COMPGI07: Homework #1

Antonio Remiro Azocar, MSc Machine Learning

Submission date: 5 December 2016

1 Gradient Descent

(1a) The following commands are used to reproduce the initial plot:

```
clear

[X,Y] =  meshgrid(linspace(0,5,15), linspace(0,5,15));
mesh(X,Y,fcarg(X,Y));
xlabel('x')
ylabel('y')
zlabel('f(x,y)')
```

The plot obtained is presented in Figure 1:

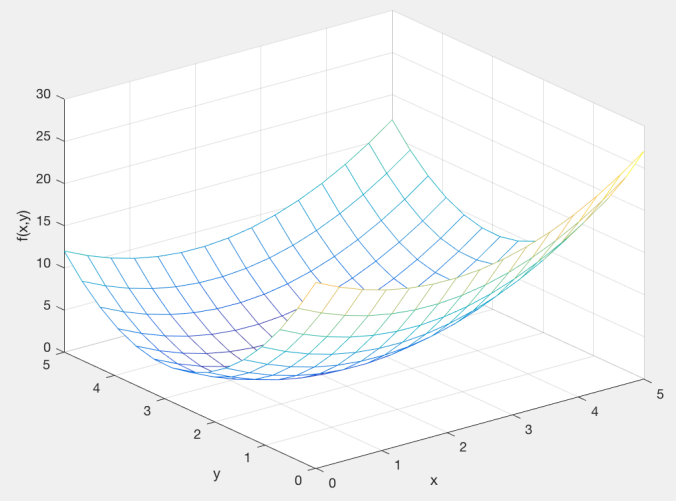

Figure 1: Visualization of the function $f(x, y) = (x - 2)^2 + 2(y - 3)^2$

(1bi) Code for the `graddesc` function, modified to produce a sequence of points as in Equation 1:

```
function soln = graddesc(f, g, i, e, tol)
% gradient descent
% f -- function
% g -- gradient
% i -- initial guess
% e -- step size
% tol -- tolerance
gi = feval(g,i) ; % evaluates initial gradient
seq = i; % sequence initialisation
while(norm(gi)>tol)  % crude termination condition
  i = i - e .* feval(g,i) ; % x(t+1) = x(t) - lambda*grad*f(x(t))
  gi = feval(g,i); % evaluates gradient at t+1
  seq = [seq;i]; % updates sequence of points
end
soln = seq; % matrix of final sequence of points (x - col 1, y - col 2)
end
```

(1bii) The following code is utilised to reproduce Figure 2, which visualises the path of the gradient descent algorithm in 3D. When calling the `graddesc` function, the step size is reduced to 0.01 to ensure the smoothness of the curve. The `graddesc` function returns a matrix of two columns; the first corresponds to the x-axis values of the points traversed and the second to the y-axis values. Each column is then massaged into the corresponding plotting function.

```
clear

% gradient descent function called
seq = graddesc('fc','dfc',[0,0],0.01,0.1);
x = seq(:,1); % x-axis values for points traversed
y = seq(:,2); % y-axis values for points traversed
fxy = fcarg(x,y); % corresponding f(x,y) value

plot3(x,y,fxy); % 3D plotting function
grid on  % displays mayor grid lines in plot
```

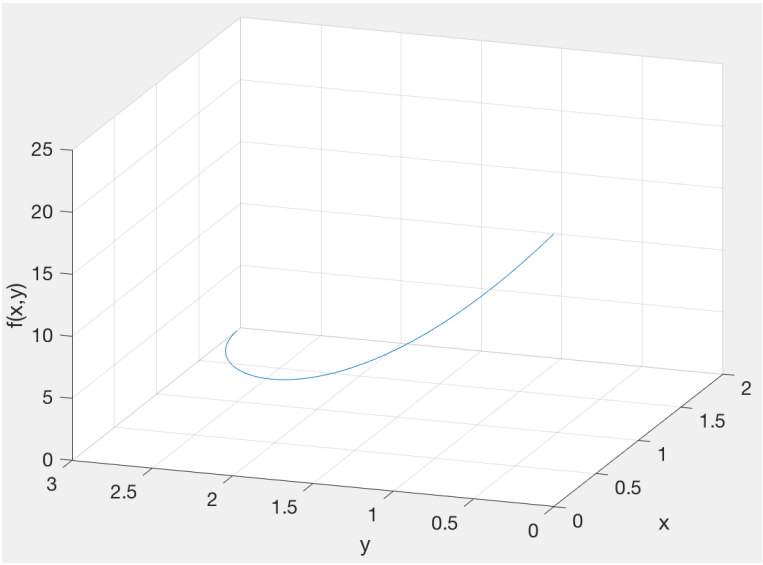

Figure 2: The gradient descent algorithm starts at (0,0), traversing a series of points towards the minima.

(1biii) The following code is used to reproduce Figure 3, which projects the path of the gradient descent algorithm down to the xy plane. The `graddesc` stepsize is again set to 0.01.

```
clear

% gradient descent function called
seq = graddesc('fc','dfc',[0,0],0.01,0.1);
x = seq(:,1); % x-axis values for points traversed
y = seq(:,2); % y-axis values for points traversed

plot(x,y) % 2D plotting function
```

(2a) Code for the `mydescent` function is presented in the next page. This function has been implemented to compute the least squares solution by gradient descent. It intakes 5 arguments: the matrix $\mathbf{A}$, the column vector $\mathbf{b}$, an initial guess `guess`, a step size `step` and a tolerance `tol`. It returns a matrix of two columns, `seq`, the first corresponding to the x_1-axis values of the points traversed and the second to the x_2 axis values. To determine the symbolic form of $\nabla_{\mathbf{x}}[(\mathbf{Ax}-\mathbf{b})^T(\mathbf{Ax}-\mathbf{b})]$, `grad`, note that e.g. for part **2b** (2 variables/cols and 3 training samples/rows):

$$\nabla_{\mathbf{x}}[(\mathbf{Ax}-\mathbf{b})^T(\mathbf{Ax}-\mathbf{b})] = \Big(\sum_{i=1}^{3}\frac{\partial}{\partial x_1}(a_{i1}x_1+a_{i2}x_2-b_i)^2, \sum_{i=1}^{3}\frac{\partial}{\partial x_2}(a_{i1}x_1+a_{i2}x_2-b_i)^2\Big).$$

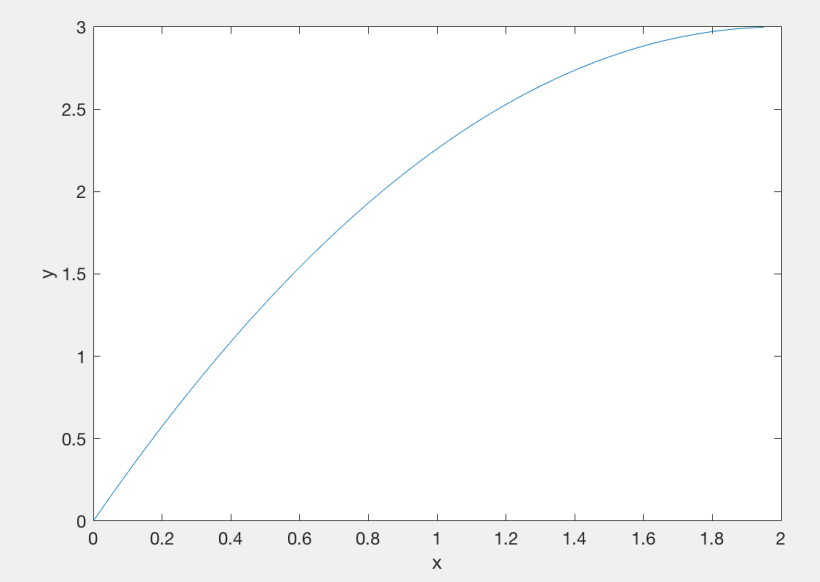

Figure 3: Question **(1biii)**. The path of the gradient descent algorithm projected on the xy plane.

It can be seen explicitly that taking the partial derivative over x_1, reduces an expression $a_{i1}x_1 + a_{i2}x_2 - b_i$ to a_{i1}. Similarly, taking the partial derivative over x_2 reduces the expression to a_{i2}. These values correspond to each column of $\mathbf{A}$. Then, each component of $\nabla_{\mathbf{x}}[\mathbf{e}^T\ \mathbf{e}]$ is symbolically ($\mathbf{Ax}$ - $\mathbf{b}$) multiplied element-wise by its corresponding column in $\mathbf{A}$, summing over all rows. In our MATLAB program, this operation is represented by the line `grad = [sum(e.*A(:,1)), sum(e.*A(:,2))]`. Gradient descent is performed until the norm of the gradient dips below a specified tolerance level.

```
function soln = mydescent(A, b, guess, step, tol)
% gradient descent for linear regression
% A - matrix
% b - column vector
% guess - initial guess
% step - step size
% tol - tolerance
e = A * guess - b;    % initial error calculation
grad = [sum(e.*A(:,1)), sum(e.*A(:,2))];  % initial gradient calculation
seq = guess'; % sequence initialisation
while(norm(grad)>tol)  % crude termination condition for gradient descent.
    guess = guess - step*grad'; % coordinates update
    seq = [seq; guess']; % updates sequence of points
    e = A * guess - b; % error evaluation at t+1
    grad  = [sum(e.*A(:,1)), sum(e.*A(:,2))]; % gradient evaluation at t+1
end
soln = seq; % matrix of final sequence of points
end
```

(2b) We call the implemented function `mydescent` to give a least squares solution to the system of linear equations. `step` is set to 0.01 and `tol` is set to 0.00001.

```
clear

% matrix/vector declarations
A = [1, -1; 1, 1; 1, 2];
b = [1;1;3];

% initial guess= [0,0], step = 0.01, tol = 0.00001
mydescent(A, b, [0;0],0.01,0.00001)
```

Gradient descent converges to $x_1 = 1.29$, $x_2 = 0.57$ (2 decimal points). The output of the code corresponding to the last 10 iterations is shown below.

```
1.2857    0.5714
1.2857    0.5714
1.2857    0.5714
1.2857    0.5714
1.2857    0.5714
1.2857    0.5714
1.2857    0.5714
1.2857    0.5714
1.2857    0.5714
1.2857    0.5714
```

(**2c**) The following code is used to reproduce Figure 4, which projects the path of the gradient descent algorithm down to the x_1x_2 plane.

```
clear

% matrix/vector declarations
A = [1, -1; 1, 1; 1, 2];
b = [1;1;3];

% initial guess= [0,0], step = 0.01, tol = 0.00001
seq = mydescent(A, b, [0;0],0.01,0.00001);
x1 = seq(:,1); % x-axis values for points traversed
x2 = seq(:,2); % y-axis values for points traversed

plot(x1,x2) % 2D plotting function
```

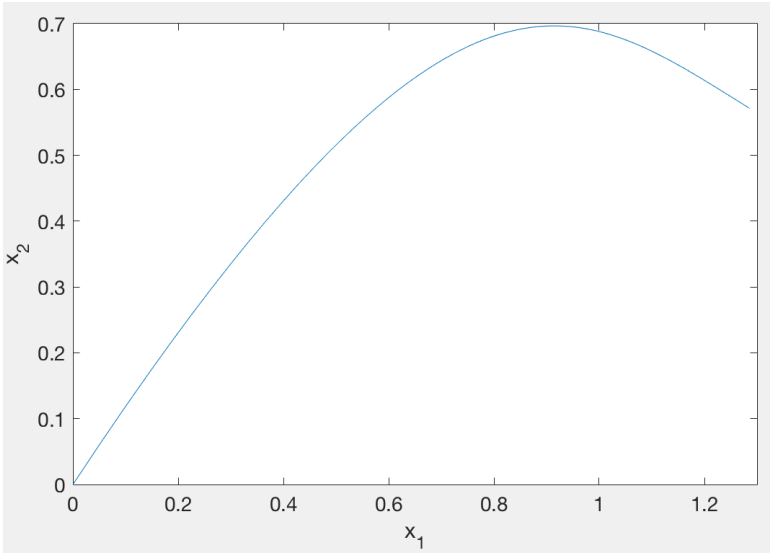

Figure 4: Question **(2c)**. The path of the gradient descent algorithm projected on the x_1x_2 plane.

(3a) Gradient descent nontrivially converges to 1. $f(x)$ is differentiable at 1.
(3b) Gradient descent does not converge converge to 1. $f(x)$ is non-differentiable at 1 but has a minima at 1.
(3c) Needs to be shown formally e.g. with a Cauchy convergence test it can be shown to converge to 1. $f'(x) = 4x^3 + 10x$

2 Linear Regression

(1a) For this question, two functions are implemented: `polynomialfit` and `polynomialeval`. `polynomialfit` is shown below . It intakes three variables: `X`, `Y` and `k`. `X` and `Y` correspond to the x-axis and y-axis values of the specified data points. k represents the basis dimension of the polynomial $p(X)$ (degree $k-1$) that will fit the data. The function outputs the array `pfit`, which includes the coefficients of the polynomial fit. To calculate these coefficients, we first construct a Vandermonde matrix, which is specified as:

$$\boldsymbol{\phi} = \begin{pmatrix} \phi(x_1) \\ \vdots \\ \phi(x_m) \end{pmatrix} = \begin{pmatrix} \phi_k(x_1) & \cdots & \phi_1(x_1) \\ \vdots & \ddots & \vdots \\ \phi_k(x_m) & \cdots & \phi_1(x_m) \end{pmatrix} = \begin{pmatrix} x_1^{k-1} & \cdots & 1 \\ \vdots & \ddots & \vdots \\ x_m^{k-1} & \cdots & 1 \end{pmatrix}.$$

To obtain a least squares solution for this system we can use the QR factorisation method. Built-in MATLAB function `qr`, intakes the Vandermonde matrix $\boldsymbol{\phi}$ and factorises it; $\boldsymbol{\phi} = QR$, where R is an upper triangular $m \times k$ matrix and Q is an orthogonal ($Q^TQ = 1$) unitary $m \times m$ matrix. For a normal system of equations, $\boldsymbol{\phi}^T\boldsymbol{\phi}\boldsymbol{\alpha} = \boldsymbol{\phi}^T\boldsymbol{y} \rightarrow \boldsymbol{R}^T\boldsymbol{Q}^T\boldsymbol{Q}\boldsymbol{R}\boldsymbol{\alpha} = \boldsymbol{R}^T\boldsymbol{Q}^T\boldsymbol{y} \rightarrow \boldsymbol{R}^T\boldsymbol{R}\boldsymbol{\alpha} = \boldsymbol{R}^T\boldsymbol{Q}^T\boldsymbol{y} \rightarrow \boldsymbol{R}\boldsymbol{\alpha} = \boldsymbol{Q}^T\boldsymbol{y} \rightarrow \boldsymbol{\alpha} = \frac{\boldsymbol{Q}^T\boldsymbol{y}}{\boldsymbol{R}} = \left(\frac{\boldsymbol{R}}{\boldsymbol{Q}^T\boldsymbol{y}}\right)^T$, where (α) are the parameters of the polynomial fit. In the code this calculation is represented by `pfit = (R \(Q' * Y(:)))'`.

```
function pfit = polynomialfit(X, Y, k)
% polynomialfit finds the coefficients of a polynomial p(X) of degree k-1 (basis
% dimension k) that fits the data
%
% X -> data point x-coords
% Y -> data point y-coords
% k - basis dimension
%
% construct Vandermonde matrix phi, a matrix of m rows and k columns, where
% phi(i,j) = phi_(k-j)(x_i) i.e. the last col is all ones (phi_1), the second last
% column represents X_1, X_2,...X_m, etc.
X = X(:);
phi = ones(length(X), k); % initialise Vandermonde matrix (m x k)
for j = k-1:-1:1  % update columns from last to first
    phi(:, j) = phi(:, j + 1) .* X; % recursively add a power of k
end
% solve least squares problem computing the QR factorisation, which partitions the
% Vandermonde matrix phi into Q and R. phi = Q*R where phi is mxn, Q is a unitary
% (mxm) matrix and R is a mxn matrix.
[Q, R] = qr(phi);
pfit = (R \ (Q' * Y(:)))'; %
end
```

The implemented function `polynomialeval` is shown below. This function intakes two variables: `P` and `X`. `P` is the variable output by our previous `polynomialfit` function: an array whose elements are the coefficients of a polynomial. `polynomialeval(P,X)` is the value of the polynomial evaluated at `X`.

```
function Y = polynomialeval(P,X)
% if P is an array whose elements are the coefficients of a polynomial,
% polynomialeval(P,X) is the value of the polynomial evaluated at X.
    len = length(X);    % number of x-coords
    Y = zeros(1, len);  % number of y-coords
    for i = 1:len       % calculations for each point
        N = length(P) - 1;   % max polynomial degree
        e = N:-1:0;
        for n = 1:N+1   % calculations for each polynomial degree
            Y(i) = Y(i) + P(n) * X(i).^e(n); % evaluate polynomial recursively
        end
    end
    Y;
end
```

We call the two functions in the code below, where plots for the curves corresponding to each polynomial basis fit (dimensions $k = 1, 2, 3, 4$) are produced. These plots are presented in Figure 5.

```
clear

X=[1,2,3,4];
Y=[3,2,0,5];

p1 = polynomialfit(X,Y,1);
p2 = polynomialfit(X,Y,2);
p3 = polynomialfit(X,Y,3);
p4 = polynomialfit(X,Y,4);

x = linspace(0,5,500);
y1 = polynomialeval(p1,x);
y2 = polynomialeval(p2,x);
y3 = polynomialeval(p3,x);
y4 = polynomialeval(p4,x);

plot(X,Y,'or')
axis([0,5,-4,8])
ax = gca;
ax.XAxisLocation = 'origin';

hold on
plot(x,y1,'--b','LineWidth',2)
plot(x,y2, ':k','LineWidth',2)
plot(x,y3, '-.m','LineWidth',2)
plot(x,y4, '-g','LineWidth',2)
hold off

legend('Given data','k=1','k=2','k=3','k=4','Location','eastoutside')
```

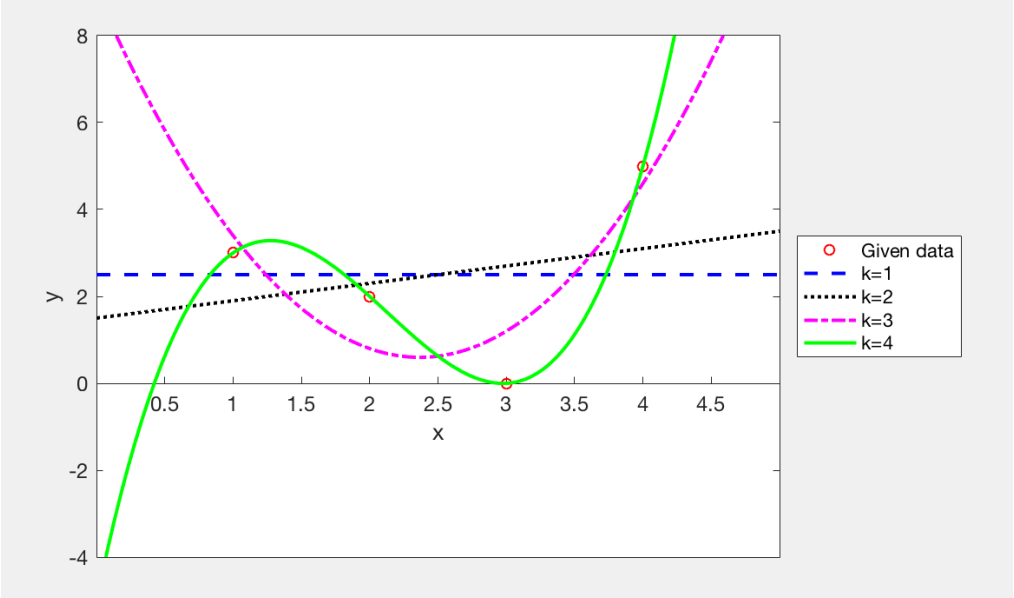

Figure 5: Polynomial fits of bases dimension $k = 1 - 4$, superimposed over the given data points

(1b) The equation corresponding to the curve fitted for $k = 1$ is $y = 2.5x$. For $k = 2$,

$$y = 0.4x + 1.5$$

For $k = 3$,

$$y = 1.5x^2 - 7.1x + 9$$

For $k = 4$,

$$y = 1.33x^3 - 8.5x^2 + 15.17x - 5$$

These equations are found by simply calling our `polynomialfit` function for different basis dimensions. Calling these functions returns the coefficient values as shown below:

```
>> X = [1,2,3,4];
>> Y = [3,2,0,5];
>> p1 = polynomialfit(X,Y,1)

p1 =

    2.5000

>> p2 = polynomialfit(X,Y,2)

p2 =

    0.4000    1.5000

>> p3 = polynomialfit(X,Y,3)

p3 =

    1.5000   -7.1000    9.0000

>> p4 = polynomialfit(X,Y,4)

p4 =

    1.3333   -8.5000   15.1667   -5.0000
```

(1c) To compute the mean square error, the following function, `mean_square_error` is implemented.

```
function MSE = mean_square_error(x, y)
% mean_square_error intakes two arrays x and y and calculates the MSE
% between them. both arrays can have any dimensions but must be of the same
% size and class.
MSE = (norm(x(:)-y(:),2).^2)/numel(x);  % SSE/m
```

The function intakes two arrays x and y and calculates the MSE ($= SSE/m$) between them. Both arrays can have any dimensions but must be of the same size and class. Calling this function as shown below, gives MSE $= 3.25$ for $k = 1$, MSE $= 3.05$ for $k = 2$, MSE $= 0.8$ for $k = 3$, MSE $= 4.65 \times 10^{-29}$ for $k = 4$.

```
>> X=[1,2,3,4]; Y = [3,2,0,5];
>> p1 = polynomialfit(X,Y,1);
>> p2 = polynomialfit(X,Y,2);
>> p3 = polynomialfit(X,Y,3);
>> p4 = polynomialfit(X,Y,4);
>> x = 1:4;
>> y1 = polynomialeval(p1,x);
>> y2 = polynomialeval(p2,x);
>> y3 = polynomialeval(p3,x);
>> y4 = polynomialeval(p4,x);
>> e1 = mean_square_error(y1,Y)

e1 =

    3.2500

>> e2 = mean_square_error(y2,Y)

e2 =

    3.0500

>> e3 = mean_square_error(y3,Y)

e3 =

    0.8000

>> e4 = mean_square_error(y4,Y)

e4 =

   4.6543e-29
```

(2a) The matlab function `randn` samples numbers with the distribution $N_{0,1}$ (a normal distribution with a mean of 0 and a standard deviation of 1). To generate random numbers with a $N_{\mu,\sigma}$ distribution, bare in mind the following: if a set of values is multiplied by a constant, its standard deviation is multiplied by such and its mean is invariant (all else kept equal). On the other hand, the addition of a constant does not affect the standard deviation but adds to the mean. Hence, a distribution $N_{\mu,\sigma}$ can be obtained by rescaling the output of the `randn` function, multiplying by σ and adding μ, as shown in the code below:

```
function numbers = randomnorm(mu, sigma, n)
% randomnorm.m returns a size n array of random numbers selected from a
% normal distribution with standard deviation sigma and mean mu.
i = 1;
while(i<n+1)   % while loop over the array setting random elements
  % rescaling operation
  numbers(i) = randn()* sigma + mu;
  i = i+1;
end
end
```

Calling our newly implemented `randomnorm` function with mean $\mu = 4$, standard deviation $\sigma = 2$ and outputting an array length 8 gives:

```
>> randomnorm(4,2,8)

ans =

    5.2375    3.9169    3.2895    7.0165    7.6156    0.1164    6.5403    3.2238
```

(2b) We define:

$$g_\sigma(x) = \sin^2(2\pi x) + N_{0,\sigma},$$

where $g_\sigma(x)$ is a *random* function such that $\sin^2(2\pi x)$ is computed and normal noise $N_{0,\sigma}$ is added on each function call. We implement the matlab function `gfunction`, code for which is shown below, to compute $g_\sigma(x)$. This function takes an array `x` and a value for standard deviation `sigma` as arguments. The random noise for each element of the array is computed invoking the `randomnorm` function, implemented in **(2a)**.

```
function result = gfunction(x,sigma)
% This function takes as input an array x and a standard deviation value
% sigma. sin^2(2*pi*x) is computed for each element of the array. Normal
% noise is added afterwards and the array result is outputted.
i = 1;
% noise values are selected from a normal distribution of mean 0, sd sigma.
noise = randomnorm(0,sigma,length(x));
% loop to compute resulting elements, updated with their respective noise.
while(i<length(x)+1)
  result(i) = (sin(2*pi*x(i)))^2 + noise(i);
  i = i+1;
end
end
```

Calling our newly implemented `gfunction` with an array length 5 and a standard deviation σ=1 as inputs yields:

```
>> gfunction([1,6,8,2,4],1)

ans =

    0.7147   -0.2050   -0.1241    1.4897    1.4090
```

(2bi) The function $\sin^2(2\pi x)$, along with the points of dataset $S_{0.07,30} = (x_1, g_{0.07}(x_1), ..., (x_{30}, g_{0.07}(x_{30})))$, is plotted in Figure 6. The code utilised for this purpose is below:

```
clear

% part i

% function plot
X1 = linspace(0,1,100); % x-values for sin plot
Y1 = (sin(2*pi*X1)).^2;  % sin function
% noisy dataset plot
X2 = linspace(0,1,30); % sample uniformly 30 times from [0,1]
Y2 = gfunction(X2,0.07); % g_0.07 applied to each element of X2

plot(X1,Y1,'b') % sin function
ax = gca;
ax.XAxisLocation = 'origin';
hold on
plot(X2,Y2,'or') % noisy dataset superimposed
hold off
```

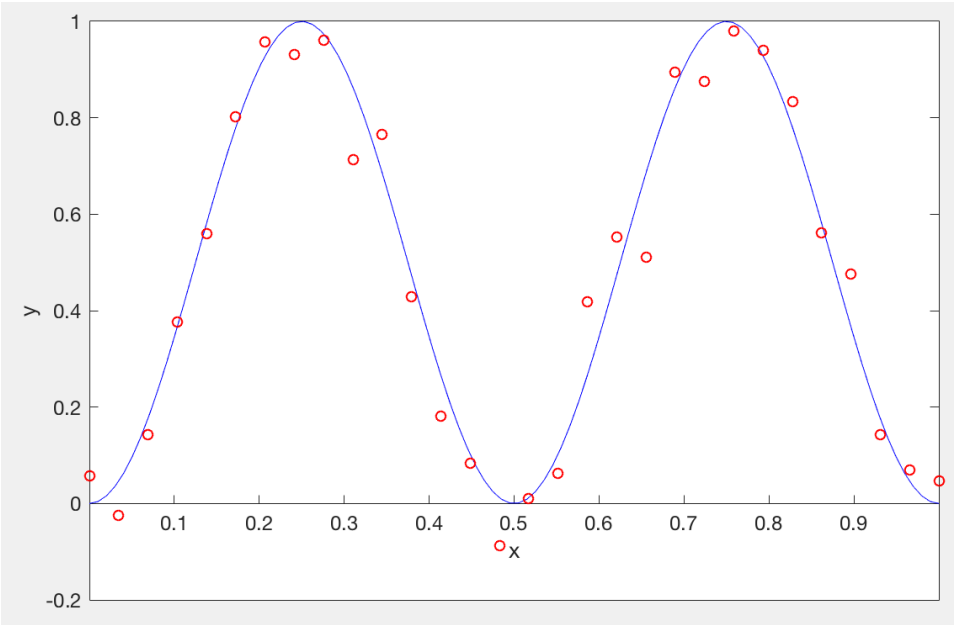

Figure 6: The function $\sin^2(2\pi x)$ in the range $0 \leq x \leq 1$ with points of dataset $S_{0.07,30}$ superimposed.

(2bii) The dataset $S_{0.07,30}$ is fit to polynomial bases of dimension $k = 2, 5, 10, 14, 18$, using the code below:

```
clear

% for noisy dataset plot
X1 = linspace(0,1,30); % sample uniformly 30 times from [0,1]
Y1 = gfunction(X1,0.07); % g_0.07 applied to each element of X2

% coefficients returned for polynomial fits of noisy dataset
P2 = polynomialfit(X1,Y1,2);
P5 = polynomialfit(X1,Y1,5);
P10 = polynomialfit(X1,Y1,10);
P14 = polynomialfit(X1,Y1,14);
P18 = polynomialfit(X1,Y1,18);

% for polynomial fits plot
XP = linspace(0,1,100);
Y2 = polynomialeval(P2,XP);
Y5 = polynomialeval(P5,XP);
Y10 = polynomialeval(P10,XP);
Y14 = polynomialeval(P14,XP);
Y18 = polynomialeval(P18,XP);
```

```
21
22 -    figure
23 -    plot(X1,Y1,'or') % noisy dataset plot
24 -    ax = gca;
25 -    ax.XAxisLocation = 'origin';
26 -    hold on
27 -    plot(XP,Y2,'--+c','LineWidth',0.4,'MarkerSize',2) % k=2 polynomial fit plot
28 -    plot(XP,Y5,'--b','LineWidth',2) % k=5
29 -    plot(XP,Y10, ':k','LineWidth',2) % k=10
30 -    plot(XP,Y14, '-.m','LineWidth',2) % k=14
31 -    plot(XP,Y18, '-g','LineWidth',1.5) % k=18
32 -    hold off
33
34 -    legend('S_{0.07,30}','k=2','k=5','k=10','k=14','k=18','Location','eastoutside')
```

This program outputs the graph in Figure 7.

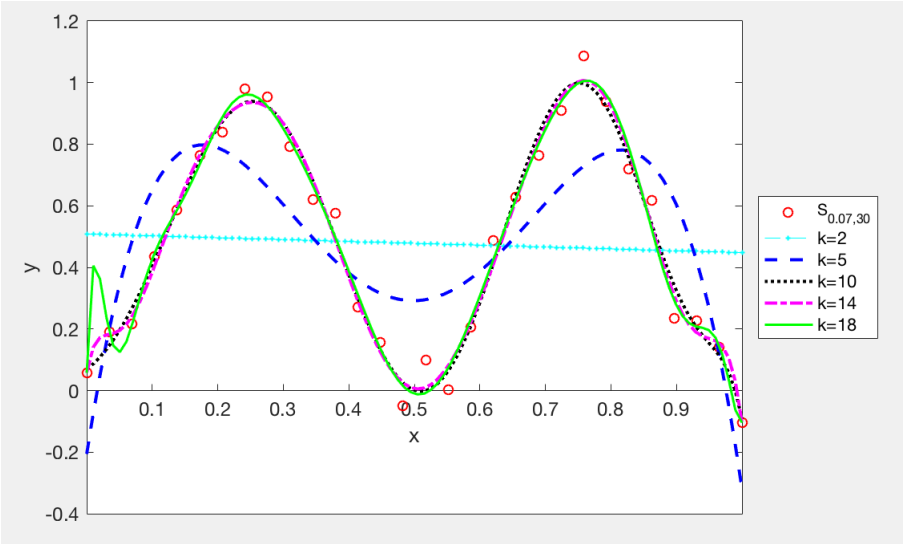

Figure 7: Curves corresponding to polynomial fits $k = 2, 5, 10, 14, 18$ superimposed over a dataset $S_{0.07,30}$.

(2c) The following code is written to plot the log of the training error versus the polynomial dimension $k = 1, ..., 18$.

```
1 -    clear
2
3 -    sample_no = 30; % number of samples from [0,1]
4 -    poly_no = 18;  % number of polynomial fits
5 -    st_dev = 0.07; % standard deviation
6
7      % for noisy dataset plot
8 -    X = linspace(0,1,sample_no); % sample uniformly
9 -    Y = gfunction(X,st_dev); % g_0.07 applied to each element of X1
10 -   P = zeros(poly_no, sample_no); % matrix storing y-coords of polynomial fits
11 -   E = zeros(1, poly_no); % array containing MSE for each fit
12
13 -   i=1;
14     % loop to store coefficients for each polynomial fit in a cell array coeffs
15 -   while(i<poly_no+1)
16 -     coeffs{i} = polynomialfit(X,Y,i); % coefficients of each fit stored here
17 -     P(i,:) = polynomialeval(coeffs{i}, X); % P row (each row is a different fit) update
18 -     E(i) = mean_square_error(P(i,:),Y); % errors stored here using immse function
19 -     i = i+1;
20 -   end
21
22     % plot
23 -   basis_dim = 1:poly_no;
24 -   plot(basis_dim, log(E), 'r', 'Linewidth', 1.5) % plot dim
25 -   ax = gca;
26 -   ax.XAxisLocation = 'origin';
```

As seen above, matrix `P`, with 18 rows (each representing a different basis dimension) and 30 columns

(each representing a sampled x-axis point), is initialised. So is the array `E`, which has 18 elements, each corresponding to the MSE at a given basis set dimension. `M` and `E` are updated using a `while` loop. The training error is calculated using the `mean_square_error` function, which returns the MSE between two arrays that must be of the same size (we sample 30 points for both the noisy dataset and the polynomial fits). Calling the `plot` function yields the graph presented in Figure 8.

Figure 8: The natural logarithm of the training error plotted versus the polynomial basis dimension.

Indeed, Figure 8 shows a decreasing value of the training error with increasing basis dimensionality.

(2d) The following code is written to plot the log of the test error versus the polynomial dimension $k = 1, ..., 18$.

```
clear
st_dev = 0.07; % standard deviation
poly_no = 18;  % number of polynomial fits
sample_no_train = 30; % number of samples from [0,1] for train set
sample_no_test = 1000; % number of samples from [0,1] for test set

% for train set
X_train = linspace(0,1,sample_no_train); % sample uniformly
Y_train = gfunction(X_train,st_dev); % g_0.07 applied to each element of X_train
P_train = zeros(poly_no, sample_no_train); % matrix storing polynomial train coords
E_train = zeros(1, poly_no); % array containing MSE for each train fit
% for test set
X_test = linspace(0,1,sample_no_test); % sample uniformly
Y_test = gfunction(X_test,st_dev); % g_0.07 applied to each element of X_test
P_test = zeros(poly_no, sample_no_test); % matrix storing polynomial test coords
E_test = zeros(1, poly_no); % array containing MSE for each test fit

i=1;
% loop to store coefficients for each fit in a cell array coeffs_train
while(i<poly_no+1)
  coeffs_train{i} = polynomialfit(X_train,Y_train,i); % poly coefficients stored here
  P_train(i,:) = polynomialeval(coeffs_train{i}, X_train); % P_train row update
  P_test(i,:) = polynomialeval(coeffs_train{i}, X_test); % P_train row update
  E_train(i) = mean_square_error(P_train(i,:),Y_train); % train error
  E_test(i) = mean_square_error(P_test(i,:),Y_test); % train error
  i = i+1;
end

% plot
basis_dim = 1:poly_no;
plot(basis_dim, log(E_test), 'r', 'Linewidth', 1.5) % plot dim
ax = gca;
ax.XAxisLocation = 'origin';
```

Note how the test set T is fitted using the coefficients of the polynomial fit of training set S (`P_test(i,:) = polynomialeval(coeffs_train(i), X_test)`). Figure 9 shows the test error $\text{tse}_k\ (S,T)$ plotted against the polynomial basis fit dimension k .

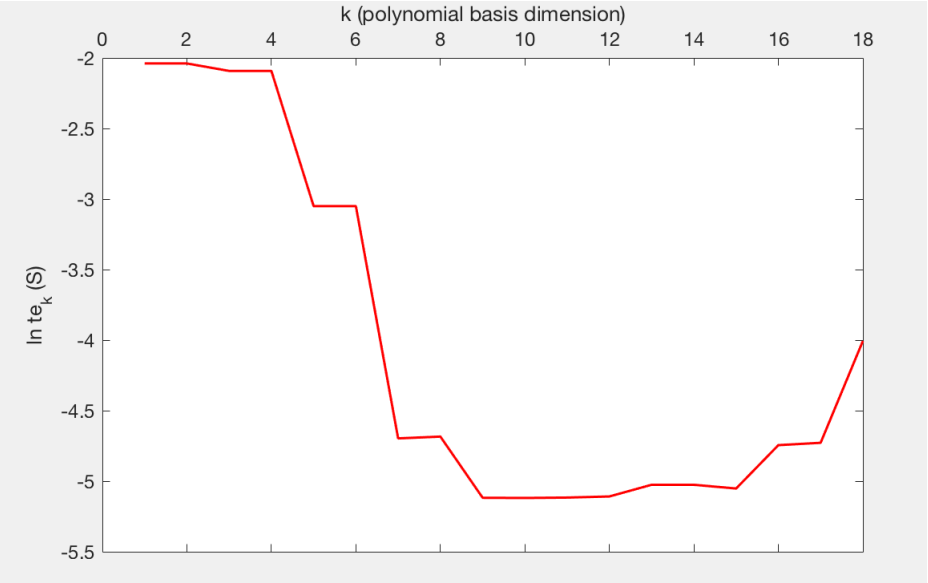

Figure 9: The natural logarithm of the test error plotted versus the polynomial basis dimension

Effectively, unlike the training error , this is not a decreasing function. There has been overfitting: the test error eventually increases since rather than fitting the function, in a loose sense, we begin to fit the noise.

(2e) The code below is used to plot the natural logarithm of the average train and test error for 100 runs.

```
clear

st_dev = 0.07; % standard deviation
poly_no = 18;  % number of polynomial fits
sample_no_train = 30; % number of samples from [0,1] for train set
sample_no_test = 1000; % number of samples from [0,1] for test set
runs = 100; % 100 runs

% for train set
X_train = linspace(0,1, sample_no_train); % sample uniformly
P_train = zeros(poly_no, sample_no_train); % matrix storing polynomial train coords
E_train = zeros(1, poly_no); % array containing MSE for each train fit
E_tot_train = zeros(1, poly_no); % total err for each train fit for avg calculation

% for test set
X_test = linspace(0,1, sample_no_test); % sample uniformly
P_test = zeros(poly_no, sample_no_test); % matrix storing polynomial train coords
E_test = zeros(1, poly_no); % array containing MSE for each train fit
E_tot_test = zeros(1, poly_no); % total err for each test fit for avg calculation

m = 0;
while (m<runs)
    Y_train = gfunction(X_train, st_dev); % g_0.07 applied to each element of X_train
    Y_test = gfunction(X_test, st_dev); % g_0.07 applied to each element of X_test
    i=1;
    while(i<poly_no+1)
        coeffs_train{i} = polynomialfit(X_train,Y_train,i);
        P_train(i,:) = polynomialeval(coeffs_train{i}, X_train); % P_train row update
        P_test(i,:) = polynomialeval(coeffs_train{i}, X_test); % P_test row update
        E_train(i) = mean_square_error(P_train(i,:),Y_train); % train error
        E_test(i) = mean_square_error(P_test(i,:),Y_test); % test error
        E_tot_train(i) = E_tot_train(i) + E_train(i); % total train err for avg
        E_tot_test(i) = E_tot_test(i) + E_test(i); % total test err for avg
        i = i+1;
    end
    m = m+1;
end

E_avg_train = E_tot_train/runs;
E_avg_test = E_tot_test/runs;

% plot
basis_dim = 1:poly_no;
plot(basis_dim, log(E_avg_train), 'r', 'Linewidth', 1.5) % plot dim
ax = gca;
ax.XAxisLocation = 'origin';
hold on
plot(basis_dim, log(E_avg_test), '--b', 'Linewidth', 1.5) % plot dim
hold off
legend('ln te_k(S)','ln tse_k(S,T)','Location','eastoutside')
```

Figure 10 shows the logarithm of the average of the train error and the test error (dashed) for 100 runs plotted against the polynomial basis dimension k. Effectively, the curves are now more smoothed out.

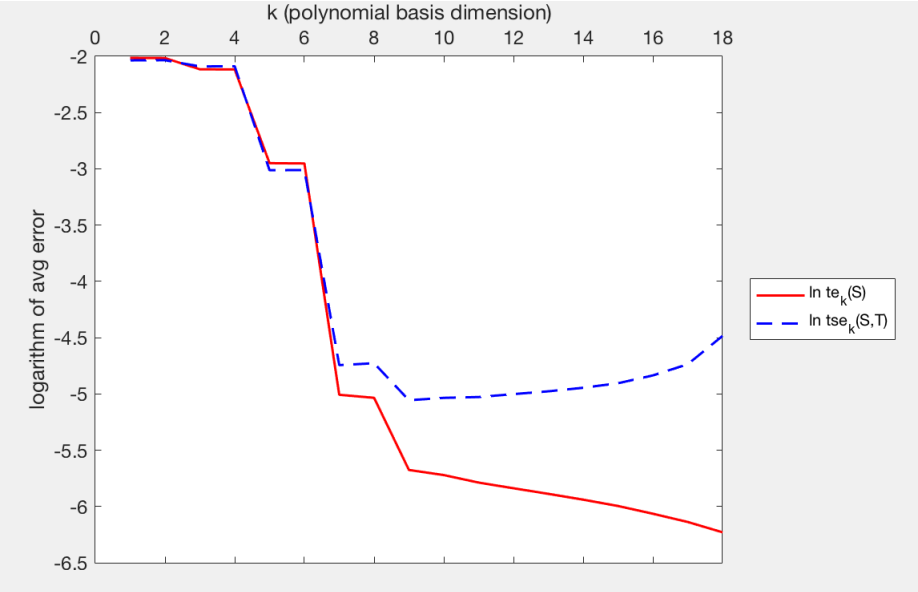

Figure 10: Natural logarithm of the avg. train and test (dashed) error plotted versus k for 100 runs.

(3) The function `sinfit` is implemented to fit our data points using a basis $\sin(1\pi x), \sin(2\pi x), \sin(3\pi x), ..., \sin(k\pi x)$. `sinfit` returns the coefficients of the fit (in array `s`) and is shown below.

```
function [s A] = sinfit( x, y, k )

for i = 1:k
    A(:,i) = sin(i*pi*x);
end
s = A\y;
```

These coefficients are evaluated by the function `sineval`, which takes as inputs a matrix of coefficients `S` and input x-values `X` and returns predicted values `Y`. `sineval` is shown below.

```
function Y = sineval(S,X)
    size_s = size(S);   % coefficient size
    size_x = size(X);   % size of input x-coords
    Y = zeros(size(X));   % initialise matrix for predicted Y
    for i =1:size_x  % loop over rows
        for j=1:size_s % loop over columns
            Y(1,i) = Y(1, i) + S(j,1)*sin(j*pi*x(i));
        end
    end
end
```

Then steps (c) to (e) can be repeated, substituting `polynomialfit` for `sinfit` and `polynomialeval` for `sinval`.

(3) Whac-a-mole is a one player game played on a square $n \times n$ grid of holes. Each hole can be in one of two states: either with a mole popping up or with a mole hiding. A move consists of "whacking" a mole in a hole, thereby toggling the states of such hole and its vertically and horizontally adjacent holes. This can be formulated as the following algebraic problem:

1. Each hole configuration can be represented as a matrix $\mathbf{M}$ with entries in $\mathbb{Z}_2$ (i.e. a (0,1) matrix where each entry denotes the state of a hole; 1 when a mole is present and 0 if it hides). For example, the initial 4×4 game configuration provided by the question gives,

$$\mathbf{M} = \begin{bmatrix} m_{13} & m_{14} & m_{15} & m_{16} \\ m_9 & m_{10} & m_{11} & m_{12} \\ m_5 & m_6 & m_7 & m_8 \\ m_1 & m_2 & m_3 & m_4 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 \end{bmatrix},$$

   where m_i are the components of the matrix, with positions indexed as per the question.

2. The action of 'whacking' a mole at i can be modelled as the operation $\mathbf{M} + \mathbf{A}_i$, where $\mathbf{A}_i$ is the matrix in which the only entries equal to one are those at i and its adjacent positions, vertically and horizontally. There are only three possible configuration types for a matrix $\mathbf{A}_i$ and these depend on the position of i relative to the grid:

   i i is in the "middle" e.g. $i = 6, 7, 10$ or 11 . In this case, for $i = 10$,

$$\mathbf{A}_{10} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}.$$

   ii i is in a "side" e.g. $i = 2, 3, 5, 8, 9, 12, 14, 15$. In this case, for $i = 2$,

$$\mathbf{A}_2 = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}.$$

   iii i is in a corner e.g. $i = 1, 4, 13, 16$. In this case, for $i = 4$,

$$\mathbf{A}_4 = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix}.$$

3. The reason these observations are interesting are that they give us a clear mathematical framework to solve the question. In such framework, a winning combination of moves is formulated as:

$$\mathbf{M} + \sum_i \alpha_i \mathbf{A}_i = \mathbf{0} \qquad \text{(modulo 2)},$$

   where the zero matrix represents a state where all moles are hiding and α_i denotes the number of times a mole/hole is "whacked" or if it has been "whacked" or not (in modulo 2 "whacking" a hole 2, 4, 6... times gives $\alpha_i = 0$). Since the matrices have entries in $\mathbb{Z}_2$, the above expression is equivalent to $\mathbf{M} = \sum_i \alpha_i \mathbf{A}_i$. Note that from the commutative property of matrix addition, the order of the move combination is irrelevant.

4. Now lets change our notation a bit. Matrices $\mathbf{M}$ and $\mathbf{A}_i$ can be converted into column vectors by configuring their values into row-major order. e.g, for $i = 10$:

$$\mathbf{A}_{10} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} = [0\ 1\ 0\ 0\ 1\ 1\ 1\ 0\ 0\ 1\ 0\ 0\ 0\ 0\ 0\ 0]^T,$$

Hence, $\sum_i \mathbf{A}_i \alpha_i = \mathbf{M}$ is just a linear system of equations over a set of vectors, representing different grid patterns. For example, the system corresponding to the initial pattern in the question is the following:

$$\begin{bmatrix} 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \alpha_{13} \\ \alpha_{14} \\ \alpha_{15} \\ \alpha_{16} \\ \alpha_9 \\ \alpha_{10} \\ \alpha_{11} \\ \alpha_{12} \\ \alpha_5 \\ \alpha_6 \\ \alpha_7 \\ \alpha_8 \\ \alpha_1 \\ \alpha_2 \\ \alpha_3 \\ \alpha_4 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 1 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 1 \\ 0 \\ 1 \end{bmatrix}.$$

Our goal is clear: to find a sequence of holes that one can hit in order to empty the board, one must solve this system of linear equations. For this purpose, standard Gaussian elimination can be used to solve for a value of $\alpha = \sum_i \alpha_i$. In the example provided above, the square 16×16 matrix corresponding to $\sum_i \mathbf{A}_i$ has maximal rank (since it also has a non-zero determinant). Hence Whac-A-Mole! for this 4×4 grid is always solvable. This can be extended to another square $n \times n$ grid.

5. **Gaussian elimination:**

Now that we have a toggle matrix $\mathbf{A}$ and an initial configuration $\mathbf{M}$, we can append $\mathbf{M}$ to $\mathbf{A}$, arriving at a 16×17 matrix. We can now proceed to solve this linear system; our aim is to reduce it to its row-reduced echelon form in gf(2) (Galois field of two elements). The code below, corresponding to the implemented function `solve_appended` illustrates this procedure. The comments should be fairly explanatory.

```
function [AM] = solve_appended(AM)
% The function solve_appended solves the linear system of equations
% specified by the augmented matrix (A|M), where A is the toggle matrix of
% the grid and M is a column vector specifying the game's initial
% configuration. The function intakes the augmented matrix (A | M) and outputs
% its row-reduced echelon form in gf(2). The output matrix corresponds to the
% positions of the grid you have to whack to solve the problem
[m,n] = size(AM);
i = 1;
j = 1;
while (i <= m) && (j <= n) % Perform a while loop over the rows/cols of (A|M)
   % We will illustrate the steps for i=1, j=1.
   %
   % First step:
   % We know that the element (1,1) is 1 from the configuration of our
   % toggle matrix. We proceed to find the second lowest entry (furthest up)
   % in the first column which is equal to 1. This will be s - our pivot.
   s = i - 1 + find(AM(i:m,j),1);
   % Second step: We proceed to swap the position of the row corresponding to
   % s - for our toggle matrix this is the 2nd row -  with the first row (i=1).
   AM([i s],j:n) = AM([s i],j:n);
   % variable right_of_pivot is made up of the variables in the sth row to
   % the right of pivot s.
   right_of_pivot = AM(i,j:n);
   % variable current_column consists of the jth column (in this case
   % column 1)
   current_column = AM(1:m,j);
```

```
28        % our objective is to obtain a matrix "turn", that will indicate with ones, which
29        % entries of the matrix need to change status to become zeroes. we
30        % obtain the matrix turn by multiplying current_column*right_of_pivot,
31        % leaving all entries to the right of s in row s  equal to zero. also
32        % leaving the entries of the jth column except that of s equal to zero.
33        % we will have to set the pivot to zero in this operation so that it is not
34        % flipped and is kept at one. (turn(1,1) will equal 1 and (1,1) is
35        % switched from 0 to 1.
36 -      current_column(i) = 0;
37 -      turn = current_column*right_of_pivot;
38        % XOR pivot RHS with all other rows
39 -      AM(1:m,j:n) = xor( AM(1:m,j:n), turn);
40 -      i = i + 1;  % repeat for all rows/columns
41 -      j = j + 1;
42 -    end
43
44      % We will eventually arrive at an augmented matrix (A | M) where A is
45      % diagonal. M will indicate the value of its corresponding diagonal matrix,
46      % 1 or 0. 1 indicates that this hole needs to be whacked to empty the grid,
47      % 0 the contrary.
```

This procedure is made possible by the toggle matrix being symmetric. Also note that we use a XOR gate to update the matrix since in GF(2) addition between two entries is equivalent to an exclusive OR gate. Once we have looped over all/rows columns, we have a matrix in reduced-row echelon form. This diagonal matrix will have final column values of either 1 or 0. 1 means that its corresponding position in the grid will have to be "whacked" to empty the board. We call the `solve_appended` function below.

```
1 -   clear
2     %
3     % Whac-A-Mole! Solver
4     %
5     % Intaking a matrix representation of a Whac-a-Mole board configuration, this
6     % program finds the sequence of holes that need to be hit in order to empty
7     % the board. This implementation makes use of Gaussian Elimination, which solves
8     % the puzzle but with the possibility of many redundant steps.
9     %
10    % user specified number of rows/columns in the grid
11 -  n = 3;
12 -  grid = randi([0 1],n,n)
13    %
14 -  tog_rownum = n*n;
15 -  tog_mat = toggle_maker(n);
16 -  grid = grid';
17 -  grid = (reshape(grid, 1, tog_rownum))';
18 -  appended = [tog_mat, grid];
19
20 -  sol = solve_appended(appended)
21 -  result = sol(:,((n*n)+1));
22 -  result = (reshape(result, 3,3))'
```

`randi` initialises an initial grid of randomly assignated 0 and 1 entries. Note how the column vector corresponding to the solution is re-converted into a square matrix indicating visually which positions should be whacked to empty the board. Our Whac-a-Mole! solver makes use of one more function. This is `toggle_maker`. `toggle_maker` generates an $(n^2) \times (n^2)$ toggle matrix. The function is presented below.

```
function [tog_mat] = toggle_maker(n)
%
%
tog_mat = zeros(n*n); % nrow/ncol of A = n*n
  for i=1:n     % each M element has an associated n*n matrix
    for j=1:n
      tog_row = n * (i - 1) + j; % row in A corresponding to each hole
      tog_mat(tog_row, tog_row) = 1; % whacked hole is toggled
      % whacking hole not in top row toggles a hole below it
      if (i > 1)
        tog_mat(tog_row, tog_row - n) = 1;
      end
      % whacking hole not in bottom row toggles a hole above it
      if (i < n)
        tog_mat(tog_row, tog_row + n) = 1;
      end
      % whacking hole not in leftmost row toggles the hole to its left
      if (j > 1)
        tog_mat(tog_row, tog_row - 1) = 1;
      end
      % whacking hole not in rightmost row toggles the hole to its right
      if (j < n)
        tog_mat(tog_row, tog_row + 1) = 1;
      end
    end
  end
end
```

When the Whac-a-Mole! solver is called from the command line, the following matrices appear (for a square 3×3 initial grid:

```
>> Exercise2_4

grid =

     0     0     0
     0     0     1
     1     1     0


sol =

     1     0     0     0     0     0     0     0     0     1
     0     1     0     0     0     0     0     0     0     0
     0     0     1     0     0     0     0     0     0     1
     0     0     0     1     0     0     0     0     0     1
     0     0     0     0     1     0     0     0     0     0
     0     0     0     0     0     1     0     0     0     1
     0     0     0     0     0     0     1     0     0     0
     0     0     0     0     0     0     0     1     0     0
     0     0     0     0     0     0     0     0     1     1


result =

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

`grid` represents the grid's initial configuration. `sol` represents the augmented solution to the `solve_appended` function. `result` corresponds to the final column vector of `sol`, re-shaped into a square matrix. This matrix shows the moles/holes that must be whacked to empty the board. Using pen and paper, the answer has been shown to be correct.

Complexity: The algorithm is polynomial in n.
Proof: The first thing that needs to be computed in our program is the toggle matrix for the different positions in the grid. There are $O(n^2)$ toggle "components", an $n \times n$ matrix for each position. Filling up **each** of these components requires $O(n^2)$ runtime. Hence, only this step gives $O(n^2 \times n^2) = O(n^4)$. After this step, within our solve_appended procedure, we must firstly find a column j and secondly, empty each other column. This will possibly require inspecting all the $O(n^2 \times n^2)$ entries of our toggle matrix, increasing the complexity of these steps to $O(n^6)$. These steps dominate most of our computation so the complexity is $O(n^6)$. Note however, that since the input matrix to solve the problem is a $(n^2 \times n^2)$ matrix, the procedure can have a runtime proportional to $O(n^3)$.