# Problem 1.

In this problem, you can use the Matlab program posted on course website and Canvas (also mentioned in the lecture notes on 11/08) that computes the interpolation polynomial. We want to see how well a given function can be approximated by the interpolation polynomials. Let f be some function. On the interval $[-5, 5]$, take $N$ equally spaced points $-5 = x_1 < x_2 < \ldots < x_N = 5$. Take $N$ points $(x_1, y_1), \ldots, (xN, yN)$ on the graph of $f$.

**a)** For the function $f(x) = \sin(x)$, plot the graph of the interpolation $P$ onthe interval $[-5, 5]$ in the case $N = 3, N = 6, N = 11, N = 21$. What do you notice? Does the interpolation polynomial approximate well the function $f$ on the interval $[-5, 5]$ when $N$ gets larger?

**b)** Repeat part $a$ with the objective function $f(x) = \frac{1}{1+10x^2}$.

## Solution

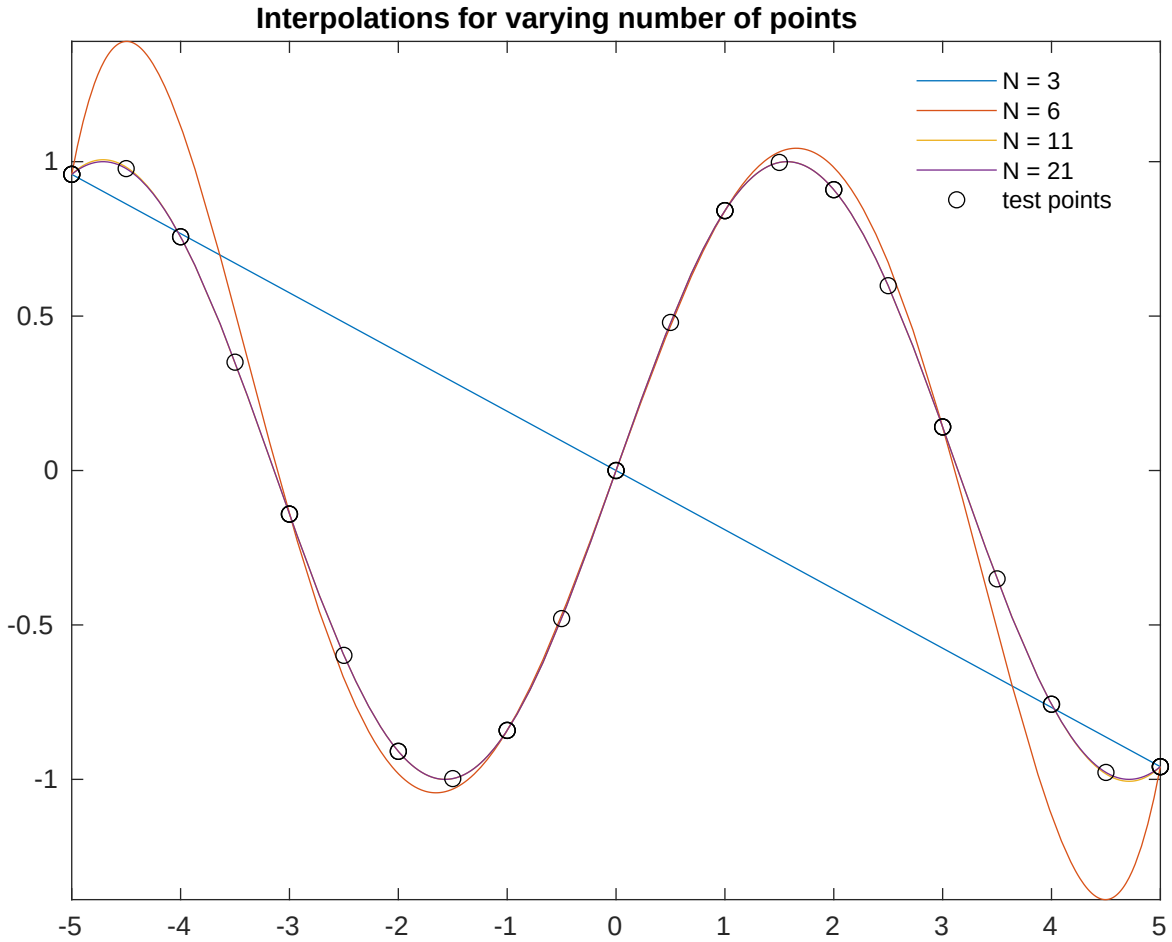
**Grader's note:** Please do not submit all of the following Matlab code on paper, only the plots produced and your analysis/interpretation.

**a)** We do this in Matlab with the following code modified from the starter code posted on the course website.

```
Nlist = [3, 6, 11, 21];  % Make a list of candidates
P = sym('p',[1 length(Nlist)]); % A vector of symbolic objects to store
   our polynomials
legend_key = cell(1, length(Nlist)+1);  % To store the function names for
   the legend

% Create empty vectors to store interpolation points
test_points_x = [];
test_points_y = [];
% it's a best practice to pre-allocate arrays, but the index arithmetic
% can get a bit messy

for Nindex = 1:length(Nlist)  % Loop over the list values
        N = Nlist(Nindex);  % Set N from our list
        legend_key{Nindex} = strjoin({'N =', num2str(N)});
        % Find & store the points to interpolate with
        x = linspace(-5,5,N); % Get list of N values equally spaced over
           [-5,5]
        test_points_x = [test_points_x, x];

        y = sin(x);
        test_points_y = [test_points_y, y];

        syms t
        % contructing Lagrange polynomials L1, L2,..., Ln
        L = zeros(1,N,'sym');
        for i = 1:N
                L(i) = 1;
                z = x;
                z(i) = [];     % z is an array obtained from array x by
                   obmitting the i'th entry
                for j = 1:N-1
```

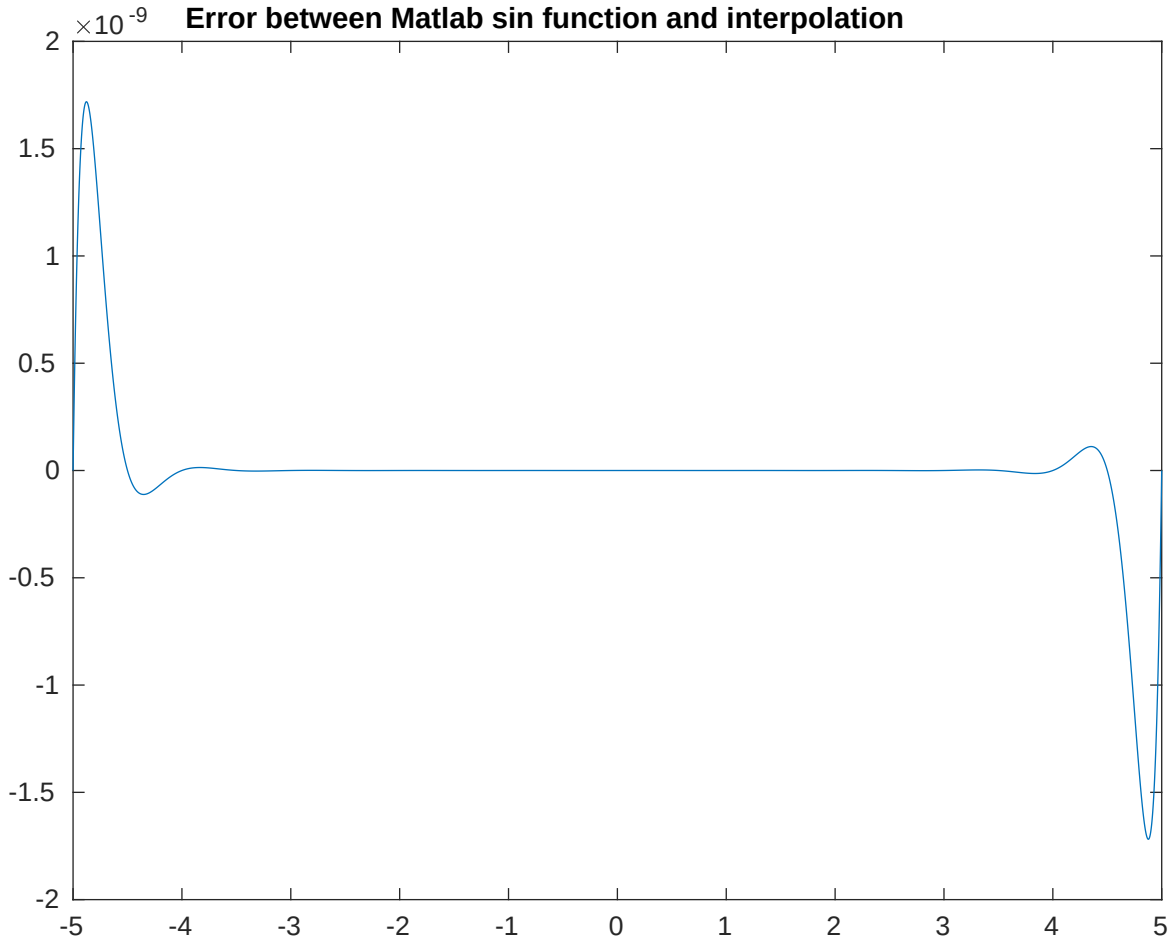
```
                        L(i) = L(i)*(t - z(j))/(x(i) - z(j));
        end
end

% construct P = y1*L1 + y2*L2 + ... + yn*Ln
Leg_Polynomial = L*transpose(y);
%Store the polynomial and a label
P(Nindex) = simplify(Leg_Polynomial);
end

legend_key{end} = "test points";

for Nindex = 1:length(Nlist)
        fplot(P(Nindex),[-5 5])
        hold on
end
title('')
scatter(test_points_x,test_points_y,'ok')
legend(legend_key)
legend('boxoff')
hold off
```

We can save the plot produced with the saveas function.




We can see that $N = 3$ (blue line) does not produce a good interpolation. Once we have 6 data points (red

line), the interpolation captures the general shape of the sin function. The $N = 11$ (yellow) and $N = 21$ (purple) interpolation curves are nearly indistinguishable, except for where $f'$ is relatively small. As we increase $N$,we obtain a better interpolation. The $N = 21$ interpolation approximates $f$ so well that we cannot see the difference easily if we tried adding $f$ to this plot. We can instead plot the error between the two plots.




And so we see that for $N = 21$, the maximum error of the interpolation is $< 2 \times 10^{-9}$ relative to the Matlab sin function. This error is too small to see on the standard (units with order 0.5) graph. This figure can be generated by appending the following code to the above code

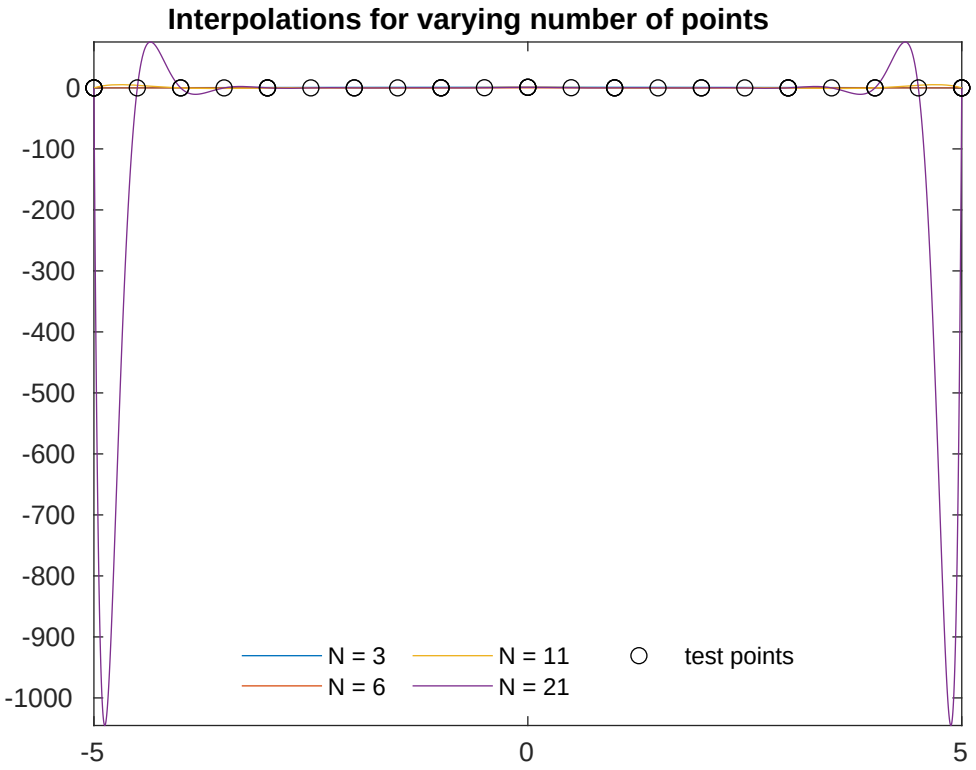
```
t = linspace(-5,5,5000);
yp = sin(t);
pp = subs(P(4));
plot(t, yp-pp)
title('Error between Matlab sin function and interpolation')
```

**b)** We can generalize the code we used in part a) and change the objective function we are interpolating. This following Matlab code does this.

```
Nlist = [3, 6, 11, 21];  % Make a list of candidates
P = sym('p',[1 length(Nlist)]); % A vector of symbolic objects to store
    our polynomials
legend_key = cell(1, length(Nlist)+1);  % To store the function names for
    the legend
```

```
% Create empty vectors to store interpolation points
test_points_x = [];
test_points_y = [];
% it's a best practice to pre-allocate arrays, but the index arithmetic
% can get a bit messy

for Nindex = 1:length(Nlist)  % Loop over the list values
        N = Nlist(Nindex);  % Set N from our list
        legend_key{Nindex} = strjoin({'N =', num2str(N)});
        % Find & store the points to interpolate with
        x = linspace(-5,5,N); % Get list of N values equally spaced over
           [-5,5]
        test_points_x = [test_points_x, x];

        y = objective(x);  % a helper function below
        test_points_y = [test_points_y, y];

        syms t
        % contructing Lagrange polynomials L1, L2,..., Ln
        L = zeros(1,N,'sym');
        for i = 1:N
                L(i) = 1;
                z = x;
                z(i) = [];     % z is an array obtained from array x by
                   obmitting the i'th entry
                for j = 1:N-1
                        L(i) = L(i)*(t - z(j))/(x(i) - z(j));
                end
        end

        % construct P = y1*L1 + y2*L2 + ... + yn*Ln
        Leg_Polynomial = L*transpose(y);
        %Store the polynomial and a label
        P(Nindex) = simplify(Leg_Polynomial);
end

legend_key{end} = "test points";

for Nindex = 1:length(Nlist)
        fplot(P(Nindex),[-5 5])
        hold on
end
title('Interpolations for varying number of points')
scatter(test_points_x,test_points_y,'ok')
legend(legend_key,'Location','south', 'NumColumns',3)
legend('boxoff')
hold off

function out = objective(x)
        out = 1./(1+10.*x.^2);
end
```

This outputs the following plot

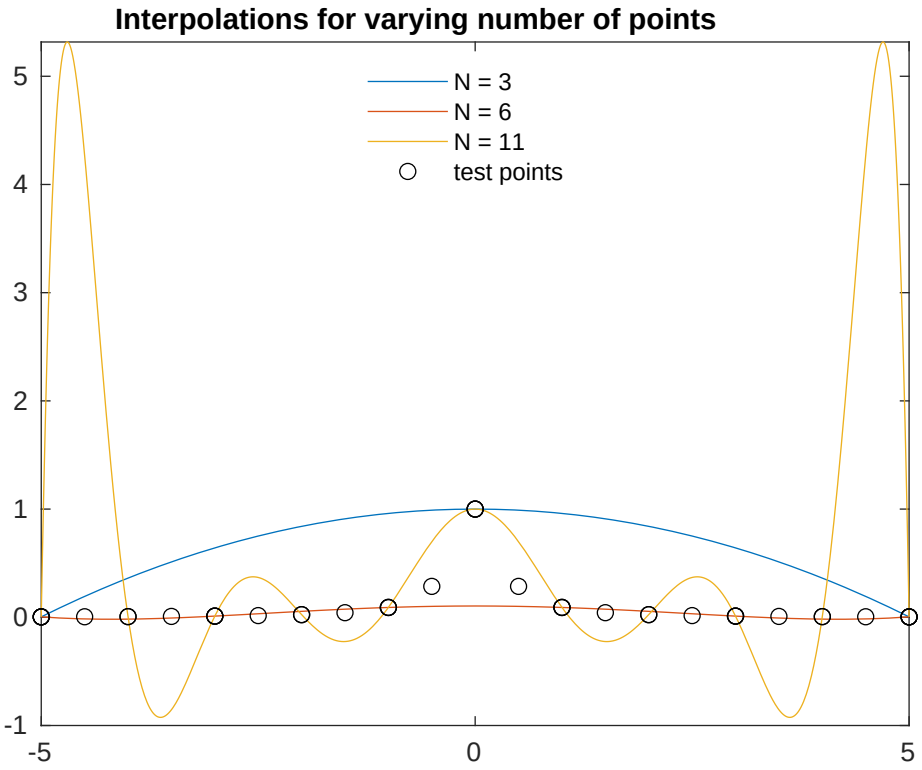



This does not particularly well interpolate the ends of the interval, but does appear to interpolate the inner $[-3, 3]$ rather nicely. We can produce a second plot excluding the $N = 21$ to observe the behavior of the other 3 interpolations.

```
% Find the interpolating polynomial using Lagrange formula

Nlist = [3, 6, 11, 21];  % Make a list of candidates
P = sym('p',[1 length(Nlist)]); % A vector of symbolic objects to store
    our polynomials
legend_key = cell(1, length(Nlist));  % To store the function names for
    the legend

% Create empty vectors to store interpolation points
test_points_x = [];
test_points_y = [];
% it's a best practice to pre-allocate arrays, but the index arithmetic
% can get a bit messy

for Nindex = 1:length(Nlist)  % Loop over the list values
        N = Nlist(Nindex);  % Set N from our list
        legend_key{Nindex} = strjoin({'N =', num2str(N)});
        % Find & store the points to interpolate with
        x = linspace(-5,5,N); % Get list of N values equally spaced over
            [-5,5]
        test_points_x = [test_points_x, x];

        y = objective(x);  % a helper function below
        test_points_y = [test_points_y, y];

        syms t
```

```
        % contructing Lagrange polynomials L1, L2,..., Ln
        L = zeros(1,N,'sym');
        for i = 1:N
                L(i) = 1;
                z = x;
                z(i) = [];      % z is an array obtained from array x by
                   obmitting the i'th entry
                for j = 1:N-1
                        L(i) = L(i)*(t - z(j))/(x(i) - z(j));
                end
        end

        % construct P = y1*L1 + y2*L2 + ... + yn*Ln
        Leg_Polynomial = L*transpose(y);
        %Store the polynomial and a label
        P(Nindex) = simplify(Leg_Polynomial);
end

legend_key{end} = "test points";

for Nindex = 1:length(Nlist)-1
        fplot(P(Nindex),[-5 5])
        hold on
end
title('Interpolations for varying number of points')
scatter(test_points_x,test_points_y,'ok')
legend(legend_key,'Location','north')
legend('boxoff')
hold off

function out = objective(x)
        out = 1./(1+10.*x.^2);
end
```

which produces the following plot (next page).




And we see that $N = 3$ poorly interpolates the data (test points), but as we increase the number of interpolation points ($N = 11, 21$), the interpolations increase in maximum error. Interestingly, when we choose $N = 6$, the interpolation is much more reasonable on $[-5, 5]$, even though it fails to capture numerically the maxima at $x = 0$. Large $N$ does not necessarily imply a better interpolation.

# Problem 2.

Use Newton's formula to find a polynomial of degree $\leq$ 3 that fits the following points $(2, 1), (1, 0), (3, -1), (0, 2)$. Convert the polynomial into standard form.

## Solution

We first find the coefficients. We can construct a table and perform the recursion to obtain

| | | | |
|---|---|---|---|
| 0 | 1 | 2 | 3 |
| 1 | 1 | -1.5 | -1 |
| 0 | -0.5 | 0.5 | 0 |
| -1 | -1 | 0 | 0 |
| 2 | 0 | 0 | 0 |

and read off the first row to obtain the sequence of coefficients $(c_n)$ is $1, 1, -1.5, -1$. Next we can construct the polynomial basis.

$$P_1(x) = x - 2$$
$$P_2(x) = (x - 2)(x - 0)$$
$$P_3(x) = (x - 2)(x - 0)(x - 3)$$

So

$$P(x) = c_0 + c_1P_1(x) + c_2P_2(x) + c_3P_3(x)$$
$$P(x) = 1 + (1)(x-2) - \frac{3}{2}(x)(x-2) - (1)(x-2)(x)(x-3)$$
$$P(x) = 1 + x - 2 - \frac{3}{2}(x^2) - \frac{9}{2}x - 3 - x^3 + 6x^2 - 11x + 6$$
$$P(x) = -x^3 + \frac{9}{2}x^2 - \frac{11}{2}x + 2.$$

This is the standard form for our polynomial.

# Problem 3.

Reorder the points in Problem 2 as follows: $(3,-1)$, $(1,0)$, $(0,2)$, $(2,1)$. Find the Newton's formula corresponding to these data points (in this order). Do you get the same polynomial as in problem 1? Explain your observation.

## Solution

We repeat the same procedure as in problem 2.

| 0 | 1 | 2 | 3 |
|---|---|---|---|
| -1 | -1/2 | 1/2 | -1 |
| 0 | -2 | 1.5 | 0 |
| 2 | 1/2 | 0 | 0 |
| 1 | 0 | 0 | 0 |

and read off the first row to obtain the sequence of coefficients $(c_n)$ is $-1, 1/2, 1/2, -1$. Next we can construct the polynomial basis.

$$P_1(x) = x - 3$$
$$P_2(x) = (x-3)(x-1)$$
$$P_3(x) = (x-3)(x-1)(x-0)$$

So

$$P(x) = c_0 + c_1P_1(x) + c_2P_2(x) + c_3P_3(x)$$
$$P(x) = -1 + -\frac{1}{2}(x-3) + \frac{1}{2}(x-3)(x-2) - (1)(x-3)(x-2)(x)$$
$$P(x) = -\left(x^3\right) + \frac{9}{2}x^2 - \frac{11}{2}x + 2$$

which gives the normal (common) form for our polynomial.

This is the same polynomial computed in the previous part. This is because the interpolating polynomial is unique. We can reorder the points and compute a different set of basis polynomials, but then the coefficients used to construct the interpolation change accordingly, so we can always find this polynomial in any basis that could be generated from these data points.

# Problem 4.

Write a function in Matlab that does the following:

- Input:
  - a function $f$
  - an array $x$, $x = (x_1, x_2, \ldots, x_n)$.
- Output: The divided difference $f[x_1, x_2, \ldots, x_n]$.

Test your function with $f(t) = \frac{1}{1+t^2}$ and $x = (1, 2, 3, 4)$.

## Solution

We can utilize the recursive definition of the divided differences formula and construct a table.

```
Xpts = [1,2,3,4];
f = @(x) 1./(1+x.^2);
lastval = last_divdif(Xpts, f);
disp(lastval)

%To check that our points are right
fplot(f,[0,4.5])
hold on
scatter(Xpts, f(Xpts),'filled')
hold off

function lastval = last_divdif(Xpts, f)
coef_array = divdif(Xpts, f(Xpts));
disp(coef_array)
lastval = coef_array(1,end);
end

function coef_array = divdif(Xpts,Ypts)
        % Xpts and Ypts are data vectors of the same length
        % Xpts = [x1, x2, x3, ... xN]
        % Ypts = [y1, y2, y3, ... yN]
        % coef_array is a table of intermediate divided difference
           coefficients
        datalength = length(Xpts);
        coef_array = zeros(datalength);
        coef_array(:,1) = Ypts';  % Write the data values to the first
           column
        for col = 2:datalength
                for row = 1 : (datalength - col + 1)
                        %and now our magic step
                        coef_array(row, col) = (coef_array(row+1, col-1) -
                            coef_array(row, col - 1) )/(Xpts(row + col -1)
                            - Xpts(row));
                end
        end

end
```

The coefficient $c_3$ is $-0.023529411764706$.

# Problem 5.

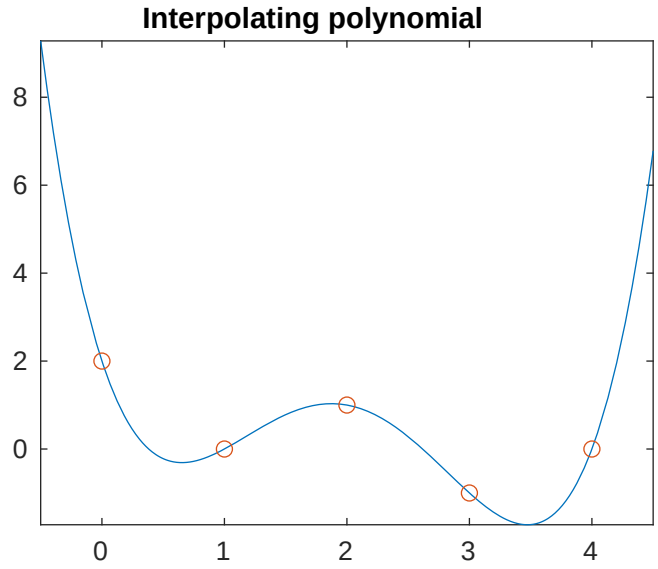
## Solution

We slightly modify the function to not need the wrapper function that computes $(y_1, y_2, \ldots, y_n)$ (as we are provided points).

```
% Read in our data
xpts = [2,1, 3,0,4];
ypts = [1,0,-1,2,0];

data_length = length(xpts);
% Find div-dif coefficients
coef_array = divdif(xpts, ypts);
coef = coef_array(1,:);

% Find the basis polynomials
basis = ones(1,data_length, 'sym');  % To store our basis polynomials
syms t  % Our symbolic variable
for basis_index = 2:length(basis)  % Loop over each basis
        for x_index = 1:basis_index-1  % Loop over the first basis_index
           data points we want
                basis(basis_index) = basis(basis_index) * (t - xpts(
                   x_index));
        end
end

% Construct the interpolating polynomial
P = basis*coef';
P = simplify(P)  % Allow output to write P to console
fplot(P, [-0.5,4.5])
hold on
scatter(xpts, ypts)
title('Interpolating polynomial')
hold off

%We built a recusive helper function that will make short work of the
   Newton's
%Divided Differences coefficients.
function coef_array = divdif(Xpts,Ypts)
        % Xpts and Ypts are data vectors of the same length
        % Xpts = [x1, x2, x3, ... xN]
        % Ypts = [y1, y2, y3, ... yN]
        datalength = length(Xpts);
        coef_array = zeros(datalength);
        coef_array(:,1) = Ypts';  % Write the data values to the first
           column
        for col = 2:datalength
                for row = 1 : (datalength - col + 1)
                        %and now our magic step
                        coef_array(row, col) = (coef_array(row+1, col-1) -
                             coef_array(row, col - 1) )/(Xpts(row + col -1)
                             - Xpts(row));
                end
        end
```

```
end
```

This produces the following plot.




The polynomial that interpolates the data set is

$$P(x) = \frac{1}{2}x^4 - 4x^3 + 10x^2 - \frac{17}{2}x + 2$$