GI07 Homework #1 John Goodacre

Question 1 - Gradient Descent

a) Produce a plot similar to figure 1 in the question sheet.

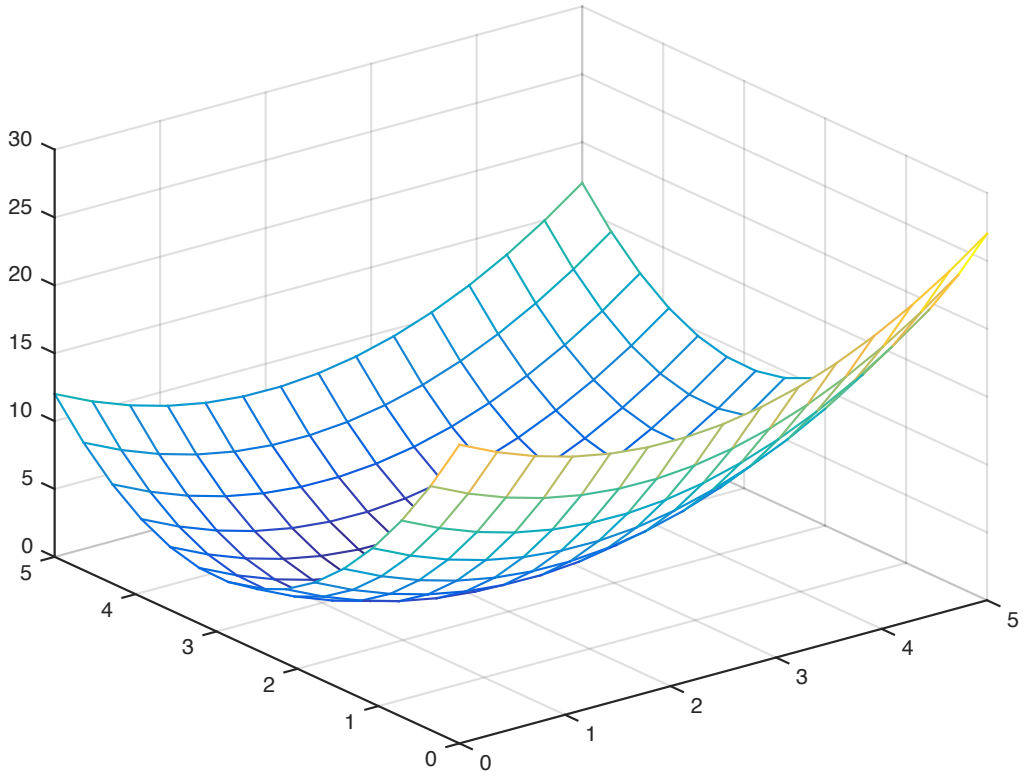


Figure 1

This was created with the following commands

```
[X,Y] = meshgrid(linspace(0,5,15),linspace(0,5,15));
mesh(X,Y,fcarg(X,Y));
```

b) Modify gradesc.m to produce figures from the question sheet.

```
function soln = graddesc(f, g, i,e, t)
% gradient descent
% f -- function
% g -- gradient
% i -- initial guess
% e -- step size
% t -- tolerance
gi = feval(g,i) ;
iarray=[i,feval(f,i)];
while(norm(gi)>t)  % crude termination condition
  i = i - e .* feval(g,i) ;
  gi = feval(g,i) ;
  iarray = [iarray;i,feval(f,i)];
end
soln = iarray ;
```

I simply stored the values in the descent in an array.

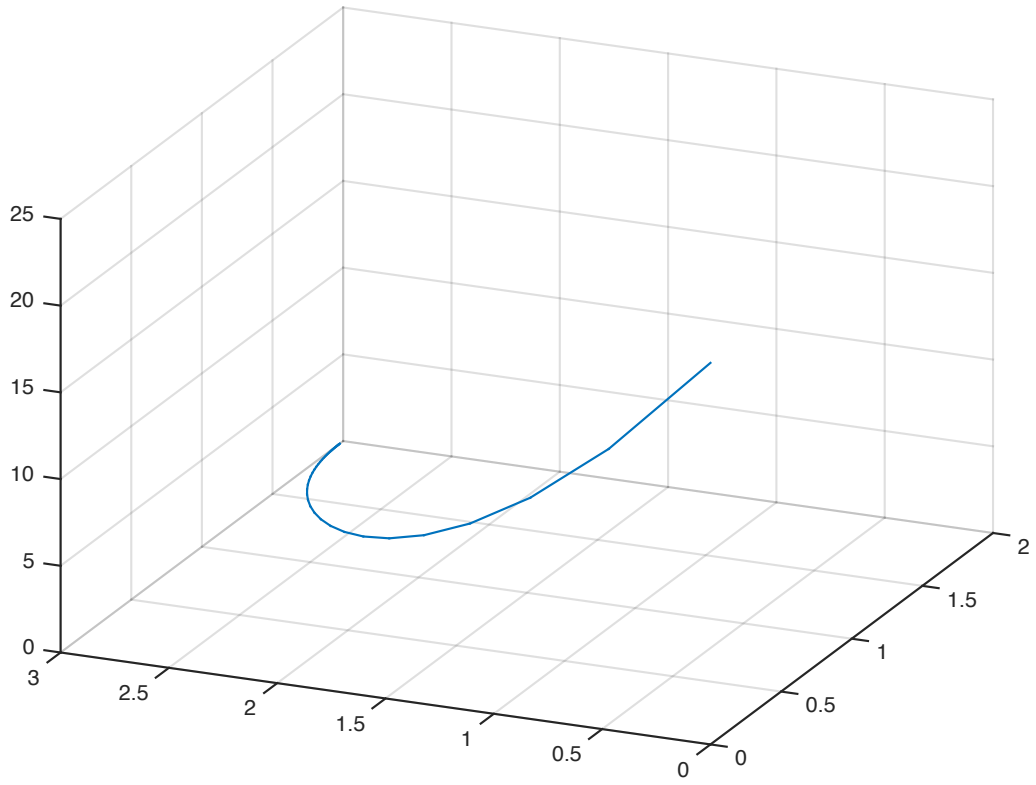


Figure 2

This was called using the following code

```
%Q1bi graddesc altered to return a matrix of values showing progress
%include changed graddesc.m in answers
result = graddesc('fc','dfc',[0,0],0.05,0.05);
figure
grid on

%Q1bii replicate the spiral plot - to make smooth i have reduced stepsize in
%graddesc call - q11b.fig
plot3(result(:,1),result(:,2),result(:,3))
grid on
```

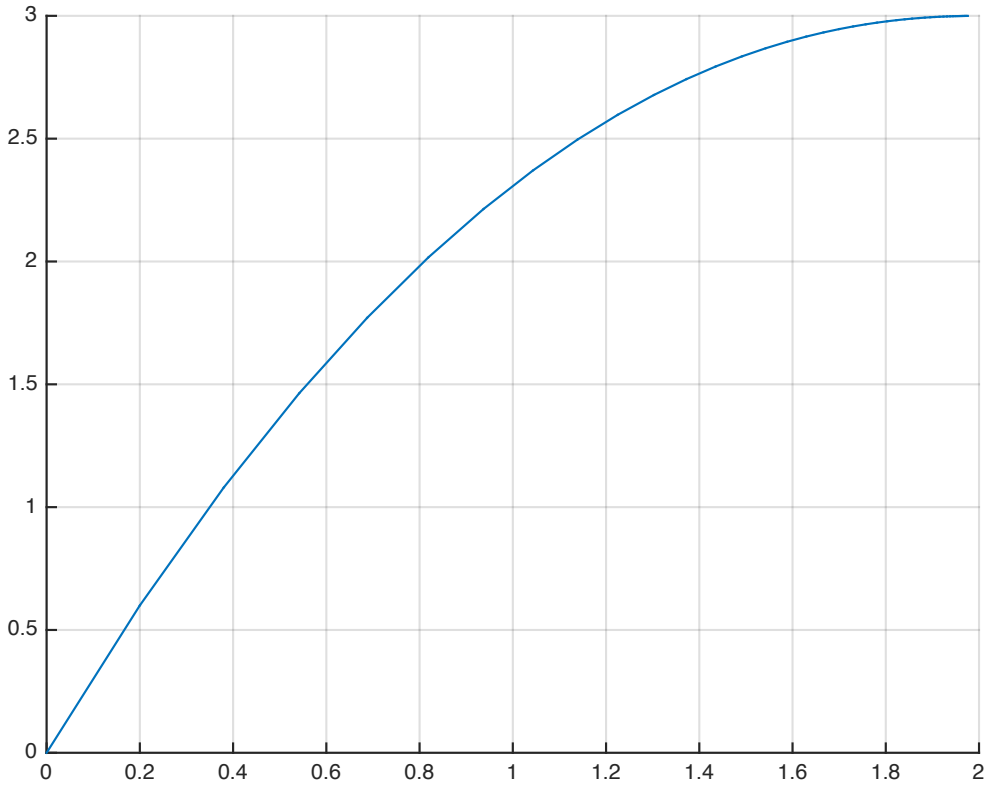


Figure 3

Question 2:

a) Give a matlab function to compute least squares by gradient descent.

```
function soln = mydescent( A,b, guess, step, tol )

% gradient descent for linear regression
% minimising (Ax-b)'(Ax-b)
% gradient is 2*A'(Ax-b)
% guess -- initial guess
% step -- step size
% tol -- tolerance

grad = 2*A'*(A*guess'-b');
iarray=[guess,(A*guess'-b')'*(A*guess'-b')];
while(norm(grad)>tol)
  guess = guess - step.* grad' ;
  grad = 2*A'*(A*guess'-b');
  iarray = [iarray;guess,(A*guess'-b')'*(A*guess'-b')];
end
%soln = guess
soln = iarray;
end
```

b) Use the function to give a least squares solution

Clearly calling the above function does a little more as I am now storing all the values in an array to answer the question afterwards. However calling mydescent(A, b, guess, step, tol), the last row will return the following values 1.2855, 0.5715.

Calling A\b' will instead give 1.2857 and 0.5714. Hence we can see this fairly inefficient method matches the more conventional.

c) Visualise with plot

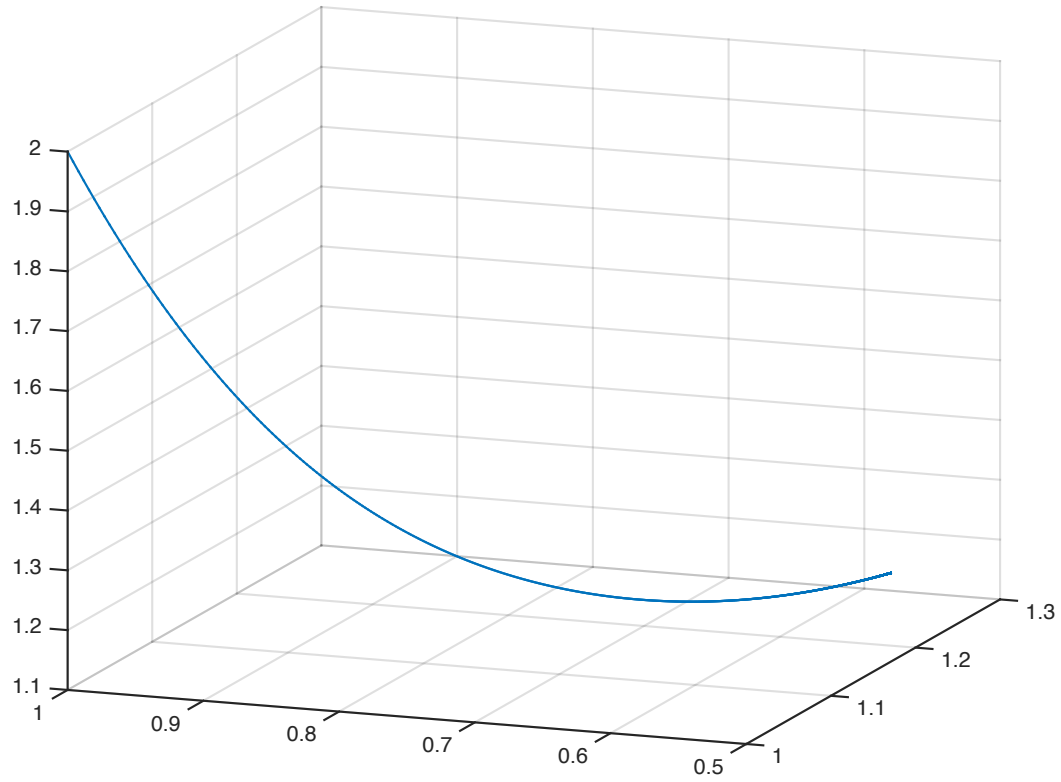


Figure 21 Plot of gradient descent Q2b

Clearly one can rotate the figure in matlab to also show the projection as for the previous question.

3 a) Does there exist a starting point and step size λ such that gradient descent on f(x)=$|x-1|^3$ nontrivially converges to 1?

A couple of observations, first we are looking for a convex function which does apply here. The question appears to imply a fixed step size λ. In this case even with a well behaved function we need the step size to be small enough relative to the starting point so we do not diverge.

In this case f'(x)=$3(x-1)|x-1|$, so without loss of generality if we assume at step size n, that x(n)-1 is positive and x(n+1) -1 is negative we definitely need

$$x_n - 1 > 1 - x_n + 3\lambda(x_n - 1)^2$$

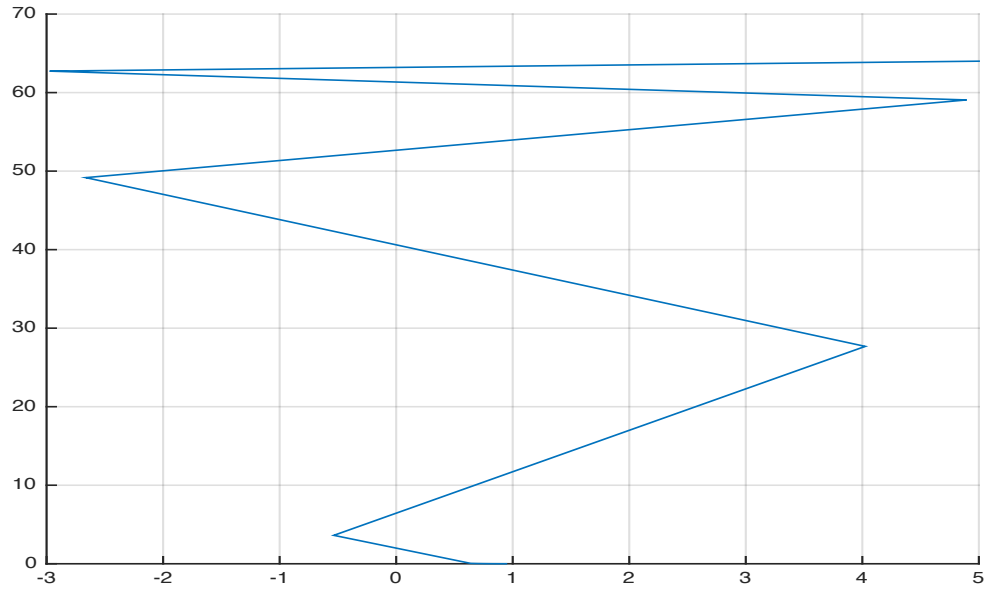
and so require

$$\lambda < \frac{2}{3(x_n - 1)}$$

For example setting our starting guess at 5, we need λ<1/6. The following code illustrates with an example.

```
result = graddesc('fd','dfd',[5],100/602,0.01);
figure
grid on
hold on
plot(result(:,1),result(:,2))
result

function rtn= fd(c)
x=c(1) ;
rtn = abs((x-1).^3);

function rtn= dfd(c)
x=c(1) ;
rtn = 3.*(x-1).*abs(x-1) ;
end
```

The x-axis is simply our value of x, and the y-axis our derivative. The point is that my step size λ is very close to the magical 1/6 value (100/602), thus we see the jumping around behavior. Clearly if the step size was >1/6 then we would diverge.



Assuming we have chosen a sensible step size then gradient descent does converge, at least in mathematical terms.

It is easy to see that we can always reach a value of 1+1/n (or 1-1/n). Without loss of generality assume we reach 1+1/n, n a large integer. The next step will thus be 1+1/n – $3/n^2$ i.e. $3/n^2$closer to 1. So in mathematical terms we can always be close to 1 (our solution) with arbitrary precision. In computer science terms this is a very convergence. We get closer at a much slower and slower rate.

3 b) Does there exist a starting point and step size λ such that gradient descent on f(x)=$|x-1|^{1/2}$ nontrivially converges to 1?

Note, a couple of things, first this function is convex. Indeed, as we get closer to 1, the derivative becomes higher and higher.

$$f'(x) = \frac{(x-1)}{2|x-1|^{3/2}}$$

This means that for a fixed step size λ, at a certain point gradient descent will move us much further away from the solution and so we do not converge.

For example if x = 1+1/n, then our next step will be $1 + \frac{1}{n} - \frac{\lambda\sqrt{n}}{2}$
For large n and fixed λ, the square root term dominates and we jump to the other side of the 'descent' further than where we were.

3 c) For which λ and interval does $f(x) = x^4 + 5x^2$ nontrivially converges to 0?
The work I did for part a) is relevant here. In short we look at conditions on λ where $x_k > x_{k+1}$.

$$f'(x) = 4x^3 + 10x$$

Without loss of generality the case we need to check is where $x_k > 0 \; and \; x_{k+1} < 0$
Therefore we need $\lambda < \frac{1}{2{x_k}^2+5}$

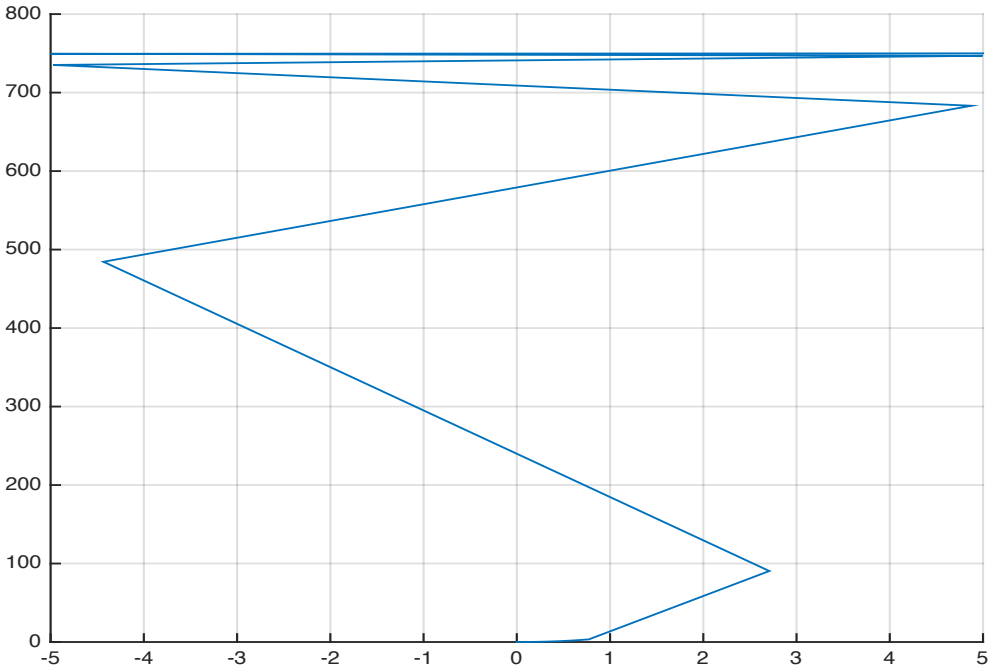
For example on the interval (-5,5) , $\lambda < \frac{1}{55}$

I coded the following to check this result.

```
result = graddesc('fe','dfe',[5],0.01818,0.001);
figure
grid on
hold on
plot(result(:,1),result(:,2));

result

function rtn= fe(c)
x=c(1) ;
rtn = x.^4+5*x.^2;

function rtn= dfe(c)
x=c(1) ;
rtn = 4.*(x.^3)+10.*x ;
end
```

The result being very similar to the work I did above, with the y-axis being the derivative and

the x axis our actual x-value.



I chose a value very close to 1/55 to show the jumping around behavior as the gradient descent attempts to converge. If the step size was increased to say 0.02 then this would diverge.

Question 3 Linear Regression:

1. For each of the polynomial bases of dimension k=1,2,3,4 fit the data set {(1,3),(2,2),(3,0),(4,5)}.

a) Produce a plot similar to figure 1 with four different curves superimposed over the four data points.

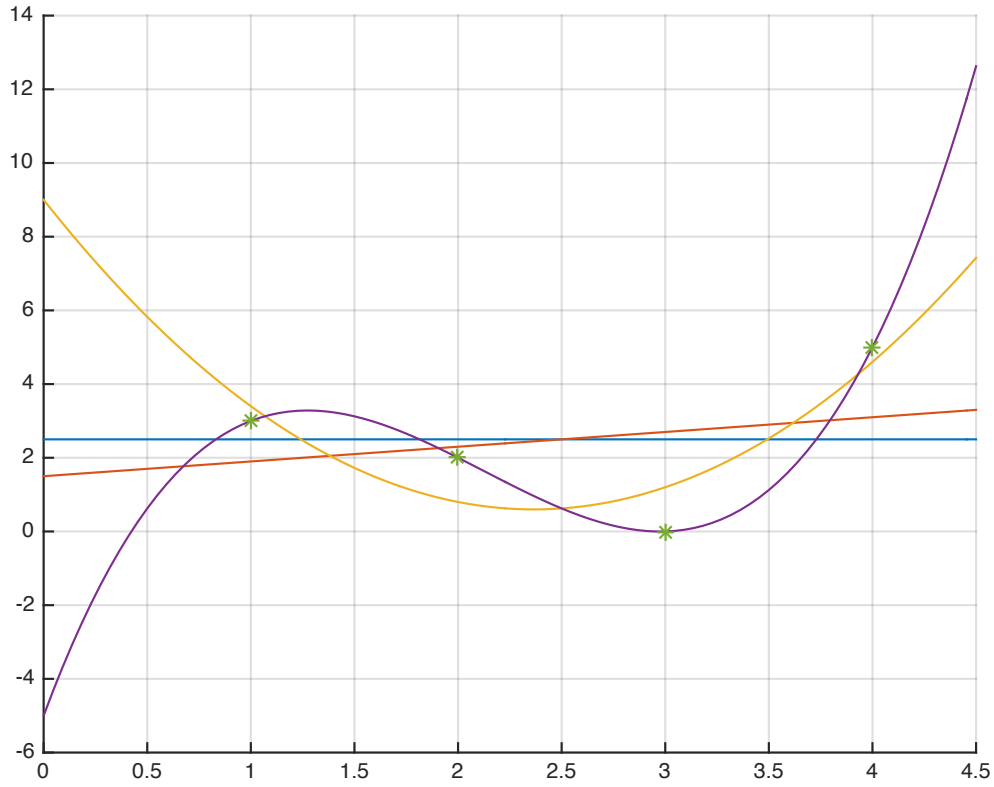


Figure 31 Data fitted with polynomial bases k=1,2,3,4

Obviously as I allow the degree of the highest polynomial to increase (more curves !) then I can fit the data easier. This is just training data so I can get closer and closer.

Because the task was reasonably small I initially coded this up manually.

```
A = [1 1;1 2;1 3;1 4];
b = [3 2 0 5];
basis2 = A\b';
C = [1 1 1 1;1 2 4 8;1 3 9 27;1 4 16 64];
basis4 = C\b';
D = [1 1 1;1 2 4;1 3 9;1 4 16];
```

```
basis3 = D\b';
E=[1;1;1;1];
F = [1;1;1;1];
basis1 = F\b';
```

Clearly this creates the coefficients for each respective polynomial basis. This can then be plotted using a linspace.

For example,

```
X = linspace(0,4.5,100);
Y4 = basis4(1,1)+basis4(2,1).*X+basis4(3,1).*(X.^2)+basis4(4,1).*(X.^3);
```

Together with the respective polynomial basis. Later on in the problem set we had to create many different bases so instead I coded up a function to do all this automatically.

b) Give the equations corresponding to the curves fitted for k=1,2,3

K=1, equation is $y = 2.5$
K=2, equation is $y = 1.5 + 0.4x$
K=3, equation is $y = 9 - 7.1x + 1.5x^2$
K=4, equation is $y = -5 + 15.17x - 8.5x^2 + 1.33x^3$

c) For each curve give the MSE

Here I wrote a function which given a set of data points and coefficients calculated the MSE assuming we are using a polynomial basis.

```
function mse = polymse( A, coeff )
% A is a matrix of points,  coeff is a vector of coefficients
% note here we are using a polynomial basis function of degree length coeff
% ie A = [ x1 y1; x2 y2; ... ; xm ym]
% coeff = [1 2 4 6...] ie 1+2x+4x^2+6x^3...
% returns MSE the mean squared error
polypt(:,1)=A(:,1);
polypt(:,2)=0;
for i=1:length(coeff)
   polypt(:,2) = polypt(:,2) +(A(:,1).^(i-1))*coeff(i);
end
sse = sum((A(:,2)-polypt(:,2)).^2);
mse = sse/length(coeff);
end
```

The mean squared errors are:
K=1, MSE1 = 13
K=2, MSE2 = 6.1
K=3, MSE3 = 1.0667

K=4, MSE4 = 0.514e-29, I suppose one should note the ridiculously low MSE4. Clearly I can fit n-data points with an nth degree polynomial perfectly.

2.

a) Write a function that takes μ and σ and returns random numbers with the normal distribution with the parameters as specified.

The code below is how I did this, n being the number of such variables I require.

```
function rv = normmusig( mu, sigma,n )
%generates n random variables with a normal distribution
% mean mu, variance sigma squared
rv = mu+sigma.*randn(1,n);
end
```

b)

i) Plot the function $sin^2(2\pi x)$ with the 30 data points superimposed.

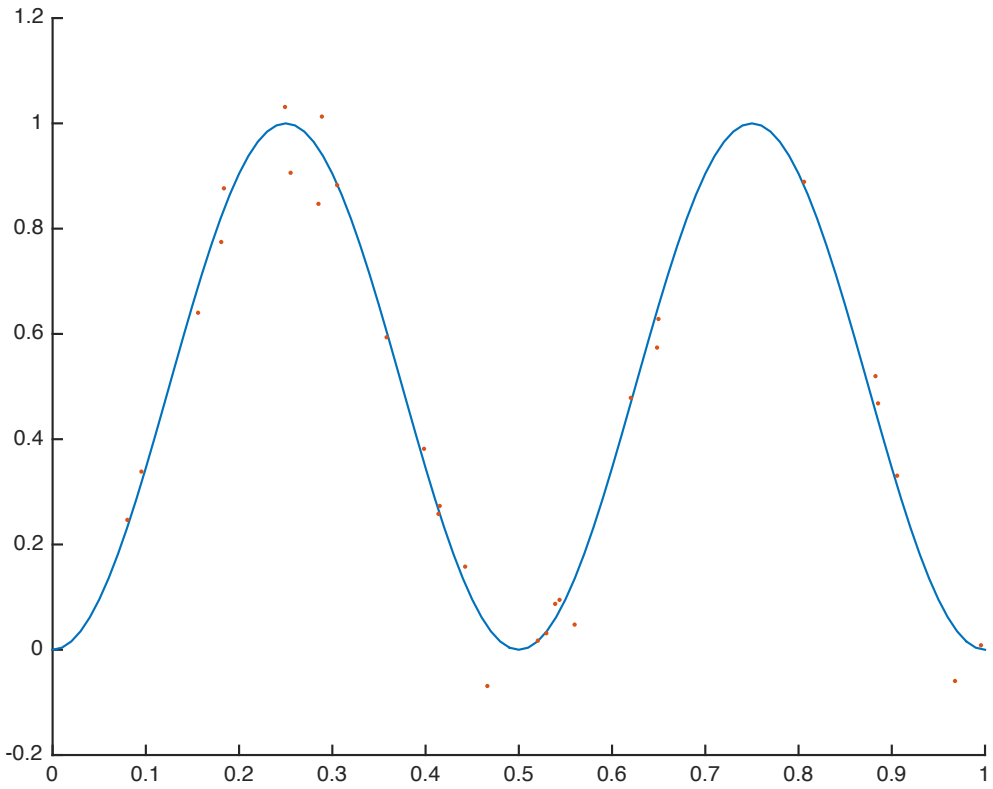


Figure 41 (sin(2pix))^2 + random data points

Here I wrote the following function

```
function gx = gphix(u, mu, sigma )
%generates n random variables where g(x) = sin^2(x) + N(mu,sigma^2)
%the input is uniform random numbers, the function returns
```

```
% a random function variable for each
rand_vec = normmusig( mu, sigma,size(u,1));

gx = (sin(2*pi*u)).^2 +rand_vec';
end
```

This uses the random normal generating function mentioned above.

ii) fit the data set with polynomial bases of dimension 2,5,10,14,18, plot them over the data points.

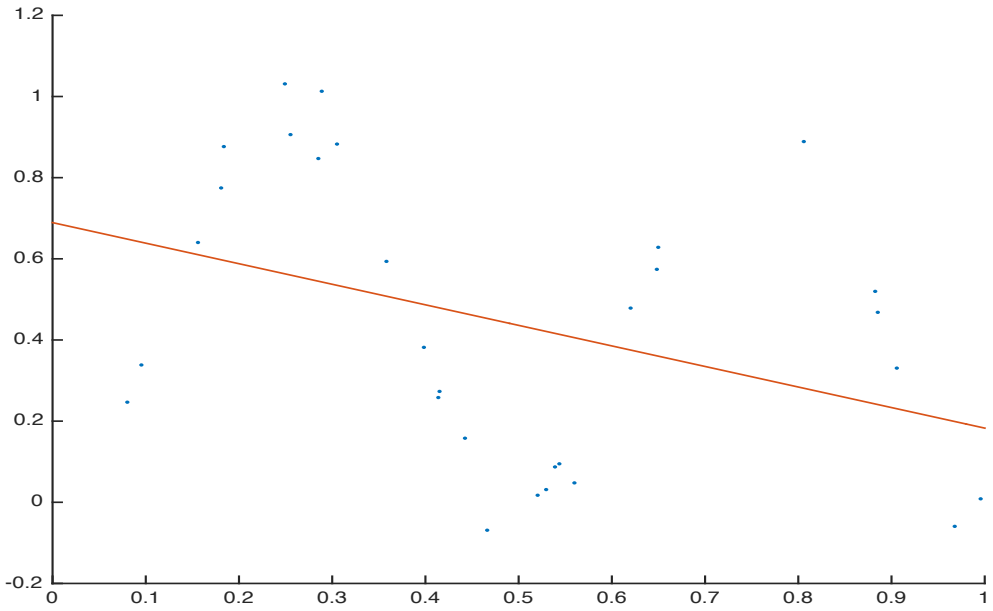


Figure 5 Data fitted with K=2

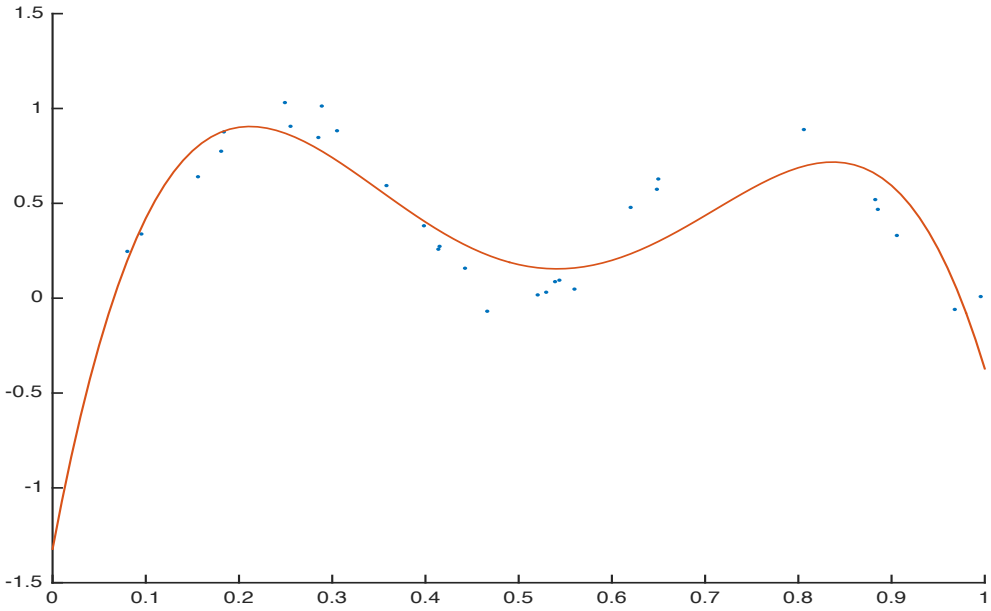


Figure 6 Data fitted with k=5

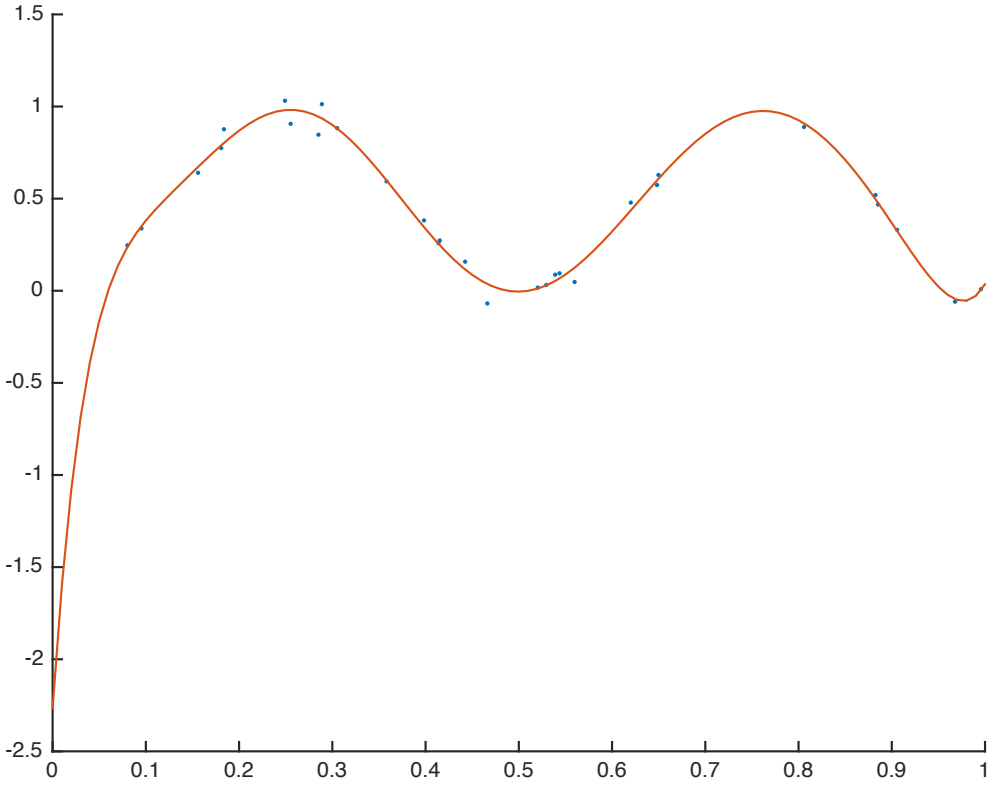


Figure 7 Data Fitted with k=10

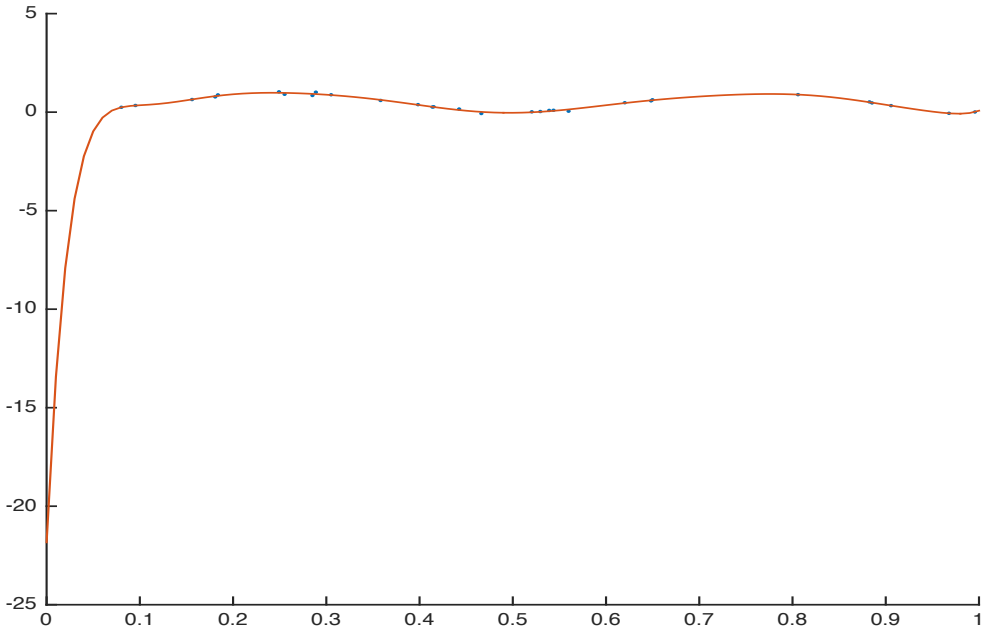


Figure 8 Data fitted with k=14

By this point I am getting far less comfortable. Note the data are barely dots, because away from the data the function is becoming quite extreme. This is basically over-fitting. What happens as the polynomial basis goes up in such situations is the coefficients of the polynomials will increase dramatically. The intuition here is that the curve is twisting very tightly to enable tight fitting to each data point. However, this means the function will veer to extremes away from the data.

The point is that if test data looks similar but not exactly the same as the training data then we will be great at fitting the training data but far worse as regards the test data. This can be mitigated with a regularisation parameter. For example, in ridge regression we effectively penalise the squared weights of these coefficients.

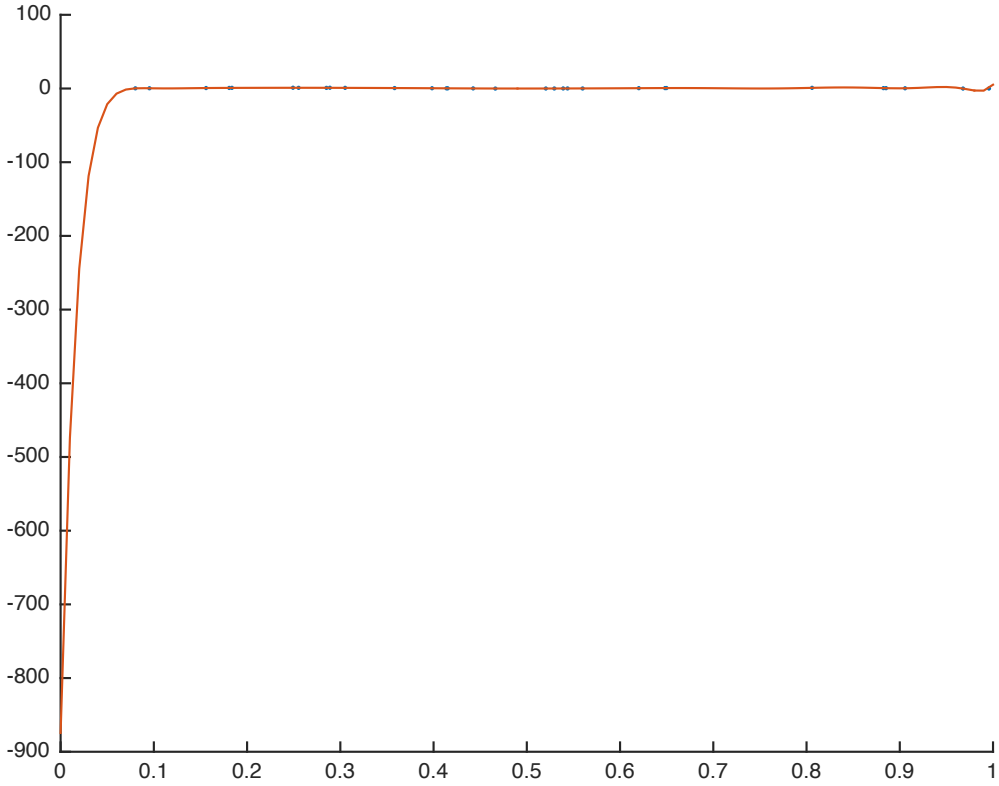


Figure 9 Data fitted with K=18 - gross overfitting

I wrote the following code to generate a polynomial basis for data with a pre-specified degree.

```
function basis = generate_poly( Data2fit, basis_degree )
%returns a polynomial basis for a set of datapoints with a pre
%specified degree for the polynomial

%A = Data2fit(:,1);
b = Data2fit(:,2);
A = [ones(size(Data2fit(:,1)))];

%loop through to create the polynonial basis
for i=2:basis_degree
   A = [A Data2fit(:,1).^(i-1)];
end
basis = A\b;
end
```

C) Let the training error denote the MSE of the fitting of the data set with polynomial basis k. Plot the log of the training error versus the polynomial basis.

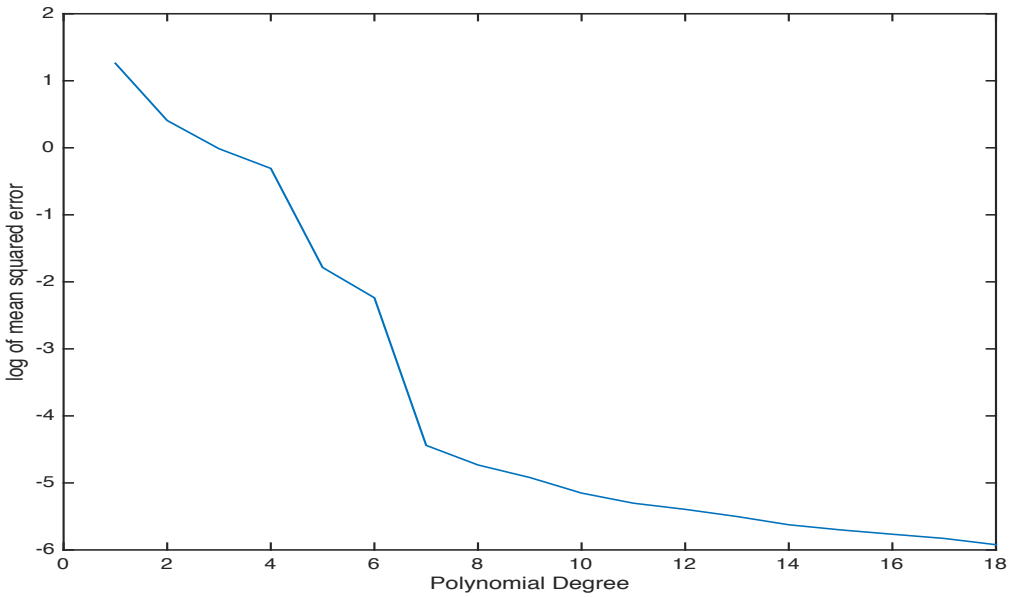


Figure 10 Training error with different polynomial bases

Here I simply looped through and made repeated calls to my polymse function. There is nothing significant in terms of concept or new code, so I haven't given any further illustration.

d) Generate a test set T of a thousand points from the same distribution mentioned above. Plot the log of the test set error.

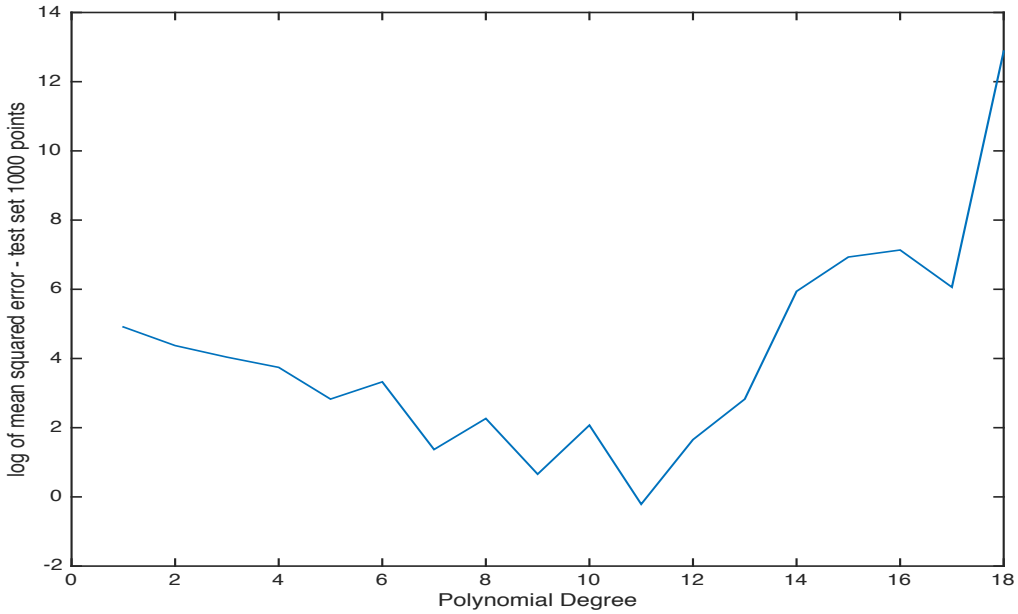


Figure 11 Polynomial degree/ MSE test set 1000 points

In figure 11, we clearly see the over-fitting as the polynomial degree increases, despite the training error decreasing, the polynomials fit the test set data increasingly badly.

e) Repeat but instead of one run, do 100 runs and average the results, then show the log of the respective MSE's.

The code to do the last part is shown below. Basically I use a cell structure to store my different polynomials. And generate a thousand test data-points using tuniforms and gphix (given above). I store the training errors and test errors and then average them (over the 100 runs). The log is then charted.

```
%do parts 2c-e 100 times and instead average the results
train_errors = zeros(100,18);
test_errors = zeros(100,18);
for j=1:100
   ms_errors = [];
   poly_cell = cell(1,18);
   for i=1:18
      poly_cell{i} = generate_poly(data2fit,i);
   end

   for i=1:18
      ms_errors = [ms_errors polymse(data2fit,poly_cell{i})];
   end
   train_errors(j,:) = ms_errors;
```

```
    %generate 1000 data points for our test set
    tuniforms = sort(rand(1000,1));
    test_gx = gphix(tuniforms,0,0.07);

    ts_errors = [];
    for i=1:18
        ts_errors = [ts_errors polymse([tuniforms test_gx],poly_cell{i})];
    end
    test_errors(j,:) = ts_errors;
end
av_train_error = mean(train_errors);
av_test_error = mean(test_errors);

figure
hold on
plot(1:18,log(av_train_error));
plot(1:18,log(av_test_error));
```

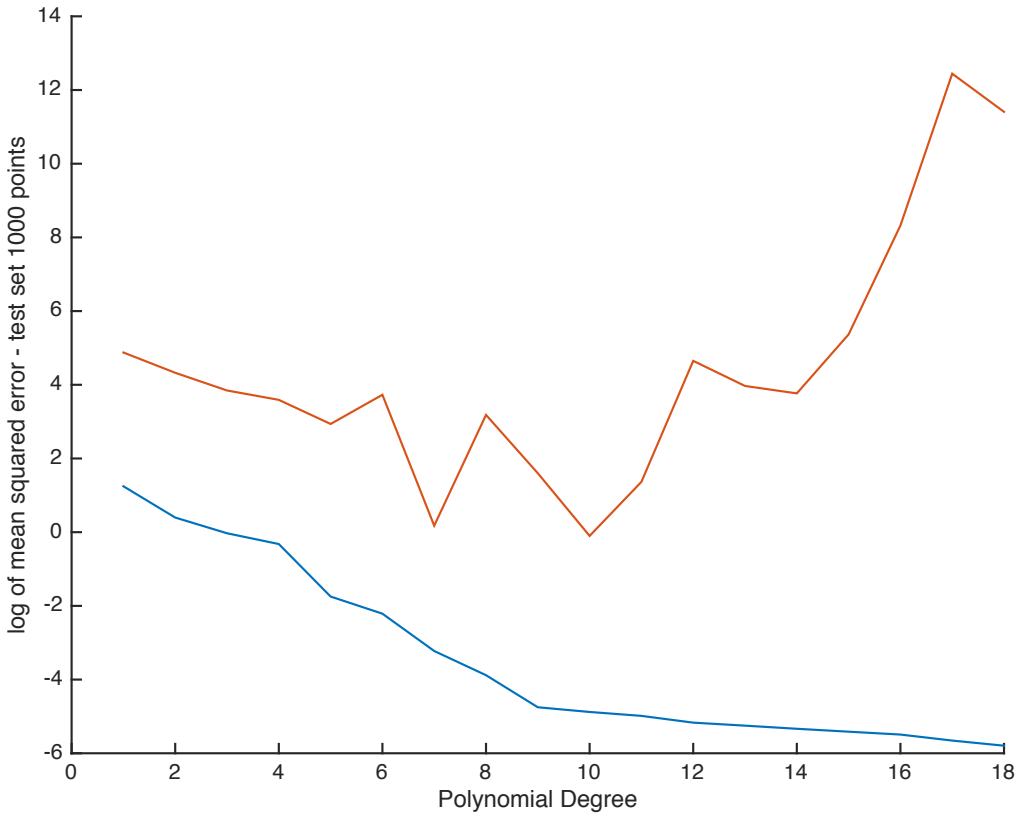


Figure 12 Red Test/ Blue Training -MSE's as K increases

f) Do the same for sin(kπx) as a basis.

This is basically the same code, however I wrote two more functions.

```
function basis = generate_sin( Data2fit, basis_degree )
%returns a basis for a set of datapoints with a pre
%specified degree for the sin basis

%A = Data2fit(:,1);
b = Data2fit(:,2);
A = [sin(pi.*Data2fit(:,1))];

%loop through to create the sin basis
for i=2:basis_degree
   A = [A sin(pi.*i.*Data2fit(:,1))];
end
basis = A\b;

end
```

```
function mse = sinmse( A, coeff )

% A is a matrix of points
% coeff is a vector of coefficients
% note here we are using a sin basis function of degree length coeff
% ie A = [ x1 y1; x2 y2; ... ; xm ym]
% coeff = [1 2 4 6...] ie 1+2sinpix+4sin2pix+6sin3pix...
% returns MSE the mean squared error
polypt(:,1)=A(:,1);
polypt(:,2)=0;
for i=1:length(coeff)
   polypt(:,2) = polypt(:,2) +sin(pi.*i.*A(:,1))*coeff(i);
end

sse = sum((A(:,2)-polypt(:,2)).^2);
mse = sse/length(coeff);

end
```

The two functions are the equivalent of the polynomial basis code. Clearly one could re-write so that any function is passed in rather than replicating each time.

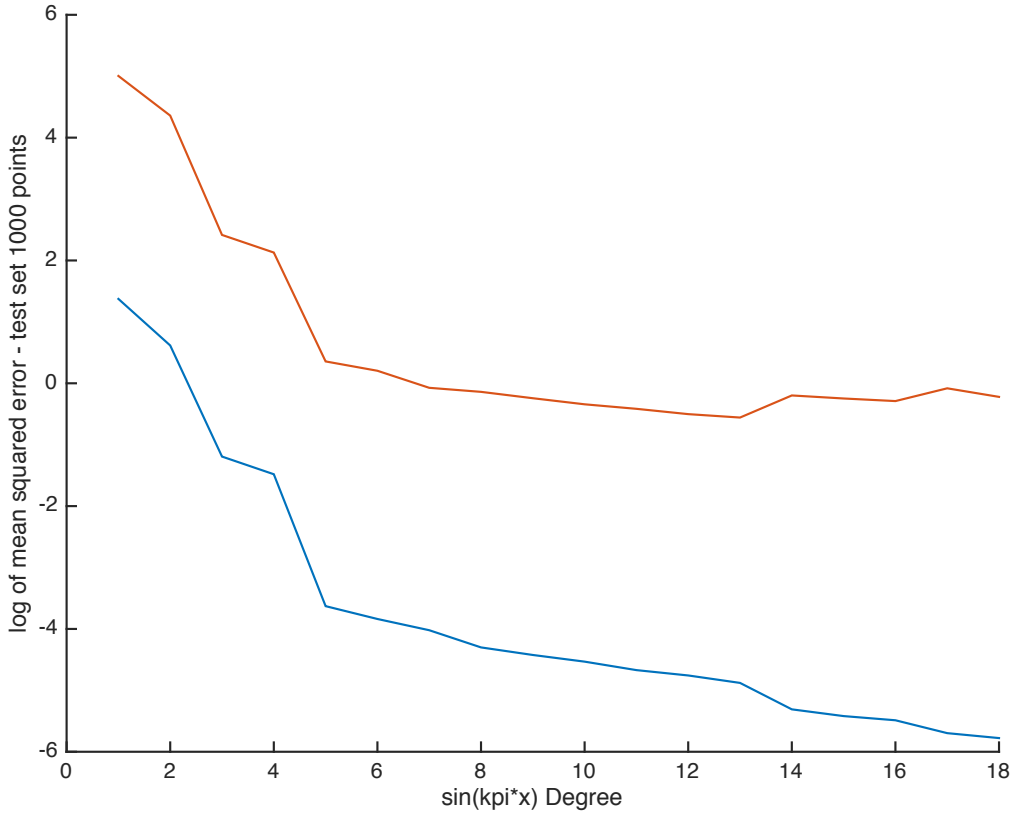


Figure 13 1000 test runs - similar to above with sin basis

Here notice we don't get the same drastic over-fitting. I haven't looked into this a great deal and could investigate further, but clearly the data was produced as a sin function plus noise and using a sin basis is advantageous. In the polynomial basis as we over-fit we will get very high coefficients on the polynomial terms, which will effect the fit to the test set.

4) Whack a Mole

A couple of observations.

a) Hitting a hole (or mole lol) and repeating puts one back in the same board state.
b) Hitting a hole and then another is the same as doing this in reverse.

At least in terms of search this reduces the problem drastically. For example, clearly on a board size n, one either hits a hole or not to reach a solution, this can be represented by a zero or one.

Also, because of the above then any solution should have less than n^2, hits. Clearly one can do worse than this, but any such solution can logically be represented with less moves.

I will reduce the board to a 3x3 board to gain intuition, each hole in the board can be set as either a 0 or 1.

If we start with a board in the state we wish to achieve …

$$\begin{matrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{matrix}$$

Then notice that a hit to the top left changes the board state to…

$$\begin{matrix} 1 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{matrix}$$

Clearly we can model this for each move, i.e. 4 corner hits, 4 edge hits, and 1 centre hit, and each one of these matrices multiplied by either a zero or 1 should equal the starting position of the game. (Because going from the starting position to all zeros is the same as the reverse – observation c!).

It is perhaps easier to show the matrix of moves for a 3x3 matrix, I represent each hit by a row (I count top left to bottom right, given 9 holes, we have a 9x9 matrix).

$$\begin{matrix} 1&1&0&1&0&0&0&0&0 \\ 1&1&1&0&1&0&0&0&0 \\ 0&1&1&0&0&1&0&0&0 \\ 1&0&0&1&1&0&1&0&0 \\ 0&1&0&1&1&1&0&1&0 \\ 0&0&1&0&1&1&0&0&1 \\ 0&0&0&1&0&0&1&1&0 \\ 0&0&0&0&1&0&1&1&1 \\ 0&0&0&0&0&1&0&1&1 \end{matrix}$$

If we call this matrix A. Then clearly Ax = b where b is the initial board position, simply a vector of zeros and ones, represents the whole problem.

The solution $x = A^{-1}b$ where of course the inverse is in $Z(2)$.
I have coded this below – starting with the 3x3 case. An observation – all operations here are in Z(2), i.e. matrix inversion is mod 2. I didn't know off the top of my head (ring theory, fields and galois theory was an awful awful long time ago), however I found a way to invert a matrix and then convert to the required base by looking it up on the web. This is included in the code below i.e. the round(det(A)*inv(A),2) part.

```
A = [1 1 0 1 0 0 0 0 0;
     1 1 1 0 1 0 0 0 0;
     0 1 1 0 0 1 0 0 0;
     1 0 0 1 1 0 1 0 0;
     0 1 0 1 1 1 0 1 0;
     0 0 1 0 1 1 0 0 1;
     0 0 0 1 0 0 1 1 0;
     0 0 0 0 1 0 1 1 1;
     0 0 0 0 0  1 0 1 1];
```

```
Ainv = mod(round(det(A)*inv(A)),2);
mod([0 0 0 1 0 0 1 0 0]*Ainv,2)
```

This gives as the solution:-

```
   1    1    1    0    1    0    1    0    0
```

To be clear hit holes, 1,2,3,5,7 to take the initial position to all zero states.

The hard bit…

At this stage I was unclear about what the 'hard' part of this question was, all that remained was to code this up. Clearly this is polynomial in the solution. The only thing we need is that A is invertible in Z(2). (i.e. matrix inversion mod 2).

I quickly found that in most dimensions this matrix is not invertible. To be clear in dimensions where it is for every board state we can always find a solution. However, where the determinant is zero it means the equation is not always solvable and depends upon the initial board state.

Somewhat frustratingly this means that there will be some initial board configurations where the sequence of moves is solvable, and some where it is not. Before continuing I will illustrate my code for populating boards of size n and checking if that dimension always has a solution.

```
mat_vals = []
for mat_size = 2:100
```

```
n=mat_size;
Whackmat = zeros(n^2);
for i=1:n
   for j=1:n
      %corner
      if i==1 && j==1
         Whackmat(i,j) = 1;
         Whackmat(i,j+1) = 1;
         Whackmat(i,j+n) = 1;
      end
      if i==n && j==n
         Whackmat(i*j,i*j) = 1;
         Whackmat(i*j,i*j-1) = 1;
         Whackmat(i*j,i*j-n) = 1;
      end
      if i==1 && j==n
         Whackmat(j,j) = 1;
         Whackmat(j,j-1) = 1;
         Whackmat(j,j+n) = 1;
      end
      if i==n && j==1
         Whackmat((i-1)*n+1,(i-1)*n+1) = 1;
         Whackmat((i-1)*n+1,(i-1)*n+2) = 1;
         Whackmat((i-1)*n+1,(i-2)*n+1) = 1;
      end
      %edge
      if i==1 && j ~=1 && j ~=n
         Whackmat(j,j) = 1;
         Whackmat(j,j-1) = 1;
         Whackmat(j,j+1) = 1;
         Whackmat(j,j+n) = 1;
      end
      if i==n && j ~=1 && j ~=n
         Whackmat((i-1)*n+j,(i-1)*n+j) = 1;
         Whackmat((i-1)*n+j,(i-1)*n+j-1) = 1;
         Whackmat((i-1)*n+j,(i-1)*n+j+1) = 1;
         Whackmat((i-1)*n+j,(i-1)*n+j-n) = 1;
      end
      if j==1 && i~=1 && i~=n
         Whackmat((i-1)*n+j,(i-1)*n+j) = 1;
         Whackmat((i-1)*n+j,(i-1)*n+j-n) = 1;
         Whackmat((i-1)*n+j,(i-1)*n+j+1) = 1;
         Whackmat((i-1)*n+j,(i-1)*n+j+n) = 1;
      end

      if j==n && i~=1 && i~=n
         Whackmat((i-1)*n+j,(i-1)*n+j) = 1;
```

```
            Whackmat((i-1)*n+j,(i-1)*n+j-n) = 1;
            Whackmat((i-1)*n+j,(i-1)*n+j-1) = 1;
            Whackmat((i-1)*n+j,(i-1)*n+j+n) = 1;
        end
        %centre
        if j~=1 && j~=n && i~=1 && i~=n
            Whackmat((i-1)*n+j,(i-1)*n+j) = 1;
            Whackmat((i-1)*n+j,(i-1)*n+j-n) = 1;
            Whackmat((i-1)*n+j,(i-1)*n+j-1) = 1;
            Whackmat((i-1)*n+j,(i-1)*n+j+1) = 1;
            Whackmat((i-1)*n+j,(i-1)*n+j+n) = 1;
        end
    end
end

mat_size
determ = det(Whackmat);
Winv = mod(round(det(Whackmat)*inv(Whackmat)),2)
%det(pinv(Whackmat))
%pseudoInv = pinv(Whackmat)
singular = sum(sum(Winv - zeros(mat_size^2)));
mat_vals = [mat_vals singular];

end
mat_vals %simply checks for singularity in various dimensions.
```

I accept this is not as pretty as could be, however the boards are populated and I am able to check if there is always a solution. For example, we always have solutions when n = 2,3,6,7,8,10,12,13,14,17,19…then it gets very difficult. We also have a very fast method for any position in each of these dimensions – i.e. invert the matrix multiply by initial position.

What to do when the matrix of moves is not invertible?

Investigation 1
It is clear that regardless of the initial position in any dimension one can simply hit the hole beneath any mole in each row and simply proceed down the rows until one reaches a position with moles in some holes in the bottom row and empty holes everywhere else. Looking at all sequences of holes in only the top row and striking with the same strategy from there one has a solution. That is if the bottom position matches one of the sequences reached from a top position configuration then we are done.

However, to enumerate all configurations from the top is not polynomial in terms of n. I suspect there is structure in the move matrix to enable us to reduce this but this was the end of my investigation here.

Investigation 2
Clearly the reason the matrix of moves is not invertible in some dimensions is because some of the rows are linearly independent. This means we can row reduce (in z(2) don't forget !) until we have rows of zeros at the bottom. These effectively correspond to redundant holes.

At this point I again needed to look up how one finds solutions to a linear system when the matrix is not invertible. I.e. there are some solutions but a solution is not guaranteed.

The answer was to find a pseudo-inverse. This is effectively augmenting my row reduced matrix with an identity matrix of the same dimension (not rank as the rank is lower due to my rows of zeros). Then one performs the usual Gaussian elimination. The result will be an $n^2$by n^2 matrix called the pseudo-inverse. Let us call it P. This whole set up is of course very similar to when we do least squares.

I was investigating but have not proven, whether $x = P^{-1}b$ is the whole solution to the problem. This is when b the board position has a solution, which is not guaranteed.

The attentive (crazy?) reader at this point will have noted a commented out line in my code above pseudoInv = pinv(Whackmat)

Whoopee, all problems solved. Matlab indeed has a pseudo-inverse function. However, I realized that the trick used earlier to convert an inverse into an inverse in base 2, does not work here (the determinant would be zero). Thus, I would be required to write my own Gaussian elimination mod 2. At this point I decided enough was enough and there was either a far easier solution without pseudo-inverses (Z2) and/or that other assignments needed taking care of. ☺

Source code:

As I ran through the exercises I used a main script called GradDecex.m, this is effectively the script that calls all the other functions I wrote. The only exception was my investigation into the mole problem. Mole.m. GradDecex does spit out all of the figures in one go (which I subsequently tidied for this document). The other functions I wrote or used have been illustrated in this document and should be in the same directory as GradDecex.m.