Problem 1.

Let $f(x) = (1+x)^{-1}$. For evenly spaced sample points $0 = x_1 < x_2 < \cdots < x_n = 2$, let P_n be the polynomial that interpolates (x_n). Find n such that

$$|f(x) - P_n(x)| \leq 10^{-4} \quad \forall x \in [0, 2]$$

Solution

We can bound the error by using the fact that the interpolating points (x_n) are uniformly spaced.

$$|f(x) - P_n(x)| \leq \frac{1}{n}\left(\frac{2}{n-1}\right)^n \max_{x\in[a,b]} |f^{(n)}(x)|$$

As $|f^{(n)}(x)| = n!(1+x)^{-n-1}$, the maximum occurs at $x = 0$ (as f is continuous on a closed and bounded interval, f attains it's maximum), then

$$|f(x) - P_n(x)| \leq \frac{1}{n}\left(\frac{2^n}{n-1}\right)^n \frac{n!}{(1+0)^{n+1}} = \left(\frac{2}{n-1}\right)^n (n-1)! < 10^{-4}$$

With a calculator, we can solve for n and find that $n \geq 31$ is sufficient to satisfy the error bound.

Problem 2.

2. Given a function f on some interval, say $[-1, 1]$, and an integer $n > 1$, we are interested in the question: what set of sample points $\{x_1, x_2, ..., x_n\}$ on $[-1, 1]$ should we choose so that the corresponding interpolation polynomial Pn can best approximate function f? Note that the number of sample points n is fixed. We are testing different choices of sample points.

To investigate this question, let us consider an example $f(x) = \frac{1}{1+10x^2}$ and let $n = 11$, partitioning $[-1, 1]$ into 10 partitions). Consider two ways of sampling:

- Evenly spaced $-1 = x_1 < x_2 < \ldots < x_n = 1$
- Unevenly spaced, $z_k = \cos\left(\frac{2k-1}{2n}\pi\right)$, for $k = 1, 2, \ldots, n$.

a) Use the Plot command to sketch each set of sample points on the interval $[-1, 1]$.

b) Let P_n be the polynomial that interpolates the set of data points $(x_1, f(x_1)), (x_2, f(x_2)), \ldots, (x_n, f(x_n))$. Plot P_n and f on the same graph.

c) Let Q_n be the polynomial that interpolates the set of data points $(z_1, f(z_1)), (z_2, f(z_2)), \ldots, (z_n, f(z_n))$. Plot Q_n and f on the same graph.

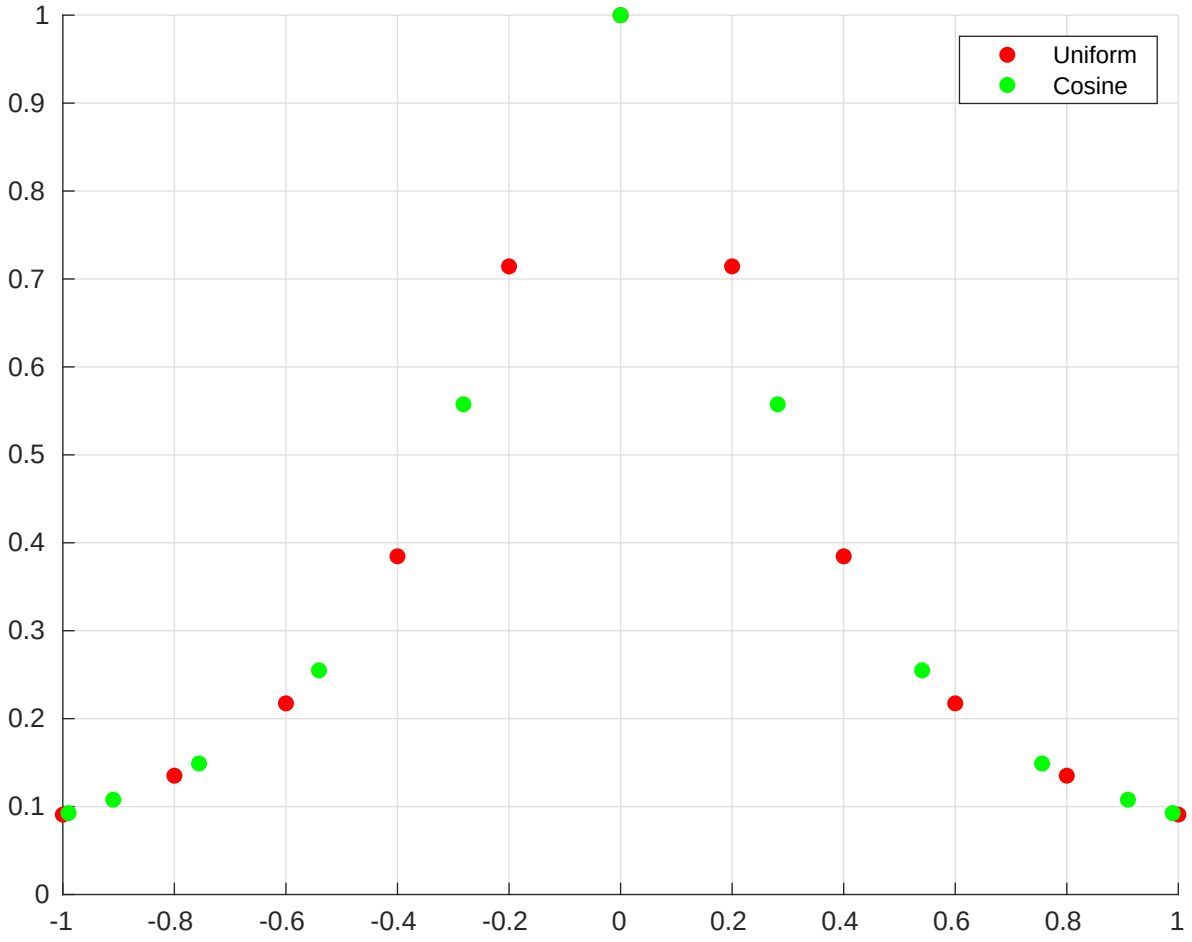
d) Based on the graphs, is one way of sampling significantly better than the other? Give a rough explanation for your observation?

e) Repeat parts $a - d$ for the objective function $f(x) = \cos(x)$.

Solution

a) Some Matlab code:

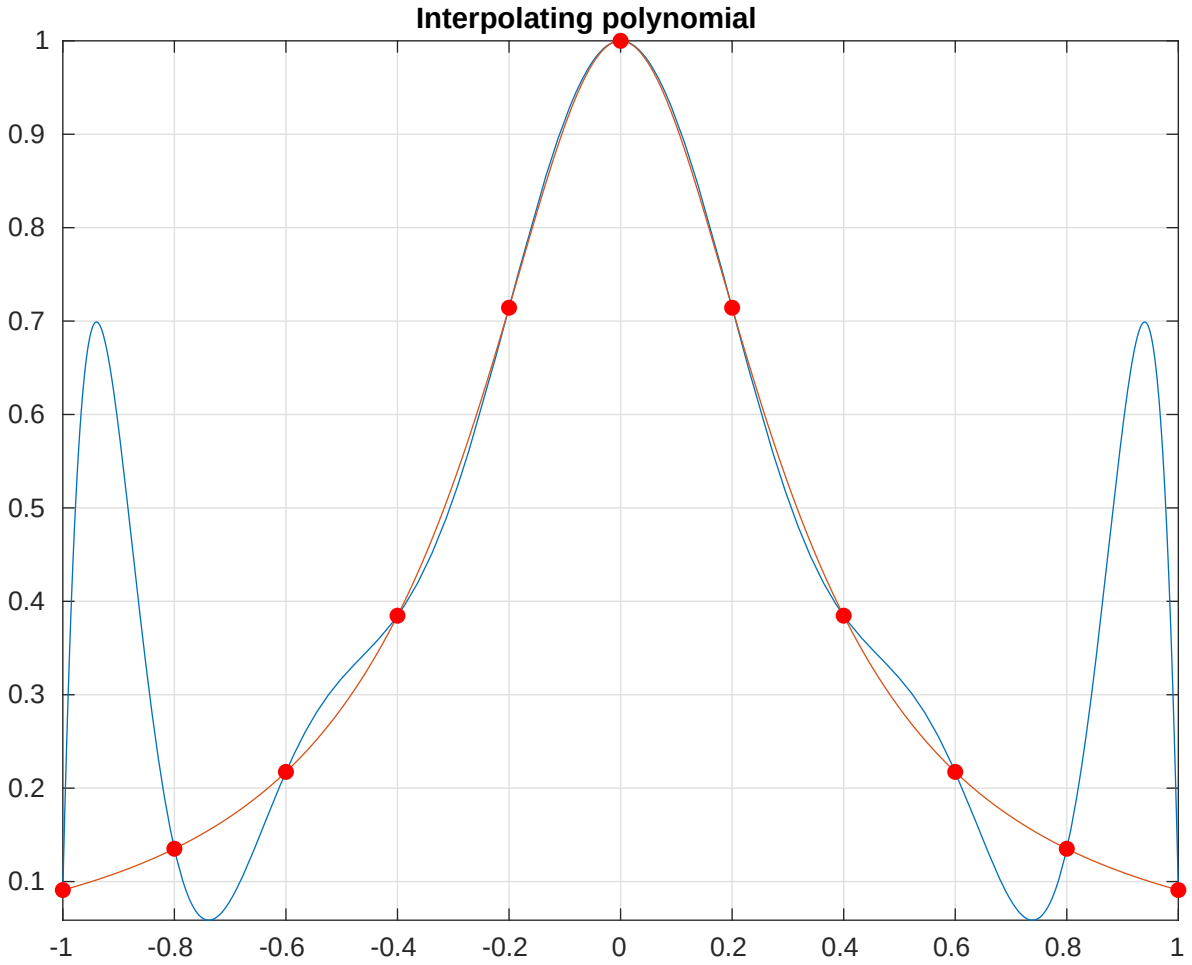
```
n = 11;
xpts = linspace(-1,1,n);
zpts = 1:1:n;
zpts = cos((2.*zpts-1)./(2*n)*pi);

yxpts = objective(xpts);
yzpts = objective(zpts);

scatter(xpts, yxpts, 'filled', 'r')
grid on
hold on
scatter(zpts, yzpts, 'filled', 'g')
legend('Uniform','Cosine')
hold off

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



b) Some Matlab code:

```
% Read in our data
n = 11;
xpts = linspace(-1,1,n);
tpts = linspace(-1,1,500);
yxpts = objective(xpts);
ytpts = objective(tpts);

syms interP
interP = make_interpolating_polynomial(xpts, yxpts);
fplot(interP, [-1,1])
grid on
hold on
plot(tpts, ytpts)
scatter(xpts, yxpts, 'filled', 'r')
title('Interpolating polynomial')
hold off

% This function is recovered from HW6#5. Lagrange's method is also
% acceptable for this problem, using the starter code on the course
   website
function poly = make_interpolating_polynomial(xpts, ypts)
    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';
    poly = simplify(P);
end


%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

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

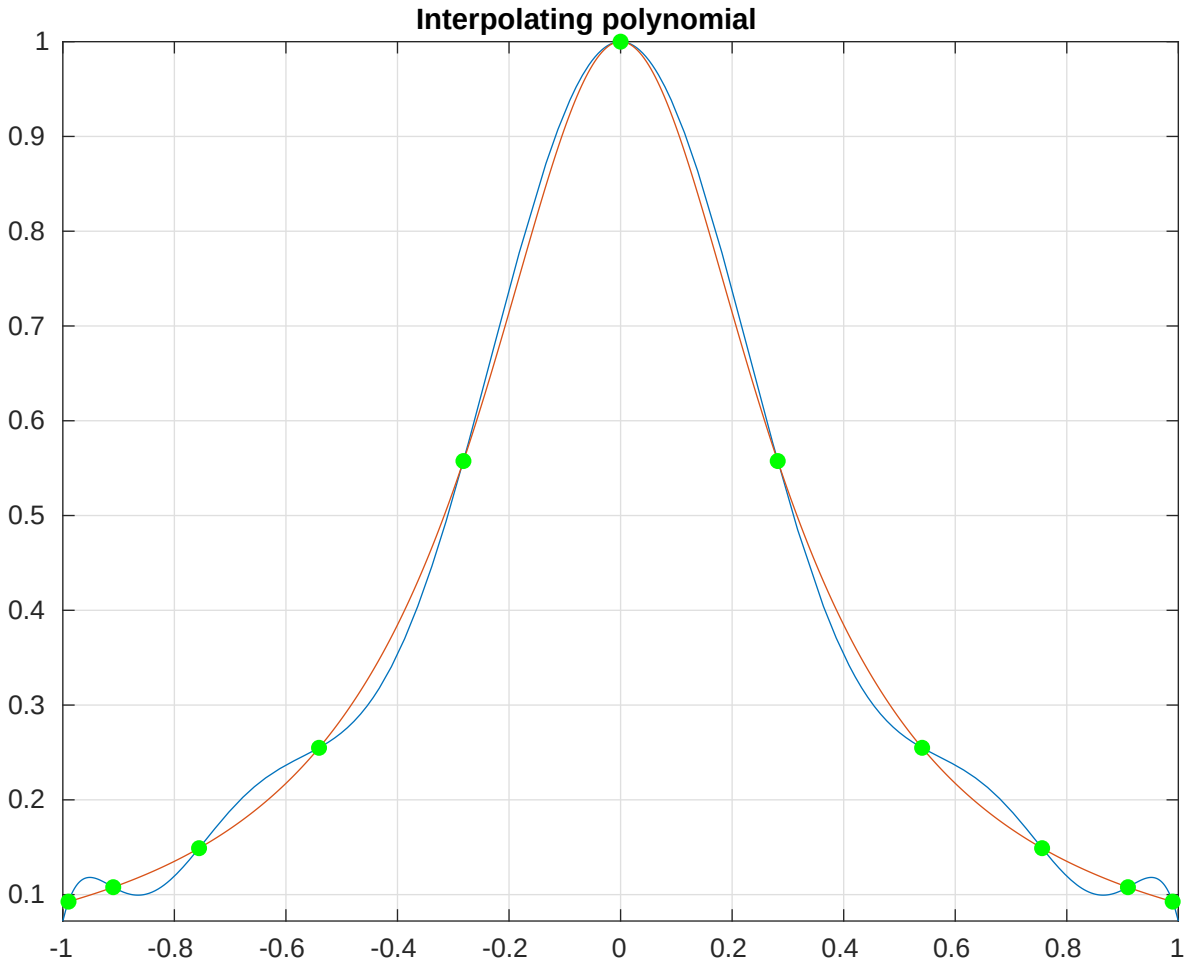


c) Some Matlab code:

```
% Read in our data
n = 11;
zpts = 1:1:n;
zpts = cos((2.*zpts-1)./(2*n)*pi);
tpts = linspace(-1,1,500);
ytpts = objective(tpts);
yzpts = objective(zpts);

syms interQ
```

```
interQ = make_interpolating_polynomial(zpts, yzpts);
fplot(interQ, [-1,1])
grid on
hold on
plot(tpts, ytpts)
scatter(zpts, yzpts, 'filled', 'g')

title('Interpolating polynomial')
hold off

% This function is recovered from HW6#5. Lagrange's method is also
% acceptable for this problem, using the starter code on the course
   website
function poly = make_interpolating_polynomial(xpts, ypts)
    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';
    poly = simplify(P);
end


%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
```

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



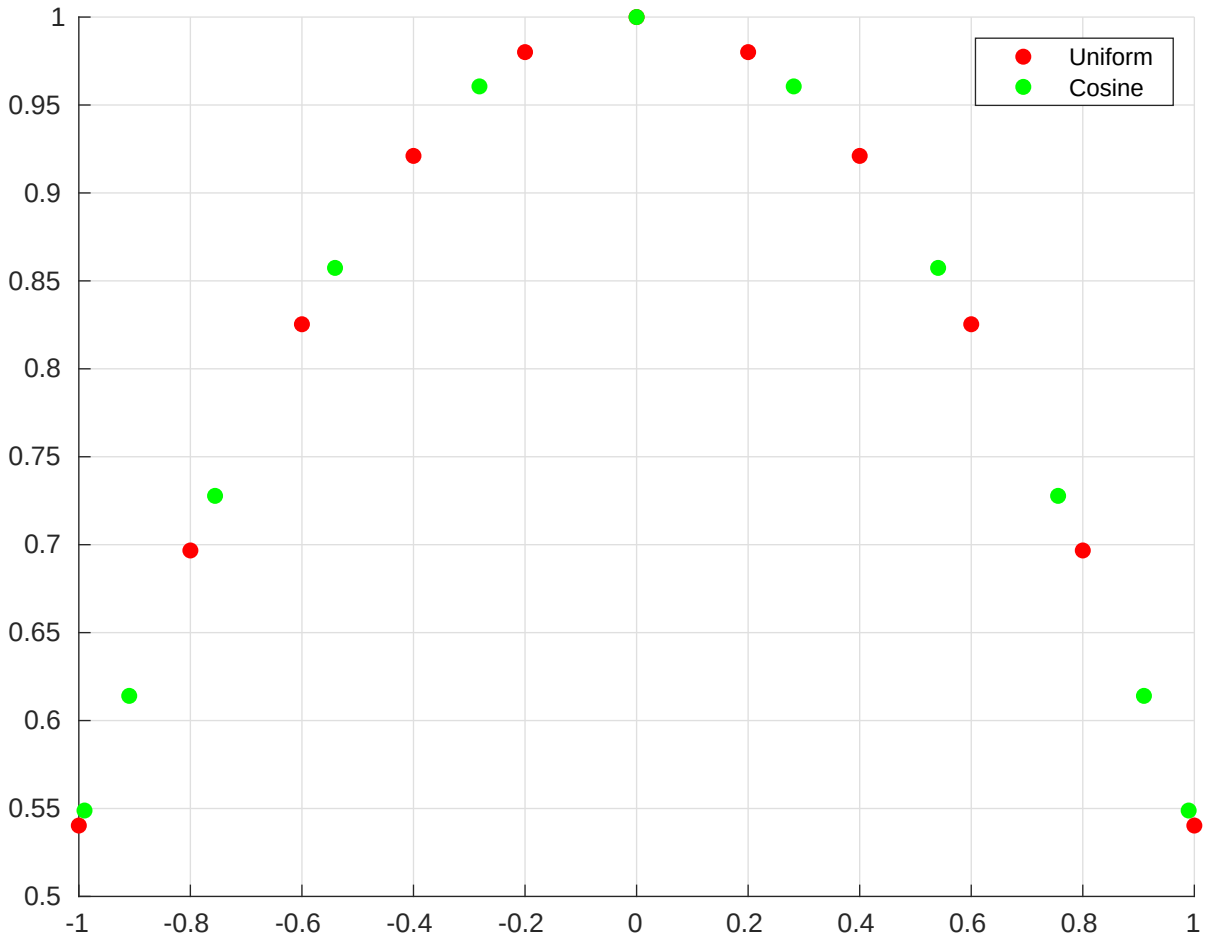
d) The points z_k produces a significantly better interpolation polynomial than the evenly spaced x_k. P_n gives a slightly better approximation f than Q_n near the center of the interval, but the error near the ends of the interval of P_n is worse than Q_n. Decreasing the distance between points on the endpoints helps resist the Runge effect as the maximum error

e) We change the objective function to $f(x) = \cos(x)$. The code is repeated for completeness.

Repeat of part a) Some Matlab code:

```
n = 11;
xpts = linspace(-1,1,n);
zpts = 1:1:n;
zpts = cos((2.*zpts-1)./(2*n)*pi);

yxpts = objective(xpts);
yzpts = objective(zpts);

scatter(xpts, yxpts, 'filled', 'r')
```

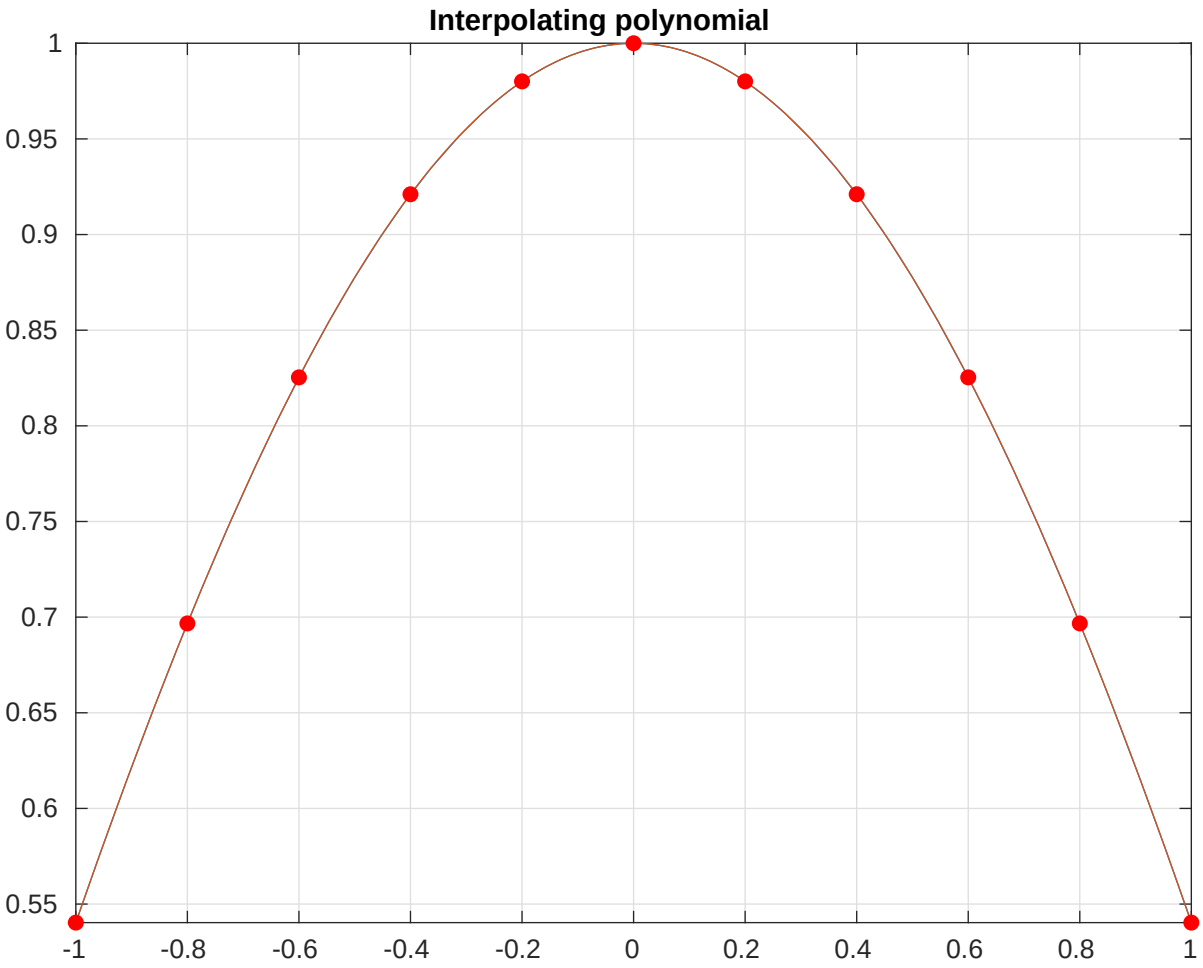
```
grid on
hold on
scatter(zpts, yzpts, 'filled', 'g')
legend('Uniform','Cosine')
hold off

function out = objective(in)
    out = cos(in);
end
```



Repeat of part b) Some Matlab code:

```
% Read in our data
n = 11;
xpts = linspace(-1,1,n);
tpts = linspace(-1,1,500);
yxpts = objective(xpts);
ytpts = objective(tpts);

syms interP
interP = make_interpolating_polynomial(xpts, yxpts);
fplot(interP, [-1,1])
grid on
hold on
```

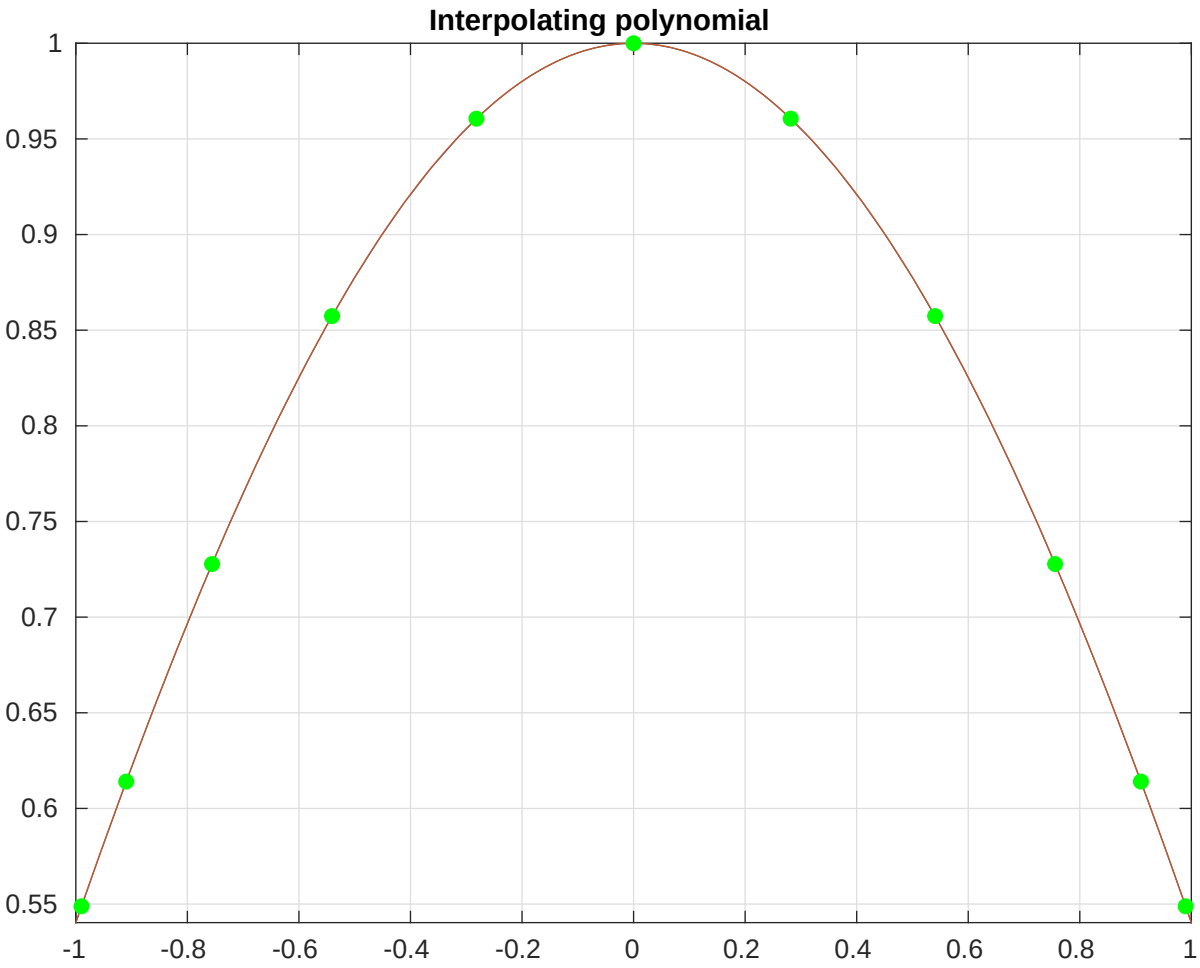
```
plot(tpts, ytpts)
scatter(xpts, yxpts, 'filled', 'r')
title('Interpolating polynomial')
hold off

% This function is recovered from HW6#5. Lagrange's method is also
% acceptable for this problem, using the starter code on the course
   website
function poly = make_interpolating_polynomial(xpts, ypts)
    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';
    poly = simplify(P);
end


%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

function out = objective(in)
    out = cos(in);
end
```



Repeat of part c) Some Matlab code:

```
% Read in our data
n = 11;
zpts = 1:1:n;
zpts = cos((2.*zpts-1)./(2*n)*pi);
tpts = linspace(-1,1,500);
ytpts = objective(tpts);
yzpts = objective(zpts);

syms interQ
interQ = make_interpolating_polynomial(zpts, yzpts);
fplot(interQ, [-1,1])
grid on
hold on
plot(tpts, ytpts)
scatter(zpts, yzpts, 'filled', 'g')

title('Interpolating polynomial')
hold off

% This function is recovered from HW6#5. Lagrange's method is also
% acceptable for this problem, using the starter code on the course
```

```
   website
function poly = make_interpolating_polynomial(xpts, ypts)
    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';
    poly = simplify(P);
end


%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

function out = objective(in)
    out = cos(in);
end
```



Repeat of part d) Neither polynomial is particularly better or worse than the other.

Problem 3.

We now test how well quadratic spline interpolation can approximate a function. Consider $f(x) = \frac{1}{1+10x^2}$ on the interval $[-1, 1]$. Choose $n = 11$ evenly spaced sample points $-1 = x_1 < x_2 < \ldots < x_n = 1$. Set $y_j = f(x_j)$. On each subinterval, $[x_j, x_{j+1}]$, the function f is approximated by a quadratic polynomial $s_j(x)$ such that the slope of the approximation curve varies smoothly across each sample point. Denote $M_j = s_j'(x_j)$ and $M_{j+1} = s_j'(x_{j+1})$.

a) For $j = 1, \ldots, n-1$, use the fact that $s_j(x_j) = y_j$ to write the formula for $s_j(x)$ in terms of y_j, M_j, and M_{j+1}.

b) For $j = 1, \ldots, n-1$, write an expression that M_j and M_{j+1} has to satisfy such that $s_j(x_{j+1}) = y_{j+1}$.

c) In part (b), we know that each $j = 1, \ldots, n-1$, yields an equation which $M_1, M_2, \ldots, M_n$. Thus there are $n-1$ equations to solve for n unknowns $M_1, M_2, \ldots, M_n$.

d) Use Matlab to draw the spline interpolation curve (defined as the concatenation of the quadratic curves $s_1, s_2, \ldots, s_n$). Plot f and the interpolation P_n from problem 2 on the same plot. Which is the better approximation of f (spline or polynomial)?

Solution

a) From the notes, we can set

$$s_j(x) = \frac{M_j - M_{j+1}}{2(x_j - x_{j+1})}(x^2 - x_j^2) + \frac{x_j M_{j+1} - x_{j+1} M_j}{x_j - x_{j+1}}(x - x_j) + y_j$$

where $x_j = \frac{2j}{n} - 1$, simplified for brevity.

b) Computing the values M_j is done iteratively, so we may assume that if $j > 0$, we can assume M_j is known (as M_0 is fixed or assumed to be a known value). Then

$$M_{j+1} = \frac{2(y_{j+1} - y_j)}{x_{j+1} - x_j} - M_j$$

c) Set $M_0 = 0$ as indicated. We can use the recursive algorithm described above, pre-computing x_j and $f(x)j$.

```
n = 11;  % n sample points
x = linspace(-1,1,n);
f = @(x) (1 + 10.*x.^2).^(-1);
y = f(x);
M = zeros(1,n);
% We don't need to define M(0) as 0 again

for ii=2:n  % Generate the coefficients M_j
    M(ii) = 2 .* (y(ii) - y(ii-1))./(x(ii) - x(ii-1)) - M(ii-1);
end

disp(M)
```

This produces the sequence of values of M_j as

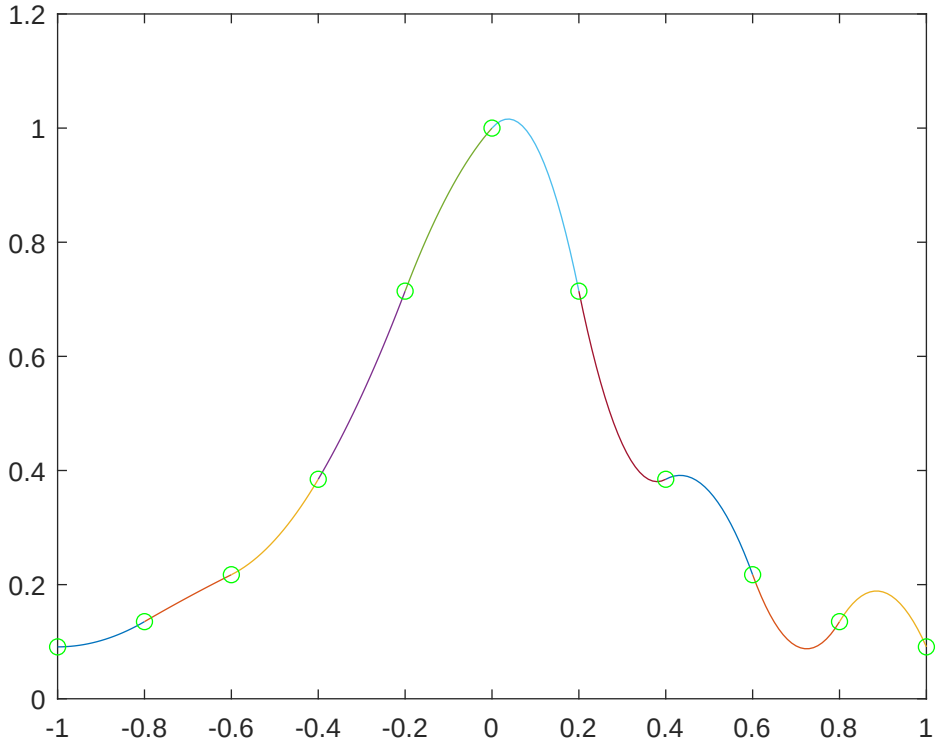
$$0 \quad 0.44226 \quad 0.3803 \quad 1.2919 \quad 2.0048 \quad 0.85238 \quad -3.7095 \quad 0.41282 \quad -2.0851 \quad 1.2625 \quad -1.7048$$

d) We start with the prior algorithm for computing values of M, then build the splines

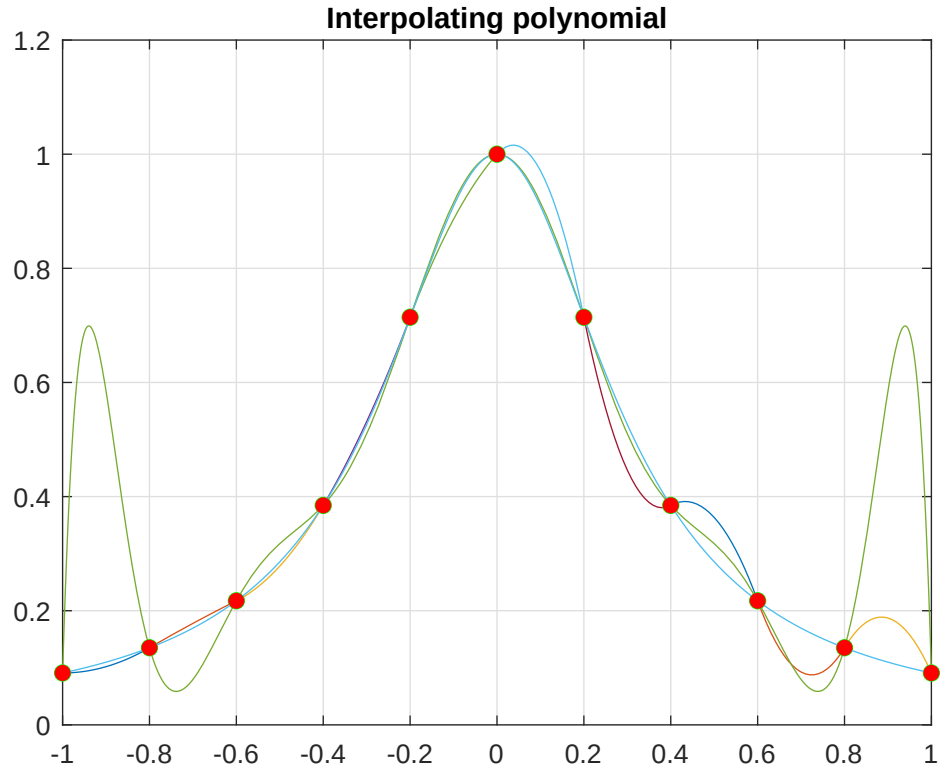
```
%A script to compute and plot quadratic spline approximations for a
   function.

% Precompute a few terms
n = 11;  % n sample points
x = linspace(-1,1,n);
f = @(x) (1 + 10.*x.^2).^(-1);
y = f(x);
M = zeros(1,n);
% We don't need to define M(0) as 0 again

for ii=2:n
    % Generate the coefficients
    M(ii) = (2 .* (y(ii) - y(ii-1))./(x(ii) - x(ii-1)) - M(ii-1);
end

for ii = 1:n-1  % build each spline locally
    % setup local variables
    x_local = x(ii):0.01:x(ii+1);
```

```
    % Build the polynomial
    a_local = ( M(ii) - M(ii+1) ) ./ (2.*(x(ii) - x(ii+1) ) );
    b_local = ( (x(ii) .* M(ii+1)) - (x(ii+1) .* M(ii)))./(x(ii) - x(ii+1)
        );

    y_local = a_local .* (x_local .^2 - x(ii)^2) + b_local .* (x_local-x(
        ii))...
    + y(ii);
    % Now for the plot
    plot(x_local, y_local)
    hold on

end
scatter(x, y, 'g')  % Add the interpolating points as a sanity check
hold off
saveas(gcf, 'MTH351_HW7_fig1', 'epsc')  % Save the figure to an eps file
```

This produces the image



To overlay the plots, specify `hold off` and then run your script or call your function that generates the interpolating polynomial. When we overlay the polynomial interpolation, we obtain



The blue line is f, the green line is the interpolating polynomial, and the patchwork of differently colored lines is the quadratic spline. It appears that the spline is overall a better approximation, as the maximum error of the spline is less than the maximum error of the interpolating polynomial. However, if we are primarily interested in the behavior near 0, the interpolating polynomial is a better approximation, as it better preserves the curvature of the function f. You can see this specifically by examining the slopes of f, P, and s at 0 (the slope of f and P is 0, while s' is not 0 as 0 does not maximize s).