HW 3 Solutions

1a

The matrix has 4’s on the main diagonal and 1’s on the super and sub-diagonal. The rest of the entries in most of the rows (2:n-2) are 0’s. So, for each of these rows, we have that the diagonal entry is larger than ever other entry in the row. Additionally, the sum of the off-diagonal entries is 2, which is less than the diagonal entry of 4. The matrix M is also a square matrix (n-1) by (n-1). The definition of a diagonally dominant matrix is a square matrix with diagonal entries greater than or equal to the sum of the rest of the entries in that row. Strict diagonal dominance has the diagonal greater than the sum of the rest of the row, which is what we have. It turns out there are theorems that allow us to say that this matrix is nonsingular and hence we are guaranteed a unique solution to this linear system. So, as long as boundary conditions in 1st and last row have rows that are also strictly diagonally dominant, we will have a unique solution to the linear system and a unique solution to spline coefficients—hence, a unique cubic spline.

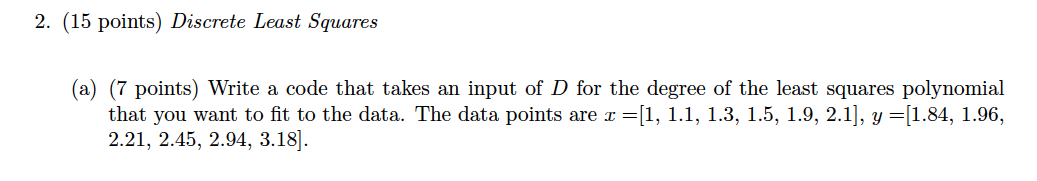
1b

Memory: To simplify the discussion, let n-1=30. That means we need to store 30\*30=900 entries. This could mean a lot of memory and a lot of calculations with all of these entries. It would be a huge memory savings if we could store <<900 entries!

If we look at the matrix, there are many zero entries. Is this really needed? Especially since we want them to be exactly zero and have operations such as 0+#=# and 0\*#=0 to be exact. However, since we have floating point error on each of these matrix entries, m\_ij=fl(m\_ij)+e\_ij where fl refers to the floating point representation and e\_ij refers to the error in representing each entry. So, this means that 0+# is not equal to that # and 0\*# is not exactly equal to zero.

The thought process here is two-fold: It would be nice to have a way to have a computational way to only store non-zero values of a large matrix and somehow have Matlab etc. know that all of the non-specified entries are exactly zero in order to reduce storage and decrease error (if it knows it is exactly zero, all computations of 0+#=# and 0\*#=0 will be exact).

Sparse matrices in matlab do exactly this. A user only specifies the non-zero entries. There is no round-off error in zero entries since Matlab knows by definition that they are 0.



clear all

prompt = 'D value?';

D = input(prompt); %Degree of Polynomial

xdata=[1 1.1 1.3 1.5 1.9 2.1];

ydata=[1.84 1.96 2.21 2.45 2.94 3.18];

n=length(xdata);

for i=1:(D+1)

    for j=1:(D+1)

        A(i,j)=sum(xdata.^((i-1)+(j-1)));

    end

end

A

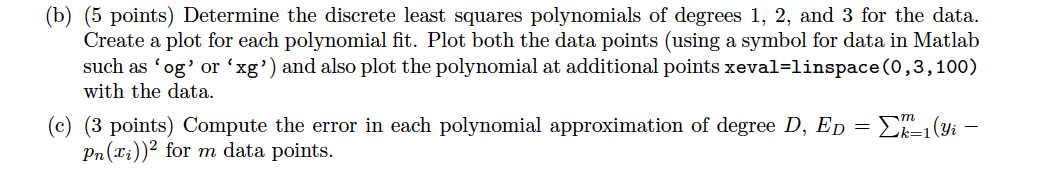
for j=1:(D+1)

    R(j,1)=sum(ydata.\*xdata.^(j-1));

end

R

c=A\R



syms x

LSfit=0;

for i=1:(D+1)

    LSfit=LSfit+c(i)\*x^(i-1);

end

xeval=linspace(0,3,100);

set(0,'defaultaxesfontsize',24)

set(0, 'defaultAxesfontweight', 'bold')

figure(1)

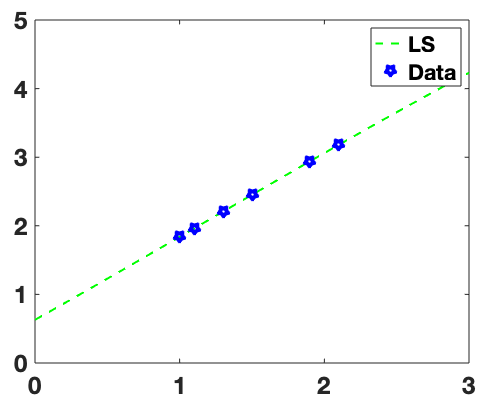
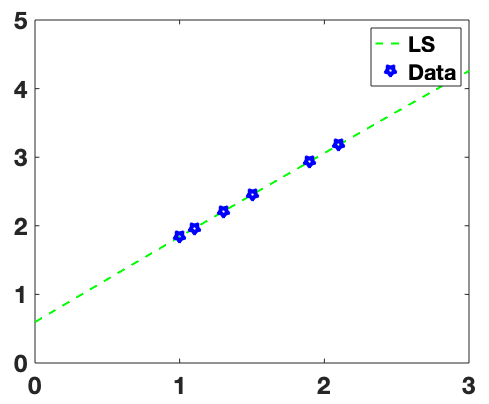
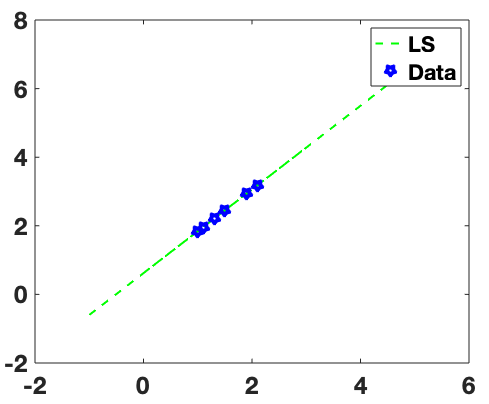
plot(xeval,subs(LSfit,xeval),'--g','LineWidth',2)

hold on

plot(xdata,ydata,'pb','LineWidth',10,'MarkerSize',10)

legend('LS','Data')

set(gcf,'Color','white')



From left to right, D=1, 2, and 3.

Outputting LSfit or coefficients c in Matlab:

D=1: 0.6209 + 1.2196 x

D=2: 0.5966+1.2533x -0.0109x^2

D=3: 0.6290+1.1850x+0.0353x^2-0.01x^3

They all look similar, we would have to zoom out to really see that these are actually degree 1, 2, and 3 polynomials. We can compare error as follows:

Error=double(sum((ydata-subs(LSfit,xdata)).^2))

For D=1: 2.7194e-05

For D=2: 1.8015e-05

For D=3: 1.7407e-05

So… higher degree polynomial has slightly less squared error between data points and function fit.