Part 1 Interpolation

a) Write a function to compute the divided difference.

Solution:
The following is Matlab code to compute the divided difference given a function f and a list of x values. A zero matrix is initialized to the size of the x input vector. The first column of this matrix is initialized to $f(x)$ for each value in the x input vector. This first column is used to start off the divided difference computation in the following nested for loop structure. The matrix is filled in as per the code discussed on the board in class. This algorithm is used later in Algorithm 3: lagrange interpolation.

Algorithm 1: divided difference

```
function dd = divided_difference(f, x)

dd = zeros(length(x), length(x));
dd(:, 1) = double(subs(f,x));

for i = 2:length(x)
    for j = 2:i
        dd(i,j) = (dd(i,j-1) - dd(i-1,j-1)) / (x(i) - x(i-j+1));
    end
end
dd = diag(dd);
end
```

b) Write a function to compute the polynomial using nested multiplication:

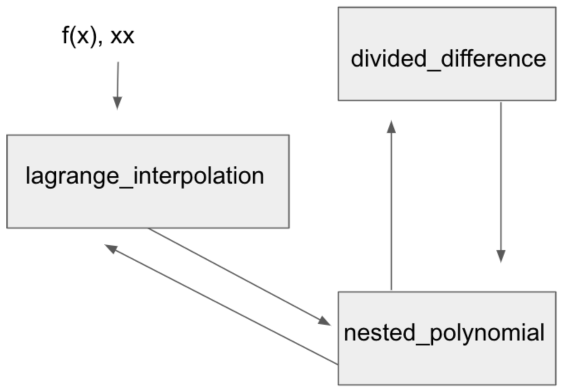
Solution:
The following is Matlab code to compute the nested polynomial $p(x)$ for the lagrange interpolation function.

Algorithm 2: nested polynomial

```
function p_x = nested_polynomial(a, xx)
syms z
p_x = a(1);
for i = 2:length(xx)+1
    term = 1;
    for j = 1:i-1
        term = term*(z-xx(j));
    end
    term = term * a(i);
    p_x = p_x + term;
end
end
```

c) Combine a) and b) to write a package for the Lagrnge polynomial interpolation.
Solution:

The following is Matlab code utilizing the two previous functions in order to compute lagrange polynomial interpolation. A flow chart showing the relationship between these three algorithms has also been included.



Algorithm 3: Lagrange polynomial interpolation

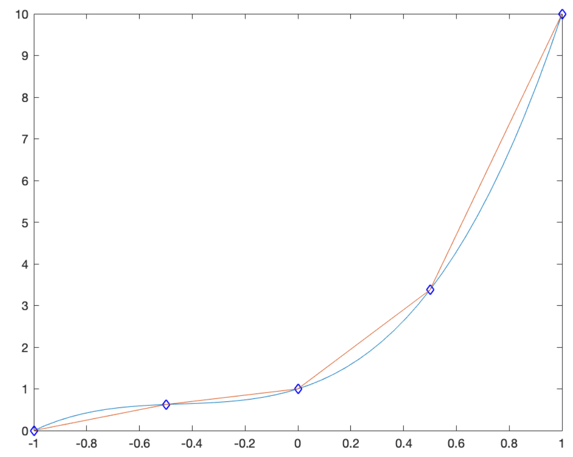
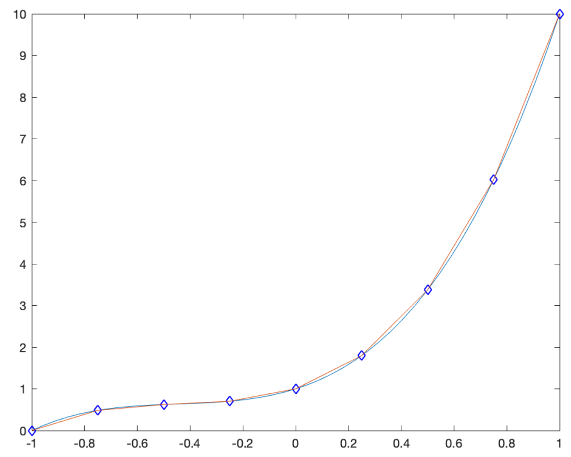
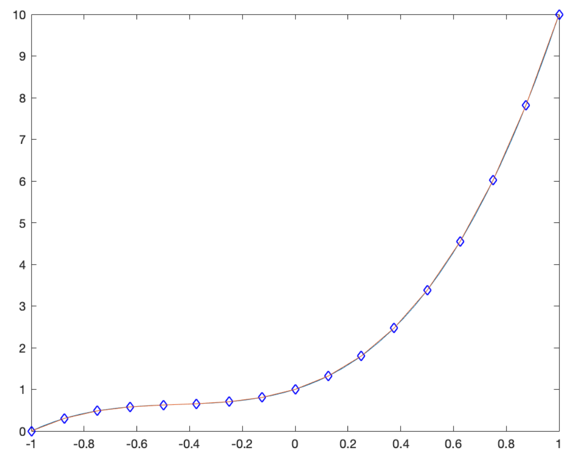
```
function [np, re] = lagrange_interpolation(f,xx)

dd = divided_difference(f, xx);
np = nested_polynomial(dd, xx(1:end-1));

y = subs(f,xx);
y_lagrange = subs(np,xx);

re = double(abs(y - y_lagrange) ./ y);
end
```

d) Use your package to obtain the Lagrange polynomial interpolation for the following three functions with n equally spaced points $\in [a,b]$

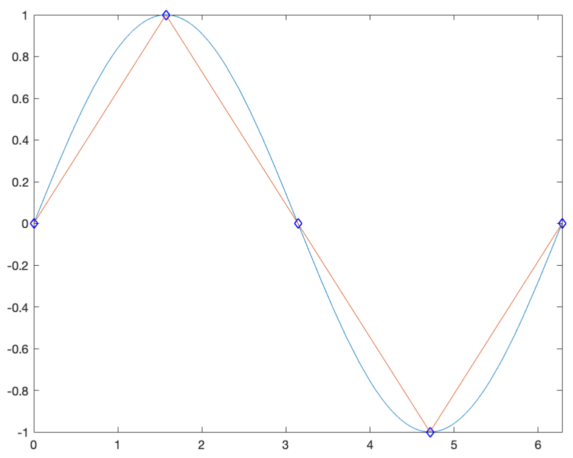
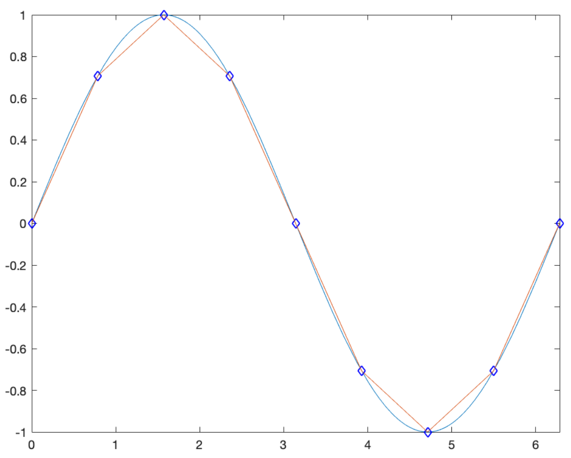
1) $f(x) = 3x^3 + 4x^2 + 2x + 1$ with $a = -1$, $b = 1$
2) $f(x) = \sin x$ with $a = 0$, $b = 2\pi$
3) $f(x) = \frac{1}{(1+25x^2)}$ with $a = -1$, $b = 1$

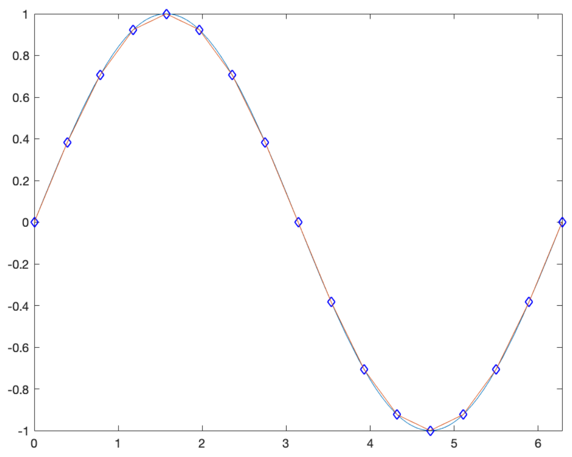
Solution:

1) $f(x) = 3x^3 + 4x^2 + 2x + 1$ with $a = -1$, $b = 1$, $n = 4, 8, 16$

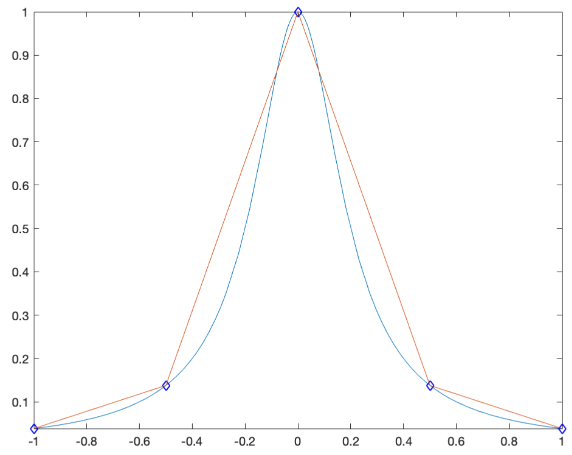




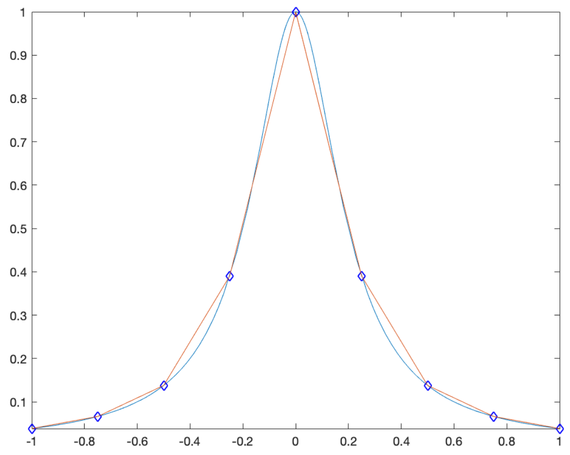
2) $f(x) = \sin x$ with $a = 0$, $b = 2\pi$, $n = 4, 8, 16$

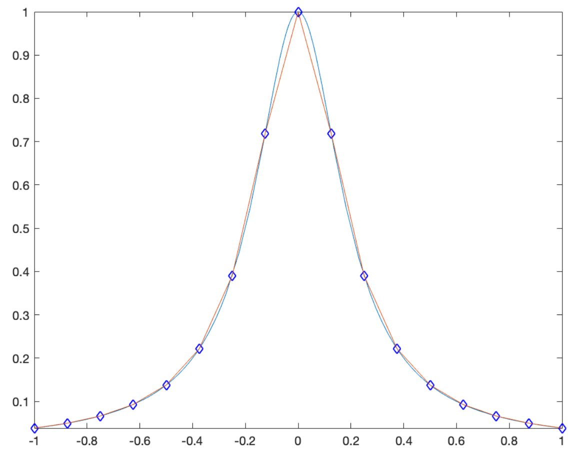





3) $f(x) = \frac{1}{(1+25x^2)}$ with $a = -1$, $b = 1$, $n = 4, 8, 16$







These plots demonstrate the correctness of the Algorithms 1,2, and 3. The plots show that the expected output aligns with the observed output of Algorithm 3, which relies on Algorithms 1 and 2. The function $f(x)$

is plotted on the same graph as the Lagrange interpolation output, shown as blue diamonds. Additionally, it can be seen in the above plots for each of the three given functions that a more accurate interpolation is achieved with larger values of n.

Part 2 Order of accuracy

Demonstrate the order of accuracy computationally for the cubic spline using Matlab. Construct at least two examples (at least two different non-trivial functions) to show the order of accuracy. Please try two different boundary conditions implemented in Matlab.

Solution:

Algorithm 4: order of accuracy

```
function p = spline_order_accuracy(f,a,b,n, boundary, plot)

    %boundary conditions: start & end slope
    start_slope = double(subs(diff(f),a));
    end_slope = double(subs(diff(f),b));

    %n random uniform numbers on the interval [a,b]
    x_sim = a + (b-a) * rand(n,1);
    if boundary
        y_sim = [start_slope double(subs(f,x_sim)).' end_slope].';
    else
        y_sim = double(subs(f,x_sim));
    end
    %compute spline
    h = (b - a) / (n-1); % h is step size
    xx = a:h:b;
    y = double(subs(f,xx));
    yy = spline(x_sim, y_sim, xx);

    %compute spline for 2n:
    x_sim2 = a + (b-a) * rand(2*n,1);
    if boundary
        y_sim2 = [start_slope double(subs(f,x_sim2)).' end_slope].';
    else
        y_sim2 = double(subs(f,x_sim2));
    end
    yy2 = spline(x_sim2, y_sim2, xx);

    if plot
        if boundary
            plot(x_sim,y_sim(2:end-1),'o',xx,yy);
        else
            plot(x_sim,y_sim,'o',xx,yy);
        end
        grid on;
    end
    e = norm(abs(y - yy), Inf);
    e2 = norm(abs(y - yy2), Inf);
    p = log(e/e2)/log(2);
end
```

The above algorithm simulates n data points for a given function $f(x)$ over the interval $[a,b]$. The Matlab function spline() is then used to fit a piecewise cubic polynomial to this simulated data. The x values that the spline() function is interpolating on is generated on the interval $[a,b]$ with step size $h = \frac{b-a}{n-1}$. A second set of simulated data was generated using $2n$ data points, and another cubic spline was computed to fit this data. We now have $S_h(x)$ and $S_{h/2}(x)$ that we can compare to $f(x)$ to compute relative error E. We can then use relative error to find the order of accuracy p.

Given: $E_h = |f(x) - S(x)| = Ch^p$ where p is the order of accuracy.

$$\frac{E_h}{E_{h/2}} = \frac{Ch^p}{C(h/2)^p}$$

$$\ln\frac{E_h}{E_{h/2}} = p\ln\frac{h}{h/2}$$

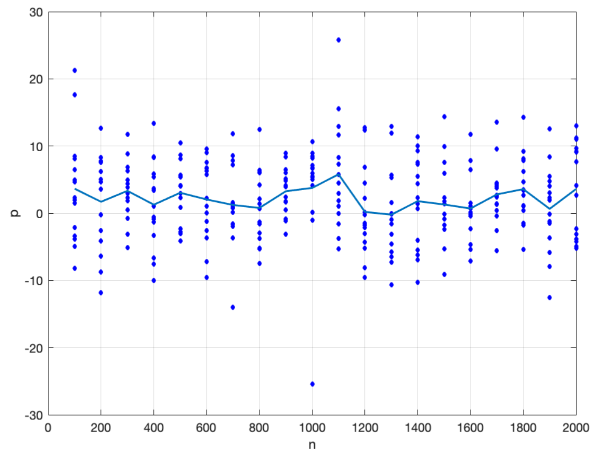
$$p = \frac{\ln E_h/E_{h/2}}{\ln 2} \tag{1}$$

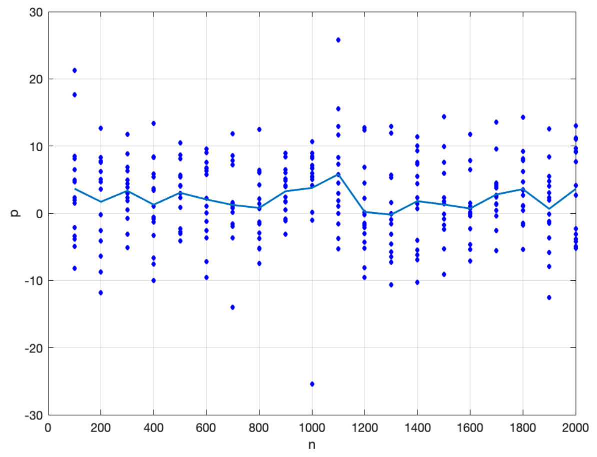
Note that each of the error terms is actually a vector constructed from comparing points from the spline to the actual values from $f(x)$. The order of accuracy p presented in equation (1) is therefore a vector of order of accuracies for each individual data point. The norm of each of these vectors were used in (1) in order to compute p, as follows:

$$p = \ln\frac{||E_h||}{||E_{\mathrm{h}/2}||} \times \frac{1}{\ln 2} \tag{2}$$

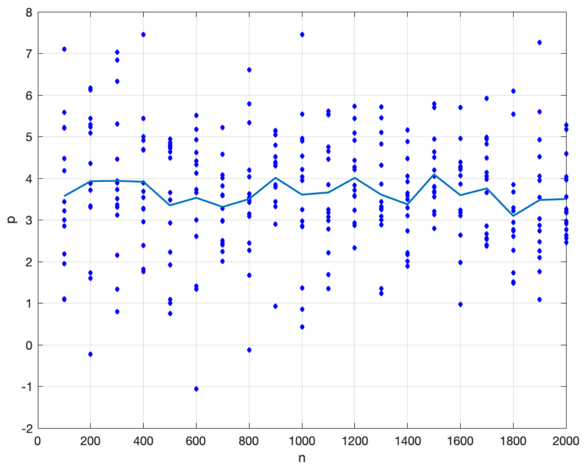
Now we aim to test this algorithm on different functions using different boundary conditions. The graphs on the left column are using the not-a-knot conditions while the graphs on the right column are using the end slopes as the boundary conditions. Conditions were repeated for 15 iterations for different values of n, which are shown as blue diamonds. The mean of these 15 iterations is graphed as a line on each of these plots.

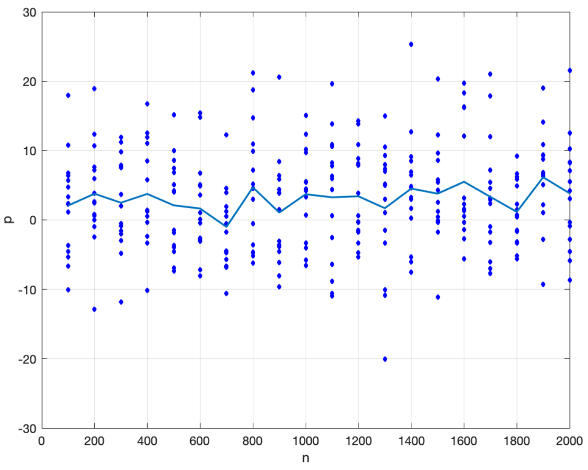
Function 1: $x^{\sin x} \in [1,5]$





Function 2: $\sin x \in [0, 2\pi]$





The not-a-knot boundary condition appears to yield a higher order of accuracy when compared to using the end slopes condition.
In order to test a wider range of n values, the following log scale plot was generated by computing order of accuracy at $n = 10, 100, 1000, 10000$ with 25 iterations at each value of n.