j)$$

$$= f[x_0] + f[x_0, x_1] \times (x - x_0) + f[x_0, x_1, x_2] \times (x - x_0)(x - x_1) \ldots$$

  So instead of evaluating the shared terms each time, we can simply store them and multiply the stored value by the value of the extra new term, in which case we reduce the inner loop (product) from being a loop to being just one multiplication. thus resulting in a total of $O(n)$.

- If we compare the two plots (from this question and the previous question) we can see that both forms yielded the same plot, that's expected because we have a unique polynomial of degree n that passes through n fixed points. In other words, The interpolating polynomial is unique, therefore Newton's form, Lagrange form, and the power form are not different polynomial, they are the same polynomial expressed in different forms, so they should all evaluate to the same value for the same x.

3 Comparison of the different forms of the polynomial interpolant

3.1 Question

Very briefly, comment on the advantages and disadvantages of the three forms we considered for representing polynomial interpolants (see page 312).

3.2 Answer

The power form main advantage is that it is the form we learned in high school and we are most familiar with, it is quite simple to understand and visualize. however, it has two main disadvantages, in practice, the matrix which is used to calculate the polynomial coefficient from has high conditioning causing errors in the coefficient, also it takes $O(n^3)$ to construct.

The Lagrange Polynomial avoids the conditioning problem, there is no set of linear equations to be solved, it is quite stable and it takes $O(n^2)$ to construct, and also $O(n)$ to evaluate (but with a higher constant than Newton or power).

Newton form is very similar to Taylor, it doesn't have conditioning problems, it is also very simple and takes $O(n^2)$ to construct and $O(n)$ to evaluate (with lower constants than Lagrange).

4 Error in interpolating polynomial

4.1 Question

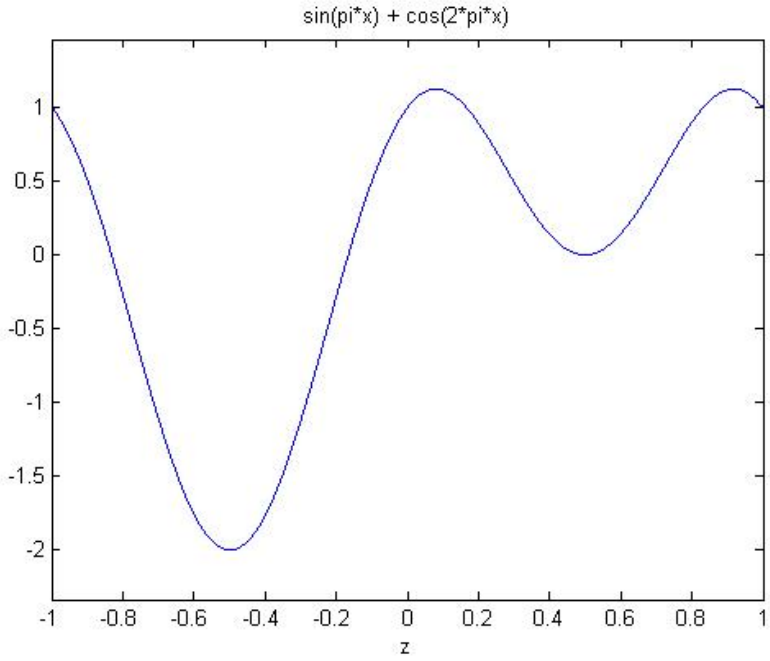
Running out of time, I will write the question here if I have time at the end.

4.2 Code

```
x = -1:10^(-2):1; % for Plotting purposes
y = sin(pi*x) + cos(2*pi*x);

syms z;
f = sin(pi*z) + cos(2*pi*z);

n = 1;
error = inf;
df = diff(f);
while error > 0.5 * 10^(-1) % 0.5 for rounding errors
    df = diff(df);
    n = n + 1;
    h = 2 / (n - 1);

    C = max(subs(df, x));

    error = C * h^(n) / (4 * n);
end

uniform = n
```

```
21
22  n = 1;
23  error = inf;
24  df = diff(f);
25  while error > 0.5 * 10^(-1) % 0.5 for rounding errors
26      df = diff(df);
27      n = n + 1;
28
29      C = max(subs(df, x));
30      error = C / (2^(n-1) * factorial(n));
31  end
32
33  chebyshev = n
34
35  plot(x, y)
36  title('sin(pi*x) + cos(2*pi*x)');
```

4.3 plot



4.4 Output

```
uniform =

    13


chebyshev =

    11
```

4.5 Comment

Chebyshev requires less points than uniform, which is expected because Chebyshev puts more points towards both ends of the interval where more oscillations (and thus more error) occurs. In this case I derived the function with each increment of n because the maximum of each derivative depends on the derivative.

NOTE: we can have an upper bound without deriving at each iteration, At each derivation we get π and $2*\pi$ out of the sin/cosine (and the sin and cos flip), so at maximum we will have $\pi^n + 2^n \times \pi^n = (2^n + 1)\pi^n$. when I used this as a bound, it gave me similar results, however I wanted to try symbolic functions out.

5 Inverse interpolation

5.1 Question

No Time For writting the question

5.2 Code

```
format long

x = 0:0.5:2;
y = exp(x) - 2;

q = polyfit(y, x, length(x)-1); % why not use some matlab built ...
    in tools ;)
root = polyval(q, 0) % the solution by evaluating inverse polynomial

abs(exp(root) - 2) % absolute value of the error
```

5.3 Output

```
root =

   0.698223843249594


ans =

   0.010179141551540
```

5.4 Comment

This works because the function given is one-to-one (has an inverse function). Also The error might be a bit large because we interpolate based on four points, if we decrease the step from 0.5 to 0.1 we get something in the order of 10^-11. I used matlab built in functions to get familiar with it, save time, and because I

am assuming that it is not the point of this exercise as I already wrote scripts to interpolate a set of points (in 3 different forms) and I could have easily re-used any of them.