Problem 1.

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

To investigate this question, let us consider an example $f(x) = \frac{1}{1+10x^2}$ and $N = 11$. Consider two different ways of sampling:

- Evenly spaced, $-1 = x_1 < x_2 < x_3 \ldots < x_n = 1$,
- 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.

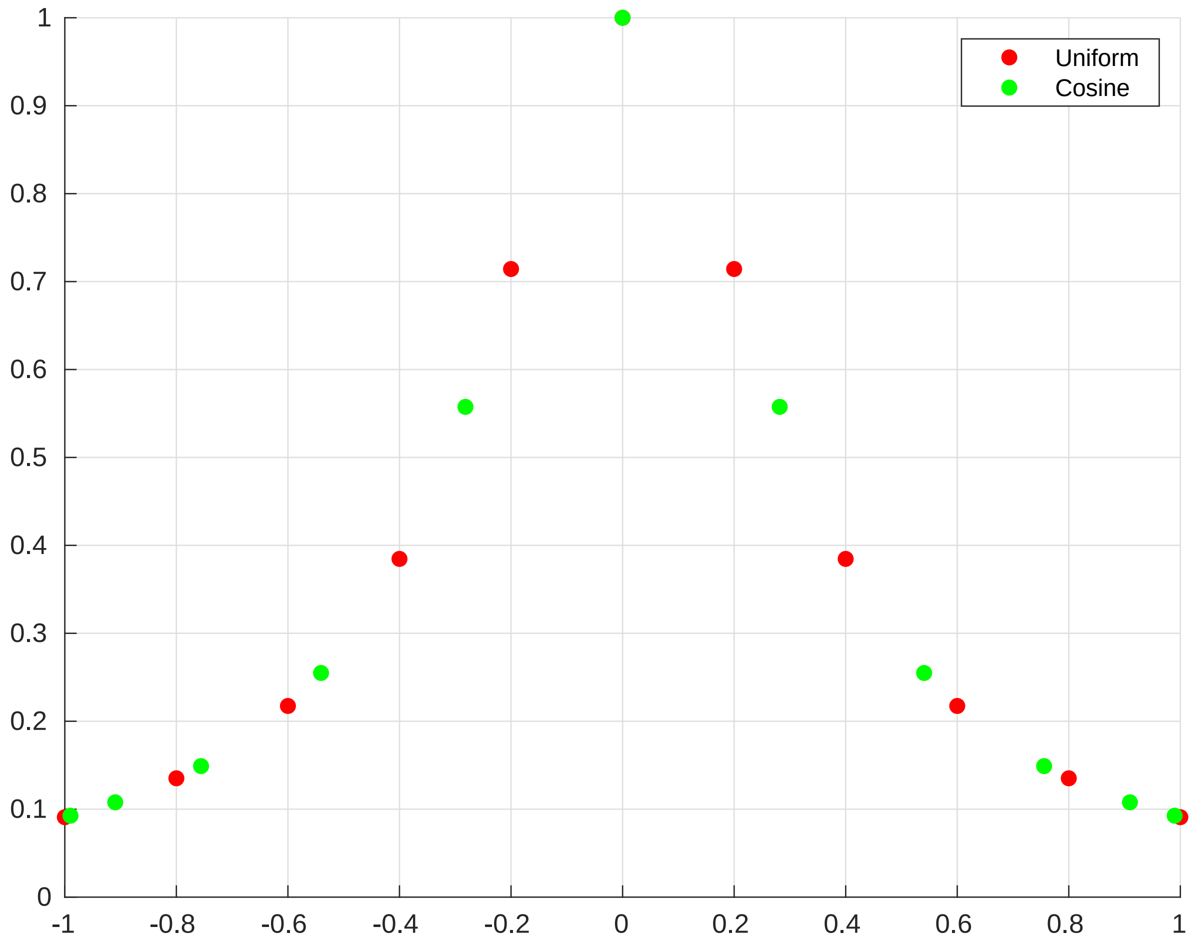
d) Based on the graphs, is one way of sampling significantly better than the other? Give a rough explanation for your observation?

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

Solution

a) Some Matlab code:

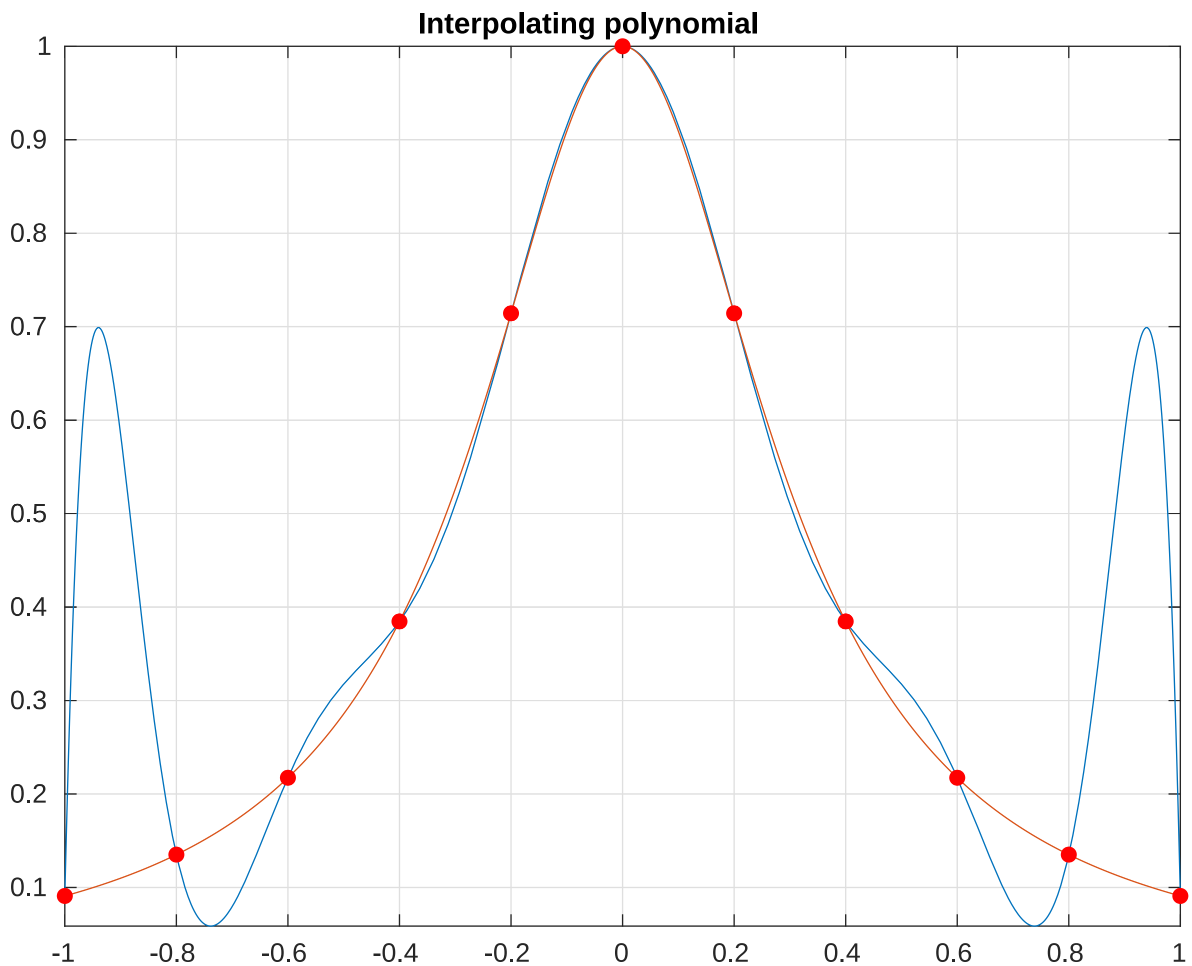
```
n = 11;
xpts = linspace(-1,1,n);
zpts = 1:1:n;
zpts = cos((2.*zpts-1)./(2*n)*pi);

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);
```

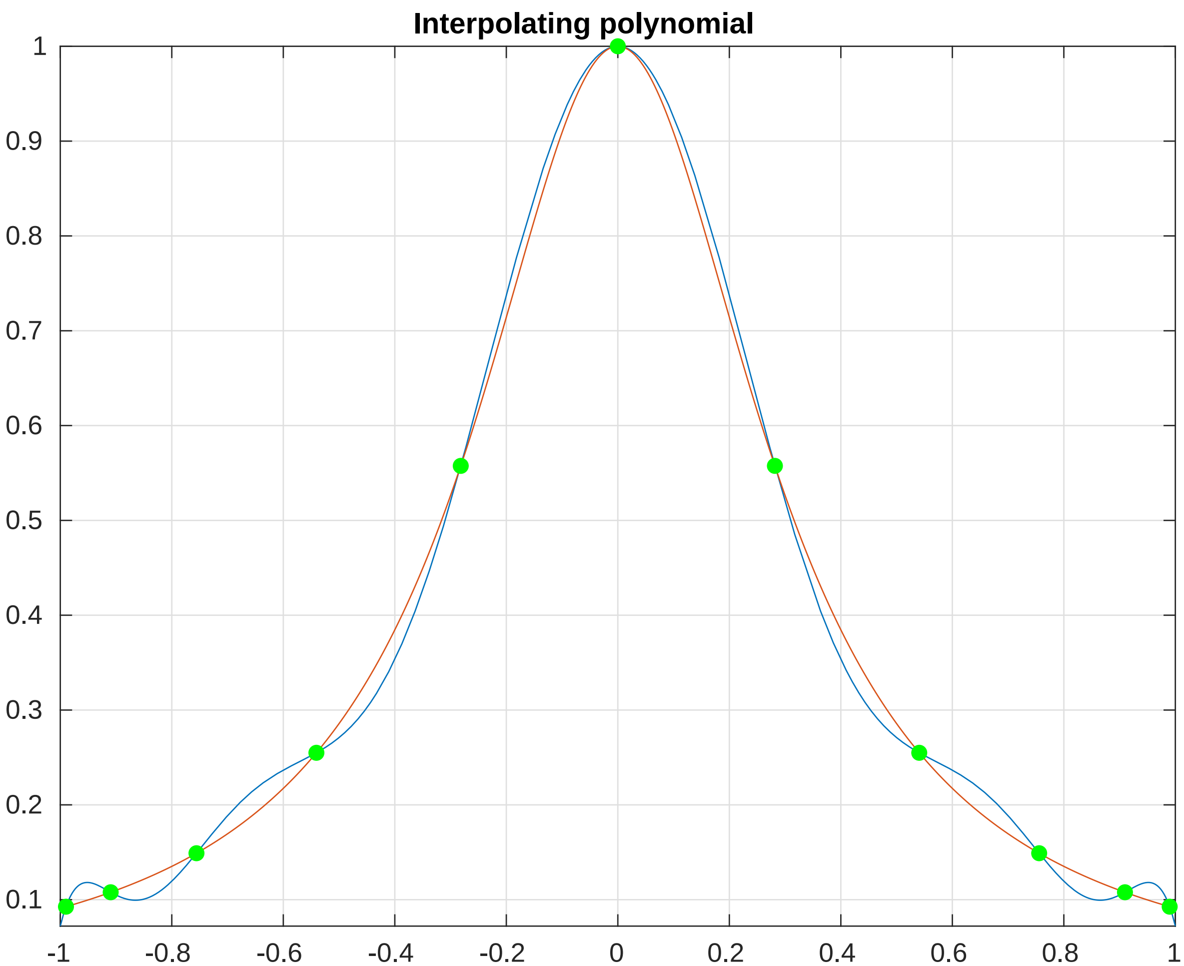
```
    % Find div-dif coefficients
    coef_array = divdif(xpts, ypts);
    coef = coef_array(1,:);

    % 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. The reason is two-fold. First, the points z_k are more crowded near the endpoints and sparser near the middle. Contrary to the uniform sample x_k, the z_k's near the endpoints are "closer" to the rest of $z_1, \ldots z_n$. Thus, when x is close to -1 or 1, the product $|x - z_1| \ldots |x - z_n|$ is smaller than $|x - x_1| \ldots |x - x_n|$. Secondly, as shown in class, the n'th derivative of $1/(x^2+1)$ grows rapidly with respect to n. In fact, one can show that it grows at order $(1/r)^n n!$ (where r is the distant from x to the closer endpoint) although the proof is more involved. Thus, the product

$$|x - x_1|...|x - x_n| \frac{1}{n!} \max_{[-1,1]} \left| \frac{d^n}{dt^n} \left(\frac{1}{1+t^2} \right) \right| \sim h^n (n-1)! \frac{1}{n!} \frac{n!}{r^n} \sim \frac{h^n}{r^n}(n-1)! \sim \left(\frac{2/r}{n} \right)^n (n-1)! \qquad (1)$$

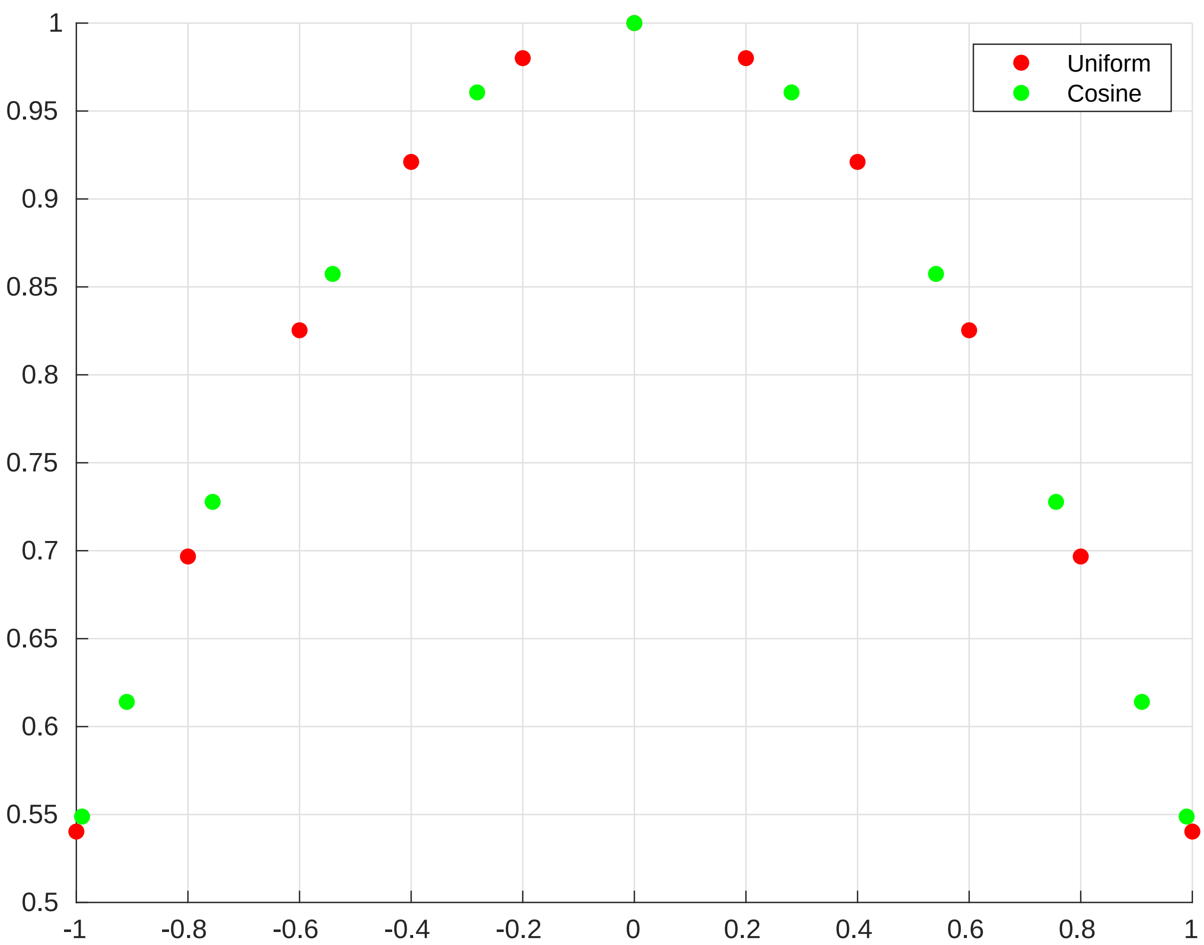
is large when x is near ± 1. It in fact goes to infinity as $n \to \infty$ if $r < 2/e$.

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
```

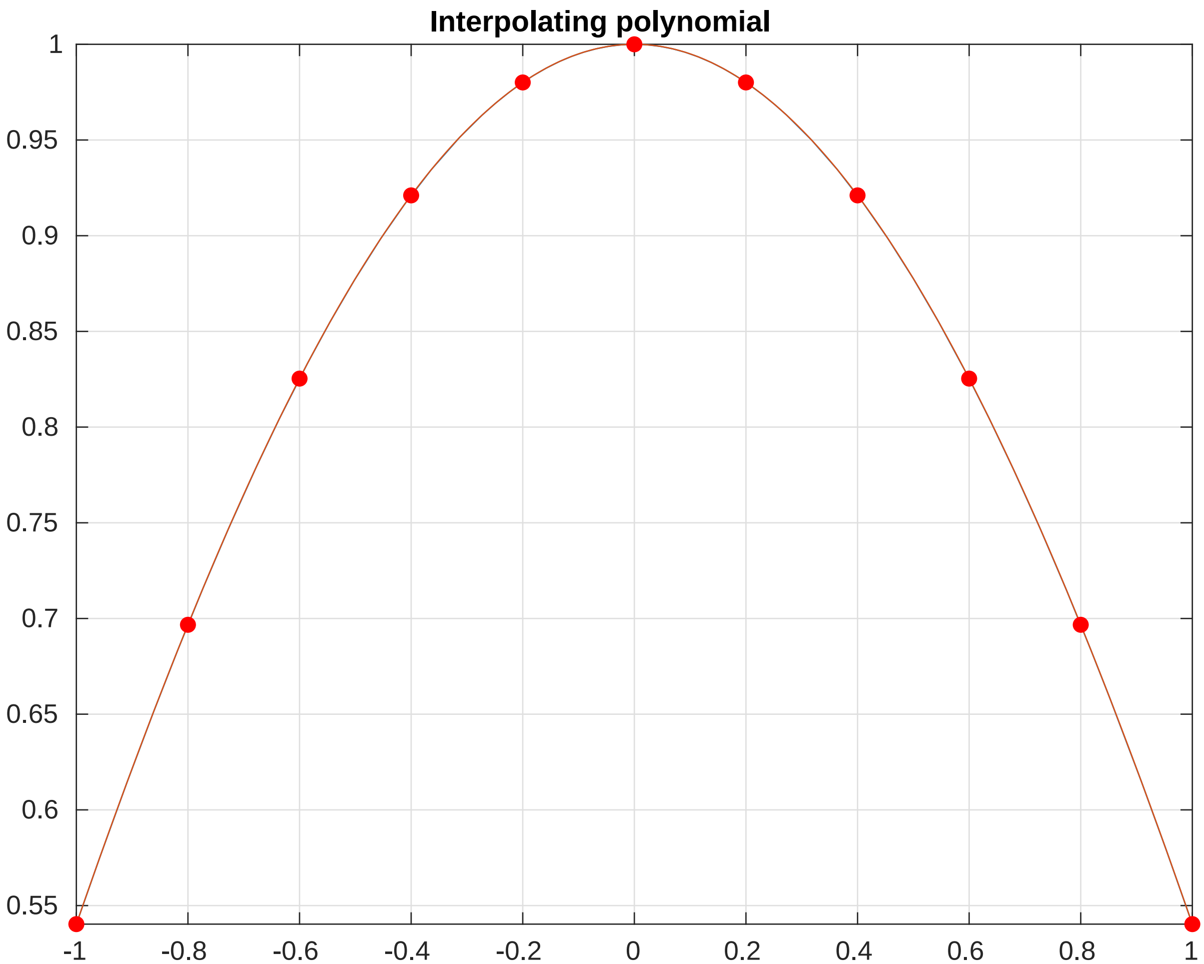


Repeat of part b) Some Matlab code:

```
% 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
```

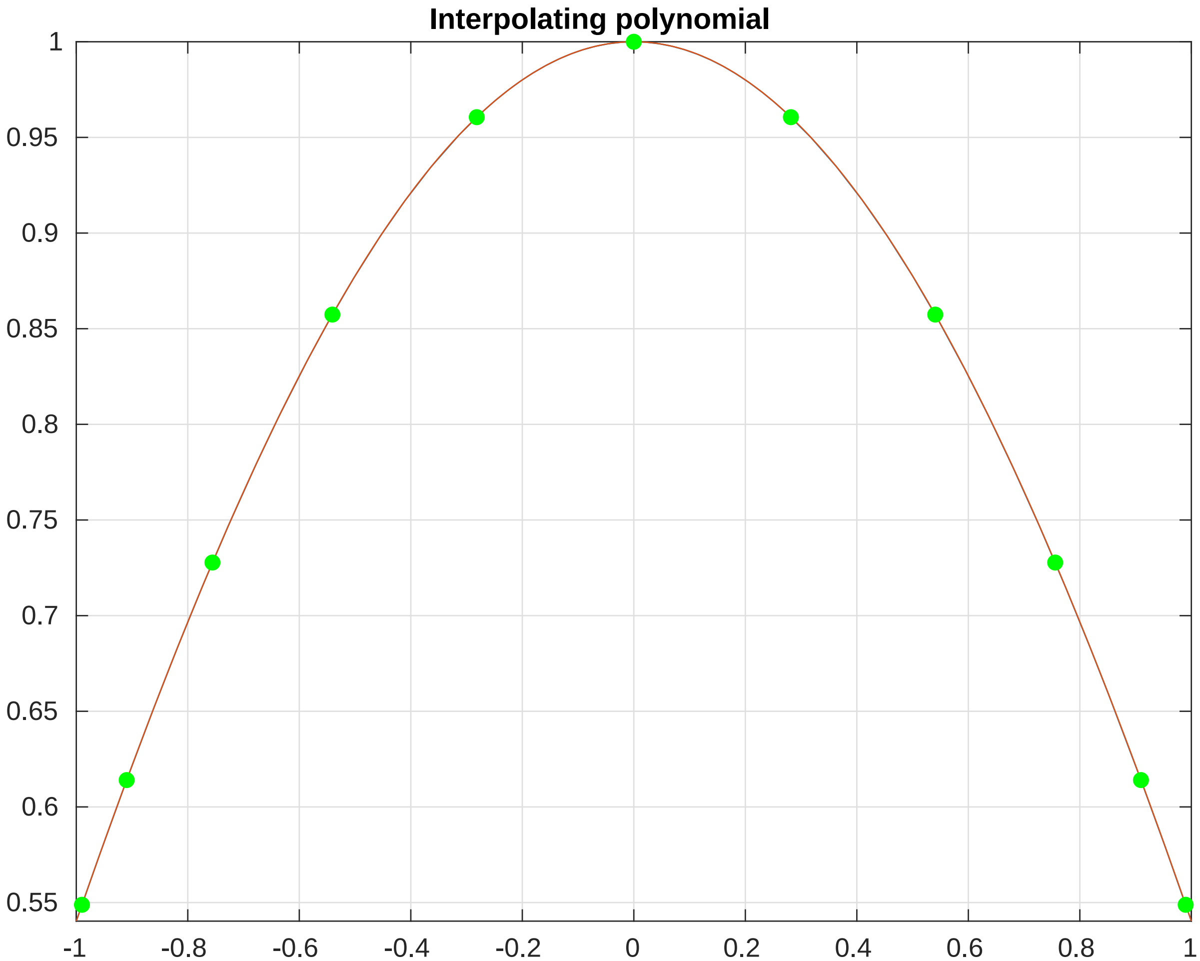
```
function out = objective(in)
    out = cos(in);
end
```



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
```



We see that either polynomial is particularly better or worse than the other. This is because the higher derivatives of $\cos x$ remain bounded in $[-1, 1]$. This is not the case for $1/(x^2 + 1)$. As shown in class, the n'th derivative of $1/(x^2 + 1)$ grows rapidly with respect to n. In fact, one can show that it grows at order $n!$. As explained earlier, it is true the non-uniform sampling z_k gives a smaller product $|x - z_1| \ldots |x - z_n|$. However, the fact that higher derivatives of $\cos x$ don't grow in n keeps the product on LHS of (1) small, regardless of the choice of sampling method.

Problem 2.

Interpolation gives an alternative method to approximate a function f by polynomials (other than a Taylor's theorem approximation). In this exercise, we investigate error estimates of this method. Let

$$f(x) = e^{\frac{x}{2}} \sin\left(\frac{x}{2}\right)$$

For evenly spaced points $0 = x_1 < x_2 < \ldots < x_n = 4$, let P_n be the corresponding interpolation polynomial.

a) Show that $|f'(x)| \leq e^{\frac{x}{2}}$ and that $|f''(x)| \leq e^{\frac{x}{2}}$ for all x.

b) It is known that (you don't have to verify) $|f^{(k)}| \leq e^{\frac{x}{2}}$ for any $x \in \mathbb{R}$ and $k \geq 1$. Find n such that

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

($\forall$ mean "for all".)

c) Find n such that the integral $\int_0^4 P_n(x)\,\mathrm{d}x$ approximates $\int_0^4 f(x)\,\mathrm{d}x$ with an error not exceeding 10^{-3}.

Solution

a) We will use the fact that $|\sin(x)| \leq 1$ and $|\cos(x)| \leq 1$ for all $x \in \mathbb{R}$.

$$f'(x) = \frac{1}{2}e^{\frac{x}{2}}\sin\left(\frac{x}{2}\right) - \frac{1}{2}e^{\frac{x}{2}}\cos\left(\frac{x}{2}\right) = \frac{1}{2}e^{\frac{x}{2}}\left(\sin\left(\frac{x}{2}\right) - \cos\left(\frac{x}{2}\right)\right)$$

Then we apply the triangle inequality to obtain

$$\left|\left(\sin\left(\frac{x}{2}\right) - \cos\left(\frac{x}{2}\right)\right)\right| \leq \left|\sin\left(\frac{x}{2}\right)\right| + \left|\cos\left(\frac{x}{2}\right)\right| \leq 1+1$$

So

$$|f'(x)| = \left|\frac{1}{2}e^{\frac{x}{2}}\left(\sin\left(\frac{x}{2}\right) - \cos\left(\frac{x}{2}\right)\right)\right| \leq \frac{1}{2}e^{\frac{x}{2}}|1+1| = e^{\frac{x}{2}}$$

Now for f''.

$$f''(x) = \frac{1}{4}\left(\cos\left(\frac{x}{2}\right) + \sin\left(\frac{x}{2}\right) - \sin\left(\frac{x}{2}\right) + \cos\left(\frac{x}{2}\right)\right) = \frac{1}{4}e^{\frac{x}{2}}\left(2\cos\left(\frac{x}{2}\right)\right) \leq \frac{1}{4}e^{\frac{x}{2}}(|1|+|2|+|1|) = e^{\frac{x}{2}}$$

(Hint for the general case: construct an induction proof that $f^{(n)}$ is a binomial of functions where $p = \sin$ and $q = \cos$.)

b) We can write an error bound for an interpolation polynomial as

$$|f(x) - P_n(x)| \leq \frac{e^{\frac{4}{2}}}{n!}\prod_{j=1}^{n}(x - x_j) \leq \frac{e^2}{n!}(n-1)!\left(\frac{4}{n-1}\right)^n = \frac{e^2}{n}\left(\frac{4}{n-1}\right)^n$$

With a calculator we can find that $n = 11$ is the smallest n which satisfies the bound on $[0, 4]$.

c) We require the inequality

$$\left|\int_a^b f(t) - g(t)\,\mathrm{d}t\right| \leq \int_a^b |f(t) - g(t)|\,\mathrm{d}t$$

(You do not have to prove this inequality)
We want to find n such that

$$\left|\int_0^4 f(t)\,\mathrm{d}t - \int_0^4 P_n(t)\,\mathrm{d}t\right| = \left|\int_0^4 f(t) - P_n(t)\,\mathrm{d}t\right| \leq 10^{-3}$$

Then

$$\left|\int_0^4 f(t) - P_n(t)\,\mathrm{d}t\right| \leq \int_0^4 |f(t) - P_n(t)|\,\mathrm{d}t \leq \int_0^4 \sup_{x\in[0,4]}|f(x) - P_n(x)|\,\mathrm{d}t = \sup_{x\in[0,4]}|f(x) - P_n(x)|\int_0^4 1\,\mathrm{d}t$$

Then

$$= (4-0)\sup_{x\in[0,4]}|f(x) - P_n(x)| \leq \frac{4e^2}{n}\left(\frac{4}{n-1}\right)^2$$

And we can test the right side with a calculator to find that $n = 10$ is the smallest n which satisfies the desired error bound. ($n = 7$ is the smallest n which gives a permissible error.)

Problem 3.

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

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

Solution

We can write an error bound for an interpolation polynomial as

$$|f(x) - P_n(x)| \leq \frac{|(-1)^n n!|}{(x+1)^{n+1}} \frac{1}{n!} \prod_{j=1}^{n} (x - x_j)$$

The product term here can be simplified further as x_j is evenly spaced.

$$\prod_{j=1}^{n} (x - x_j) \leq \frac{2}{n}(n-1)!$$

(see course lecture notes for the corresponding argument). Then

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

We can then evaluate the right side of the equality at several different values of n to find $n = 31$ is sufficient. (Partial credit will be awarded on quality of argument and accuracy of the associated result.)