+1 for presentation

CS375 HW9

Ryan Scherbarth

October 2024

1. We saw last homework the problems inherent in interpolating the range function

$$f(x) = \frac{1}{1+25x^2}, -1 \leq x \leq 1$$

with evenly spaced points. It is much better to interpolate on a grid made up of points that are clustered towards the endpoints. Interpolate $f(x)$ using Chebyshev points

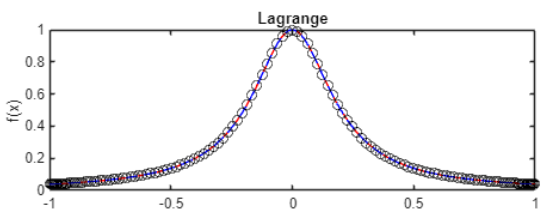
$$x_i = \cos(\frac{(2i+1)\pi}{2n+2}), i = 0, \ldots, n$$

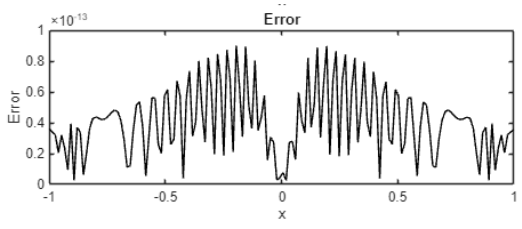
and lagrange interpolation. Use Chebychev points with $n = 150$. Do you see any of the issues that were illustrated in the previous homework (either Runge phenomena or the interpolant failing to match the given data)? Justify your answer with suitable points.

```
f = @(x) 1 ./ (1 + 25 * x.^2);
n = 150;
i = 0:n;
x_chebyshev = cos((2 * i + 1) * pi / (2 * n + 2));
y_chebyshev = f(x_chebyshev);
x_dense = linspace(-1, 1, 150);
y_dense = f(x_dense);
P = zeros(size(x_dense));

for j = 1:n+1
    L = ones(size(x_dense));
    for k = [1:j-1, j+1:n+1]
        L = L .* (x_dense - x_chebyshev(k)) / (x_chebyshev(j) - x_chebyshev(k));
    end
    P = P + y_chebyshev(j) * L;
end
```

Generates the following plots;





When using Chebychev points we've nearly entirely eliminated the Runge phenomena near the endpoints of this specific graph, so much so that our main area of error is closest to the exact center of the graph, which happens to be the most precise location.

It seems like for the given function we can completely alleviate any Runge phenomena concerns we might have.

2. For the functions $f_1(x) = \sin(x)$, $f_2(x) = |x|$ on the interval $[-1, 1]$. Tabulate the maximum error, $\max_{x\in[-1,1]} |f(x) - p(x)|$, as a function of n (make sure you compute the error on a fine grid). Take values fo n from 1 to 16. use Chebyshev points with Lagrange interpolation. Is the interpolating polynomial converging for both functions? If not, what could be going wrong?

Using the following matlab code we can generate the table;

```
f1 = @(x) sin(x);
f2 = @(x) abs(x);
n_values = 1:16;
x_fine = linspace(-1, 1, 1000);
max_f1 = zeros(length(n_values), 1);
max_f2 = zeros(length(n_values), 1);

for idx = 1:length(n_values)
    n = n_values(idx);

    i = 0:n;
    x_cheb = cos((2 * i + 1) * pi / (2 * n + 2));

    y_cheb_f1 = f1(x_cheb);
    y_cheb_f2 = f2(x_cheb);

    P_f1 = zeros(size(x_fine));
    P_f2 = zeros(size(x_fine));

    for j = 1:n+1
        L = ones(size(x_fine));
        for k = [1:j-1, j+1:n+1]
            L = L .* (x_fine - x_cheb(k)) / (x_cheb(j) - x_cheb(k));
        end

        P_f1 = P_f1 + y_cheb_f1(j) * L;
        P_f2 = P_f2 + y_cheb_f2(j) * L;
    end

    max_f1(idx) = max(abs(f1(x_fine) - P_f1));
    max_f2(idx) = max(abs(f2(x_fine) - P_f2));
end
```

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
f1	0.077	0.040	0.001	0.001	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.001	0.001	0.001
f2	0.706	0.217	0.270	0.123	0.172	0.087	0.126	0.067	0.100	0.055	0.083	0.046	0.071	0.040	0.062	0.035

We see that a smooth function like f_1 ends up converging rather quickly, especially so when compared to f_2. This is due to f_2 not being what we defined as a "nice" function in class. The main issue with f_2 is that it has a cusp at x=0. Here, the derivative from the left side does not match that of the right side, which throws off the calculation a lot.

3. We saw on previous homework that the polynomial interpolation behaves badly near the endpoints. In problem 1, we tackled this problem by using Chebyshev points instead of equispaced points. For this problem, we take a different approach and use natural cubic splines to interpolate the function. That is, we will find a natural cubic spline $S(x)$ on $[-1, 1]$ that interpolates the equispaced knots $-1 = x_0, x_1, \ldots, x_n = 1$ and function values $y_0 = f(x_1), \ldots, y_n = f(x_n)$

   (a) **Matlab problem** Write a MATLAB function coeffs = spline3_coeff(t,y) that takes as input the knot vector $t = [x_0, x_1, \ldots, x_n]$ and the function values $y = [y_0 = f(x_0), \ldots, y_n = f(x_n)]$ and return the vector $x = [0 = z_0, z_1, z_2, \ldots, z_{n-1}, z_n = 0]$. The z vector is computed by setting up and solving a tridiagonal system and discussed during lecture. You can use the MATLAB backslash for solving the tridiagonal system.

```
function coeffs = spline3_coeff(t, y)
    n = length(t) - 1;
    h = diff(t);
    d = zeros(n-1, 1);
    for i = 2:n
        d(i-1) = (6 / (h(i-1) + h(i))) * ((y(i+1) - y(i)) / h(i) - (y(i) - y(i-1)) / h(i-1));
    end

    a = h(1:n-1);
    b = 2 * (h(1:n-1) + h(2:n));
    c = h(2:n);
    T = diag(b) + diag(a(2:end), -1) + diag(c(1:end-1), 1);
    z = zeros(n+1, 1);
    z(2:n) = T \ d;
    coeffs = z;
end
```

   We store the second derivatives in vector z, then with our boundaries $z_0 = 0$ and $z_n = 0$. In the for loop each item in the right hand vector d is the difference of slope for 2 consecutive elements, and then we scale that value to make sure the spline's second derivative is continuous across each knot.

(b) **Matlab problem** Compute the natural cubic spline, $S(x)$, using the function you just wrote for $t = [-1 = x_0, x_1, \ldots, x_n = 1]$ and $y = [y_0 = f(x_0), y_1 = f(x_1), \ldots, y_n = f(x_n)]$ for $n = 11$ and plot $S(x)$ as well as $f(x)$ on $[-1, 1]$. How does the natural cubic spline behave relative to the polynomial interpolation in the last homework?
You may use the function eval_spline.m, provided with this assignment, to evaluate $S(x)$

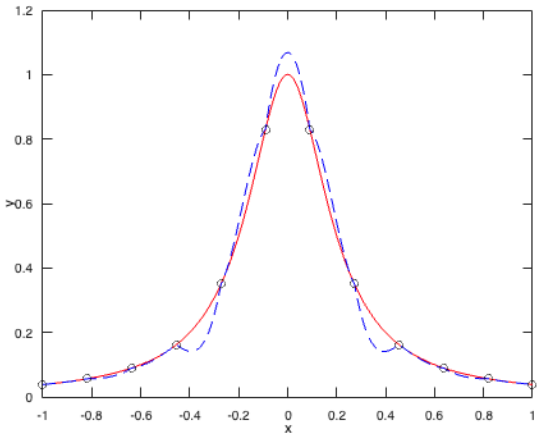
```
function coeffs = spline3_coeff(t, y)
    n = length(t) - 1;
    h = diff(t);

    d = zeros(n-1, 1);
    for i = 2:n
        d(i-1) = (6 / (h(i-1) + h(i))) * ((y(i+1) - y(i)) / h(i) - (y(i) - y(i-1)) / h(i-1));
    end

    a = h(1:n-1);
    b = 2 * (h(1:n-1) + h(2:n));
    c = h(2:n);
    T = diag(b) + diag(a(2:end), -1) + diag(c(1:end-1), 1);
    z = zeros(n+1, 1);
    z(2:n) = T \ d;

    coeffs = zeros(n, 4);
    for i = 1:n
        coeffs(i, 1) = y(i);
        coeffs(i, 2) = (y(i+1) - y(i)) / h(i) - h(i) * (2*z(i) + z(i+1)) / 6;
        coeffs(i, 3) = z(i) / 2;
        coeffs(i, 4) = (z(i+1) - z(i)) / (6 * h(i));
    end
end
```

First we calculate the second derivatives of each knot , then we connect them to form a smooth line across the entire interval. We immediately use the computed second derivatives to determine the cubic polynomial coeff for each interval, which gives us the quadratic, linear, and cubic terms we need.



this looks quite odd... I think you n