APPENDIX I Specifications of CMMs

1. CMM available at JASH Precision Tools, Indore, Model: 665N, Make: JASH 3D CMM
   a. Measuring Range: X – 600 mm, Y – 600 mm, Z – 500 mm
   b. Axes Movement: Servo Control
   c. Resolution (μm): 0.5
   d. Maximum permissible error: 2.0 + L/300 (where, L in mm)
   e. Drive Method: Air bearing on each axis
   f. Maximum Velocity (mm/s): 400
   g. Air Requirements (kg/cm2, NL/min): 5, 35
   h. Maximum work piece weight (kg): 1000
   i. Machine Weight (kg): 1000

2. CMM available at BVM Engineering College, Model:
   a. Measuring Range: X – 500 mm, Y – 700 mm, Z – 500 mm
   b. Axes Movement: Servo Control
   c. Resolution (μm): 0.1
   d. Accuracy: As per ISO 10360.2
   e. Maximum permissible error: 2.0 + 3.3L/300
   f. Guide Method: Air bearing on each axis
   g. Drive Speed (mm/s): 300
   h. Acceleration (mm/s²): 400
   i. Measuring Table: Granite
   j. Software: PC DMIS CAD 4.3 MRI
   k. Probing System: Motorized Indexable Probe head with tough trigger
APPENDIX II MATLAB Programs

1(a) A MATLAB Program to evaluate the straightness error using Maximum Distance Point Strategy.

% This program uses the MDFS to fit set of data points in a line.
% The data points should be stored in comma separated value file
% (csv). The first column in csv file represents x coordinate,
% second column represents y coordinate and third column represents
% of the point. This file is read by the program.

clear all
close all
clc

% Prompts for selecting file
prompt = 'Enter filename to fit circle (*.csv): ';
str = input(prompt, 's');

% The data points are read in matrix P
P=csvread(str);

Time_Start = tic;  % Start clock to measure performance

nop = size((P), 1);  % Number of points in set

% Initialize pair matrix, size(n x 2)
pair = zeros(nop, 2);

% In each pair the first point is fixed by selecting each point once
for row=1:nop
    pair(row,1)=row;
end

% Initialize Distance matrix, size(n x n)
Dist_m = zeros(nop, nop);

% Calculates eculidean distance between two points
for count=1:nop
    for row=1:nop
        Dist_m(row, count)=((P(count,1)-P(row,1))^2 + ...
                         (P(count,2)-P(row,2))^2 + (P(count,3)-P(row,3))^2);
    end
end

% finds the indices of the maximum values of each column of matrix
% Dist_m and returns them in output vector I. If there are several
% identical maximum values, the index of the first one found
% is returned. So matrix 'I' is row matrix (size 1 x n) and
% represents index of each maximum value in the column
[C, I] = max(Dist_m);

% Second point in pair is selected based on index value present in
% matrix 'I'. Distance for selected point is set to zero to avoid
% it's selection as third point in the triplet.
for count=1:nop
    pair(count,2)=I(count);
Dist_m(I(count), count) = 0;
end

Normal = zeros(nop, 2);
Normal_RHS = zeros(nop, 1);
direction = zeros(nop, 3);

% Calculates direction cosines of normal to the line passing
% through three selected point
for count = 1:nop
    % First point in triplet
    x1 = P(pair(count, 1), 1);
y1 = P(pair(count, 1), 2);
    z1 = P(pair(count, 1), 3);

    % Second point in triplet
    x2 = P(pair(count, 2), 1);
y2 = P(pair(count, 2), 2);
    z2 = P(pair(count, 2), 3);

direction(count, :) = [x2-x1 y2-y1 z2-z1];
if dot(direction(1,:), direction(count,:)) < 0
    direction(count, :) = -direction(count,:);
end
end

% Initialize matrix for sum of square of error for each line
% (size: n x 1)
sq_err_m = zeros(nop, 1);
ave_P = mean(P);

% Calculate sum of square of error for each line
for row = 1:nop
    for count = 1:size(P, 1)
        temp = (P(count,:) - ave_P) - (dot(direction(row,:),
                                           (P(count,:) - ave_P))/
                                           dot(direction(row,:),
                                           direction(row,:))) *
                                           direction(row,:);
        AMD = dot(temp, temp);
        sum_sq_err_m(row, 1) = sum_sq_err_m(row, 1) + AMD^2;
    end
end

MAD = mean(sum_sq_err_m);
Avg_direction = mean(direction);
Prev_direction = zeros(1, 3);

% Initialize matrix Di, size (n x 1). Stores value of normal
% distance from line to each point
Di = zeros(nop, 1);
sq_err_avg = 0; % sum of squared error from found average line

for count = 1:size(P, 1)
    temp = (P(count,:) - ave_P) - (dot(Avg_direction, P(count,:))
                                           - ave_P)/
                                           dot(Avg_direction, Avg_direction)) *
                                           Avg_direction;
    Di(count, 1) = dot(temp, temp);
end

% Calculate straightness and store as previous
Prev_Straightness = abs(max(Di)) - abs(min(Di));

% Store average line as previous line
Prev_direction(1,1) = vpa(Avg_direction(1,1), 6);
Prev_direction(1,2) = vpa(Avg_direction(1,2), 6);
Prev_direction(1,3) = vpa(Avg_direction(1,3), 6);

% Follow heuristic algorithm discussed in 4.3.1
while(1)
    k = 0; % Count number of selected line
    m = 0;
    Sum_A = 0;
    Sum_Direction=zeros(1,4);
    for count = 1:nop
        if sum_sq_err_m(count,1) <= MAD
            Sum_Direction(1,1) =
                vpa(Sum_Direction(1,1)+direction(count,1), 6);
            Sum_Direction(1,2) =
                vpa(Sum_Direction(1,2)+direction(count,2), 6);
            Sum_Direction(1,3) =
                vpa(Sum_Direction(1,3)+direction(count,3), 6);
            k = k+1;
            A(count,1) = sum_sq_err_m(count,1);
        else
            Sum_Direction(1,1) = Sum_Direction(1,1);
            Sum_Direction(1,2) = Sum_Direction(1,2);
            Sum_Direction(1,3) = Sum_Direction(1,3);
            sum_sq_err_m(count,1) = MAD;
            A(count,1) = 0;
        end
    end
    for count = 1:nop
        if A(count,1)>0
            m = m+1;
            Sum_A = Sum_A+A(count,1);
        end
    end
    MAD = Sum_A/m;
    Avg_Direction = zeros(1,4);
    Avg_Direction(1,1) = vpa(Sum_Direction(1,1)/k, 6);
    Avg_Direction(1,2) = vpa(Sum_Direction(1,2)/k, 6);
    Avg_Direction(1,3) = vpa(Sum_Direction(1,3)/k, 6);

    % sum of squared error from found average line
    sum_sq_err_avg = 0;
    for count = 1:size(P,1)
        temp = (P(count,:)-ave_P) - (dot(Avg_direction,
            (P(count,:)-ave_P))/(dot(Avg_direction,
            Avg_direction))) * Avg_direction;
        Di(count, 1) = dot(temp, temp);
    end
    Straightness = abs(max(Di))+ abs(min(Di));
    if Prev_Straightness <= Straightness
        break
    else
        Prev_Straightness = Straightness;
        Prev_direction(1,1) = Avg_Direction(1,1);
    end
Prev_direction(1,2) = Avg_Direction(1,2);
Prev_direction(1,3) = Avg_Direction(1,3);
end

sum_sq_err = 0;            % Sum of Square of errors (di^2)
for count = 1:size(P,1)
    temp = (P(count,:) - ave_P) - (dot(Avg_direction, (P(count,:)
    - ave_P))/(dot(Avg_direction, Avg_direction))) *
    Avg_direction;
    Di(count, 1) = dot(temp, temp);
    sum_sq_err = sum_sq_err + Di(count,1)^2;
end

N = size(P,1);

Emax = max(Di);
Emin = min(Di);
H_Straightness = abs(Emax) + abs(Emin);

tElapsed = toc(Time_Start);

% Display the result
disp_str_1 = sprintf('Direction cosines center of line are: %0.6f
%0.6f %0.6f', Prev_direction(1,1), Prev_direction(1,2),
Prev_direction(1,3));
disp_str_2 = sprintf('Sum of square of error is: %0.6f',
sum_sq_err);
disp_str_3 = sprintf('straightness error is: %0.6f',
H_Straightness);
disp_str_4 = sprintf('The elapsed time is : %0.6f', tElapsed);
disp(disp_str_1);
disp(disp_str_2);
disp(disp_str_3);
disp(disp_str_4);
2(a) A MATLAB Program to evaluate the flatness error using Maximum Distance Point Strategy.

% This program uses the MDPS to fit set of data points in plane.
% The data points should be stored in comma separated value file
% (csv). The first column in csv file represents x coordinate,
% second column represents y coordinate and third column represents
% of the point. This file is read by the program.

clear all
close all
c1c

% Prompts for selecting file
prompt = 'Enter filename to fit circle (*.csv): ';
str = input(prompt, 's');

Time_Start = tic;  % Start clock to measure performance

nop = size((P), 1);  % Number of points in set

% Initialize triplet matrix, size(n x 3)
% Create zero matrix of size: number of points by 3, each column is
% for point number, the first row by default 1, 2, ... N
triplets = zeros(nop, 3);

% In each triplet the first point is fixed by selecting each point
% once
for row=1:nop
    triplets(row,1)=row;
end

% Initialize Distance matrix, size(n x n)
Dist_m = zeros(nop, nop);

% Calculates euclidean distance between two points
for col=1:nop
    for row=1:nop
        Dist_m(row, col)=(P(col,1)−P(row,1))^2 + ...
        (P(col,2)−P(row,2))^2 + (P(col,3)−P(row,3))^2;
    end
end

% Find the indices of the maximum values of each column of matrix
% Dist_m and returns them in output vector I. If there are several
% identical maximum values, the index of the first one found
% is returned. So matrix 'I' is row matrix (size 1 x n) and
% represents index of each maximum value in the column
[C, I] = max(Dist_m);

% Second point in triplet is selected based on index value present
% in matrix 'I'. Distance for selected point is set to zero to
% avoid it's selection as third point in the triplet.
for col=1:nop
    triplets(col,2)=I(col);
    Dist_m(I(col),col)=0;
end
% Initialize normal distance index matrix Atti_dist_m, size(n x n)
atti_dist_m = zeros(nop, nop);

% Calculate normal distance index between line joined by first two
% points selected in triplet and each point in data set using
% equation 4.4
for col=1:nop
    for row=1:nop
        if Dist_m(row,col)== 0
            % Do nothing
        else
            x1=P(triplets(col,1),1);
y1=P(triplets(col,1),2);
z1=P(triplets(col,1),3);
x2=P(triplets(col,2),1);
y2=P(triplets(col,2),2);
z2=P(triplets(col,2),3);
x3=P(row,1);
y3=P(row,2);
z3=P(row,3);

            atti_dist_m(row,col)=( (x1-x3)^2 + (y1-y3)^2 + (z1-z3)^2) * ( (x2-x1)^2 + (y2-y1)^2 + (z2-z1)^2) - ( (x2-x1)*(x1-x3) + (y2-y1)*(y1-y3) + (z2-z1)*(z1-z3))^2;
        end
    end
end

[c, i] = max(atti_dist_m);

% Third point in triplet is selected based on index value present
% in matrix 'I'.
for col=1:nop
    triplets(col,3)=I(col);
    Dist_m(I(col),col)=0;
end

normal=zeros(nop,3);
normal_rhs=zeros(nop,1);

% Calculates direction cosines of normal to the plane passing
% through three selected point
for col=1:nop
    % First point in triplet
x1=P(triplets(col,1),1);
y1=P(triplets(col,1),2);
z1=P(triplets(col,1),3);

    % Second point in triplet
x2=P(triplets(col,2),1);
y2=P(triplets(col,2),2);
z2=P(triplets(col,2),3);

    % Third point in triplet
x3=P(triplets(col,3),1);
y3=P(triplets(col,3),2);
z3=P(triplets(col,3),3);

    a = [x2-x1 y2-y1 z2-z1];
b = [x3-x1 y3-y1 z3-z1];
Normal(col,:) = cross(a,b);
Normal_RHS(col, 1) = dot(Normal(col,:),P(triplets(col,1,:),));

% This is to select proper oriented normal
if (Normal_RHS(col, 1)<0)
    Normal(col,:) = (-1)*Normal(col,:);
    Normal_RHS(col, 1) = (-1)*Normal_RHS(col,1);
end

A = dot(Normal(col,:),Normal(col,:));
Normal(col,:)=(1/A)*Normal(col,:);
Normal_RHS(col, 1)=(1/A)*Normal_RHS(col, 1);
end

Cosines = zeros(nop,4);
for row=1:nop
    Cosines(row, 1) = vpa(Normal(row,1), 6);
    Cosines(row, 2) = vpa(Normal(row,2), 6);
    Cosines(row, 3) = vpa(Normal(row,3), 6);
    Cosines(row, 4) = vpa(Normal_RHS(row,1), 6);
end

% Initialize matrix for sum of square of error for each plane
% (size: n x 1)
sum_sq_err_m = zeros(nop,1);

% Calculate sum of square of error for each plane
for row=1:nop
    for counter=1:size(P,1)
        sum_sq_err_m(row,1) = sum_sq_err_m(row,1) + vpa(((P(counter, 1)*Cosines(row,1)) + (P(counter,2)*Cosines(row,2)) +
        (P(counter,3) * Cosines(row,3)))-Cosines(row,4))^2, 6);
    end
end

MAD = mean(sum_sq_err_m);
Ave_Cosine = mean(Cosines);
Prev_Cosine = zeros(1,4);

% Initialize matrix Di, size (n x 1). Stores value of distance from
% center of plane to the point
Di = zeros(nop,1);
sum_sq_err_avg = 0; % sum of squared error from found average plane
for counter = 1:size(P,1)
    Di(counter,1) = vpa(abs(((P(counter,1)*Avg_Cosines(1,1)) +
        (P(counter,2)*Avg_Cosines(1,2))+(P(counter,3) *
        Avg_Cosines(1,3))))-Avg_Cosines(1,4)), 6);
end

% Calculate flatness and store as previous
Prev_Flatness = abs(max(Di)) - abs(min(Di));

% Store average plane as previous plane
Prev_Cosine(1,1) = vpa(Avg_Cosines(1,1), 6);
Prev_Cosine(1,2) = vpa(Avg_Cosines(1,2), 6);
Prev_Cosine(1,3) = vpa(Avg_Cosines(1,3), 6);
Prev_Cosine(1,4) = vpa(Avg_Cosines(1,4), 6);

% Follow heuristic algorithm discussed in 4.3.1
while(1)
k = 0; % Count number of selected plane
m = 0;
Sum_A = 0;
Sum_Cosine=zeros(1,4);
for counter = 1:nop
    if sum_sq_err_m(counter,1) <= MAD
        Sum_Cosine(1,1) = Sum_Cosine(1,1)+Cosines(counter,1);
        Sum_Cosine(1,2) = Sum_Cosine(1,2)+Cosines(counter,2);
        Sum_Cosine(1,3) = Sum_Cosine(1,3)+Cosines(counter,3);
        Sum_Cosine(1,4) = Sum_Cosine(1,4)+Cosines(counter,4);
        k = k+1;
        A(counter,1) = sum_sq_err_m(counter,1);
    else
        Sum_Cosine(1,1) = Sum_Cosine(1,1);
        Sum_Cosine(1,2) = Sum_Cosine(1,2);
        Sum_Cosine(1,3) = Sum_Cosine(1,3);
        Sum_Cosine(1,4) = Sum_Cosine(1,4);
        sum_sq_err_m(counter,1) = MAD;
        A(counter,1) = 0;
    end
end
for counter = 1:nop
    if A(counter,1)>0
        m = m+1;
        Sum_A = Sum_A+A(counter,1);
    end
end
MAD = Sum_A/m;
Avg_Cosines = zeros(1,4);
Avg_Cosines(1,1) = vpa(Sum_Cosine(1,1)/k, 6);
Avg_Cosines(1,2) = vpa(Sum_Cosine(1,2)/k, 6);
Avg_Cosines(1,3) = vpa(Sum_Cosine(1,3)/k, 6);
Avg_Cosines(1,4) = vpa(Sum_Cosine(1,4)/k, 6);

sum_sq_err_avg = 0;
for counter = 1:size(P,1)
    Di(counter,1) = vpa(P(counter,1)*Avg_Cosines(1,1)+
                     P(counter,2)*Avg_Cosines(1,2)+P(counter,3) *
                     Avg_Cosines(1,3)-Avg_Cosines(1,4), 6);
    sum_sq_err_avg = sum_sq_err_avg + Di(counter, 1)^2;
end
Flatness = abs(max(Di))+ abs(min(Di));

if Prev_Flatness <= Flatness
    break
else
    Prev_Flatness = Flatness;
    Prev_Cosine(1,1) = Avg_Cosines(1,1);
    Prev_Cosine(1,2) = Avg_Cosines(1,2);
    Prev_Cosine(1,3) = Avg_Cosines(1,3);
    Prev_Cosine(1,4) = Avg_Cosines(1,4);
end
\[ J = \sqrt{\text{Prev\_Cosine}(1,1)\times\text{Prev\_Cosine}(1,1) + \text{Prev\_Cosine}(1,2)\times \text{Prev\_Cosine}(1,2) + \text{Prev\_Cosine}(1,3)\times\text{Prev\_Cosine}(1,3)}; \]
\[
\text{Prev\_Cosine} = (1/J)\times\text{Prev\_Cosine};
\]
\[
\text{sum\_sq\_err} = 0; \quad \% \text{Sum of Square of errors (di}^2\text{)}
\]
\[
\text{for} \; \text{counter} = 1: \text{size(P,1)}
\]
\[
\text{Di(counter,1)} = (P(counter,1)\times\text{Prev\_Cosine}(1,1) + P(counter,2)\times \text{Prev\_Cosine}(1,2) + P(counter,3)\times\text{Prev\_Cosine}(1,3) - \text{Prev\_Cosine}(1,4); \}
\]
\[
\text{sum\_sq\_err} = \text{sum\_sq\_err} + \text{Di(counter,1)}^2;
\]
\[
\text{end}
\]
\[
\text{N} = \text{size(P,1)};
\]
\[
\text{Emax} = \text{max(Di)};
\]
\[
\text{Emin} = \text{min(Di)};
\]
\[
\text{H\_Flatness} = \text{abs(Emax)} + \text{abs(Emin)};
\]
\[
\text{tElapsed} = \text{toc(Time\_Start)};
\]
\[
\% \text{Display the result}
\]
\[
\text{disp\_str\_1} = \text{sprintf('Direction cosines center of plane are: %0.6f %0.6f %0.6f', \text{Prev\_Cosine}(1,1), \text{Prev\_Cosine}(1,2), \text{Prev\_Cosine}(1,3));}
\]
\[
\text{disp\_str\_2} = \text{sprintf('Sum of square of error is: %0.6f', \text{sum\_sq\_err});}
\]
\[
\text{disp\_str\_3} = \text{sprintf('Flatness error is: %0.6f', \text{H\_Flatness});}
\]
\[
\text{disp\_str\_4} = \text{sprintf('The elapsed time is: %0.6f', \text{tElapsed});}
\]
\[
\text{disp(disp\_str\_1);}
\]
\[
\text{disp(disp\_str\_2);}
\]
\[
\text{disp(disp\_str\_3);}
\]
\[
\text{disp(disp\_str\_4);}
\]
2(b) A MATLAB Program to evaluate the flatness error using Least Square Method.

% This program uses the SVD Algorithm to fit set of data points in a plane. The data points should be stored in comma separated value file (csv). The first column in csv file represents x coordinate, second column represents y coordinate and third column represents of the point. This file is read by the program.

close all % Remove all figures
clear all % Remove all items from workspace, freeing up system memory
clc % Clear command window

% Prompts for selecting file
prompt = 'Enter the filename to fit plane (*.csv): ';
str = input(prompt, 's');

% The data points are read in matrix P
P=csvread(str);

tStart = tic; % Start clock to measure performance

nop = size((P), 1); % Number of points in set

sum_Pi = zeros(1,3);

for n=1:nop;
    sum_Pi = sum_Pi + P(n,:);
end

ave_Pi = sum_Pi/nop;

A = zeros( size((P), 1), 3);

for n=1:nop;
    A(n,1) = P(n,1) - ave_Pi(1, 1);
    A(n,2) = P(n,2) - ave_Pi(1, 2);
    A(n,3) = P(n,3) - ave_Pi(1, 3);
end

[U, S, V] = svd(A);

% Initialize matrix Di, size (n x 1). Stores value of normal % distance from best plane to the point
Di = zeros(nop,1);
sum_sq_err = 0; % Sum of Square of errors (di^2)

for n = 1:nop
    Di(n,1) = V(1,3).*(P(n,1) - ave_Pi(1, 1)) + V(2,3).*P(n,2) ... - ave_Pi(1, 2)) + V(3,3).*P(n,3) - ave_Pi(1, 3));
    sum_sq_err = sum_sq_err + Di(n,1)^2;
end

Flatness = max(Di) - min(Di);

tElapsed = toc(tStart); % Stop clock to measure performance
% Display the result

disp_str_1 = sprintf('Direction cosines center of circle are: %0.6f %0.6f %0.6f', V(1,3), V(2,3), V(3,3));
disp_str_2 = sprintf('Sum of square of error is: %0.6f', sum_sq_err);
disp_str_3 = sprintf('Flatness error is: %0.6f', Flatness);
disp_str_4 = sprintf('The elapsed time is : %0.6f', tElapsed);
disp(disp_str_1);
disp(disp_str_2);
disp(disp_str_3);
disp(disp_str_4);
3(a) A MATLAB Program to evaluate the circularity error using Maximum Distance Point Strategy.

% This program uses the Maximum Distance Point Strategy (MDPS) to fit set of data points in circle. The data points should be stored in comma separated value file (csv). The first column in csv file should be x coordinate and second column would be y coordinate of the point. This file is read by the program.

close all % Remove all figures
clear all % Remove all items from workspace, freeing up % system memory
clc % Clear command window

% Prompts for selecting file
prompt = 'Enter filename to fit circle (*.csv): ';
str = input(prompt, 's');

% The data points are read in matrix P
P = csvread(str);

tStart = tic; % Start clock to measure performance

nop = size({P}, 1); % Number of points in set

% Initialize triplet matrix, size(n-1 x 3)
triplets = zeros(nop, 3);

% In each triplet the first point is fixed by selecting each % point once
for row=1:nop
    triplets(row, 1) = row;
end

% Initialize distance matrix dist_m, size(n x n)
Dist_m = zeros(nop, nop);

% Calculates distance between two points
for col=1:nop
    for row=1:nop
        Dist_m(row, col) = sqrt((P(col, 1) - P(row, 1))^2 + (P(col, 2) - P(row, 2))^2);
    end
end

% finds the indices of the maximum values of each column of matrix
% Dist_m and returns them in output vector I. If there are several % identical maximum values, the index of the first one found % is returned. So matrix 'I' is row matrix (size 1 x n) and % represents index of each maximum value in the column
[C, I] = max(Dist_m);

% Second point in triplet is selected based on index value present % in matrix 'I'. Distance for selected point is set to zero to % avoid it's selection as third point in the triplet.
for col=1:nop
    triplets(col, 2) = I(col);
    Dist_m(I(col), col) = 0;
end
% Initialize normal distance index matrix Atti_dist_m, size(n x n)
Att_i_Dist_m = zeros(nop, nop);

% Calculate normal distance index between line joined by first two
% points selected in triplet and each point in data set using
% equation 4.4
for col=1:nop
    for row=1:nop
        if Dist_m(row,col)==0
            % Do nothing
        else
            x1=P(triplets(col,1),1);
            y1=P(triplets(col,1),2);
            x2=P(triplets(col,2),1);
            y2=P(triplets(col,2),2);
            x3=P(row,1);
            y3=P(row,2);
            Att_i_Dist_m(row,col)=abs((y2-y1)*(x3-x1)+(x1-x2)*(y3-
                                y1));
        end
    end
end

[C, I] = max(Att_i_Dist_m);
% Third point in triplet is selected based on index value present
% in matrix 'I'.
for col=1:nop
    triplets(col,3)=I(col);
    Dist_m(I(col),col)=0;
end

% Initialize matrix for centers (size: n x 3),
% Store x coordinate (1st column), y coordinate (second column)
% and radius (third column)
Centers = zeros(nop,3);
X1 = zeros(2,1);
X2 = zeros(2,2);

% Calculate centers and radius for each triplet
for row=1:nop
    X1(1)= P(triplets(row,2),1)^2 -
P(triplets(row,1),1)^2+P(triplets(row,2),2)^2 -
P(triplets(row,1),2)^2;
    X1(2)= P(triplets(row,3),1)^2 -
P(triplets(row,1),1)^2+P(triplets(row,3),2)^2 -
P(triplets(row,1),2)^2;
    X2(1,1)= 2*P(triplets(row,2),1) - 2*P(triplets(row,1),1);
    X2(1,2)= 2*P(triplets(row,2),2) - 2*P(triplets(row,1),2);
    X2(2,1)= 2*P(triplets(row,3),1) - 2*P(triplets(row,1),1);
    X2(2,2)= 2*P(triplets(row,3),2) - 2*P(triplets(row,1),2);
    X=X2\X1;
    R= sqrt((X(1,1)-P(triplets(row,1),1))^2 + (X(2,1)-
P(triplets(row,1),2))^2);
    Centers(row, 1)= X(1,1);
    Centers(row, 2)= X(2,1);
    Centers(row, 3) = R;
end

% Initialize matrix for sum of square of error for each circle
% (size: n x 1)
sum_sq_err_m = zeros(nop,1);
% Calculate sum of square of error for each circle
for row=1:nop
    for counter=1:size(P,1)
        sum_sq_err_m(row,1) = sum_sq_err_m(row,1) +
        (sqrt((P(counter,1) - ...)
            Centers(row,1))^2 + (P(counter,2) - Centers(row,2))^2 -
            Centers(row,3))^2;
    end
end

MAD = mean(sum_sq_err_m); % mean of sum of square of errors
Prev_Avg_Center = zeros(1,3);
Sum_Center=zeros(1,3);

% Find first circle with average of center of coordinates of each 
% circle and average of radii
for counter=1:size(Centers,1)
    Sum_Center(1,1) = Sum_Center(1,1) + Centers(counter,1);
    Sum_Center(1,2) = Sum_Center(1,2) + Centers(counter,2);
    Sum_Center(1,3) = Sum_Center(1,3) + Centers(counter,3);
end
Avg_Center = zeros(1,3);
k = 1; % Initially all circles are selected
Avg_Center(1,1) = Sum_Center(1,1)/k;
Avg_Center(1,2) = Sum_Center(1,2)/k;
Avg_Center(1,3) = Sum_Center(1,3)/k;

sum_sq_err_avg = 0; % sum of squared error from found average 
% circle

% Initialize matrix Di, size (n x 1). Stores value of distance from 
% center of circle to the point
di = zeros(nop,1);
for counter=1:nop
    di(counter, 1) = sqrt((P(counter,1) - Avg_Center(1,1))^2 + ... 
        (P(counter,2) - Avg_Center(1,2))^2 - Avg_Center(1,3));
    sum_sq_err_avg = sum_sq_err_avg + di(counter, 1)^2;
end

% Calculate circularity and store as previous
Prev_Ht = abs(max(di)) + abs(min(di));
% Store average circle as previous circle
Prev_Avg_Center(1,1) = Avg_Center(1,1);
Prev_Avg_Center(1,2) = Avg_Center(1,2);
Prev_Avg_Center(1,3) = Avg_Center(1,3);

% Follow heuristi algorithm discussed in 4.3.2 while (1)

k = 0; % Count number of selected circle
m = 0;
Sum_A = 0;
Sum_Center = zeros(1,3);
for counter=1:nop
    if sum_sq_err_m(counter,1) <= MAD
        Sum_Center(1,1) = Sum_Center(1,1) + Centers(counter,1);
        Sum_Center(1,2) = Sum_Center(1,2) + Centers(counter,2);
        Sum_Center(1,3) = Sum_Center(1,3) + Centers(counter,3);
    end
end

160
Sum_Center(1, 3) = Sum_Center(1, 3) + Centers(counter, 3);  
k = k + 1;  
A(counter, 1) = sum_sq_err_m(counter, 1);  
else  
  Sum_Center(1, 1) = Sum_Center(1, 1);  
  Sum_Center(1, 2) = Sum_Center(1, 2);  
  Sum_Center(1, 3) = Sum_Center(1, 3);  
  sum_sq_err_m(counter, 1) = MAD;  
  A(counter, 1) = 0;  
end  
end  
for counter = 1: size(Centers, 1)  
  if A(counter, 1) > 0  
    m = m + 1;  
    Sum_A = Sum_A + A(counter, 1);  
  end  
end  
MAD = Sum_A / m;  
Avg_Center = zeros(1, 3);  
Avg_Center(1, 1) = Sum_Center(1, 1) / k;  
Avg_Center(1, 2) = Sum_Center(1, 2) / k;  
Avg_Center(1, 3) = Sum_Center(1, 3) / k;  
sum_sq_err_avg = 0;  
for counter = 1: nump  
  di(counter, 1) = sqrt((P(counter, 1) - Avg_Center(1, 1))^2 +  
        (P(counter, 2) - Avg_Center(1, 2))^2) - Avg_Center(1, 3);  
  sum_sq_err_avg = sum_sq_err_avg + di(counter, 1)^2;  
end  
Ht = abs(max(di)) + abs(min(di));  
if Prev_Ht <= Ht  
  break  
else  
  Prev_Ht = Ht;  
  Prev_Avg_Center(1, 1) = Avg_Center(1, 1);  
  Prev_Avg_Center(1, 2) = Avg_Center(1, 2);  
  Prev_Avg_Center(1, 3) = Avg_Center(1, 3);  
end  
end  
sum_sq_err_avg = 0;  
for counter = 1: nump  
  di(counter, 1) = sqrt((P(counter, 1) - Prev_Avg_Center(1, 1))^2 +  
        (P(counter, 2) - Prev_Avg_Center(1, 2))^2) - Prev_Avg_Center(1, 3);  
  sum_sq_err_avg = sum_sq_err_avg + di(counter, 1)^2;  
end  
tElapsed = toc(tStart);  

% Display the result  
disp_str_1 = sprintf('Coordinate of center of circle is: %.4f, ...  
                      % .4f',  
                      Prev_Avg_Center(1, 1), Prev_Avg_Center(1, 2));  
disp_str_2 = sprintf('Radius of circle is: %.4f',  
                      Prev_Avg_Center(1, 3));  
disp_str_3 = sprintf('Sum of square of error is: %0.6f',  
                      sum_sq_err_avg);  
disp_str_4 = sprintf('Circularity error is: %0.6f',  
                      Prev_Ht);  
disp_str_5 = sprintf('The elapsed time is: %0.6f', tElapsed);
disp(disp_str_1);
disp(disp_str_2);
disp(disp_str_3);
disp(disp_str_4);
disp(disp_str_5);

% Plot circle
u = 0:0.01:2*pi;
XofC = X(1)+R*cos(u);
YofC = X(2)+R*sin(u);
plot(double(XofC), double(YofC));
hold on
n = 1:nop;
plot(P(n,1),P(n,2),'r*');
3(b) A MATLAB Program to evaluate the circularity error using Least Square Method (Line-Search Strategy – Gauss-Newton Method).

% This program uses the Gauss-Newton Algorithm to fit set of data points in circle. The data points should be stored in comma separated value file (.csv). The first column in csv file should be x coordinate and second column would be y coordinate of the % point. This file is read by the program.

close all % Remove all figures
clear all % Remove all items from workspace, freeing up % system memory
clc % Clear command window

% Prompts for selecting file
prompt = 'Enter filename to fit circle (*.csv): ';
str = input(prompt, 's');

% The data points are read in matrix P
P=csvread(str);

tStart = tic; % Start clock to measure performance

nop = size(P, 1); % Number of points in set
A = zeros(nop, 3); % Initialization of matrix A, size (n x 3)
B = zeros(nop, 1); % Initialization of matrix B, size (n x 1)
J = zeros(nop, 3); % Initialization of matrix J, size (n x 3)

% Calculates A and B
for n=1:nop;
    A(n,1)=2*P(n,1);
    A(n,2)=2*P(n,2);
    A(n,3)=-1;
    B(n)=P(n,1)^2+P(n,2)^2;
end

% Calculates initial estimate of coordinate of center and radius
X = (A'*A)/(A'*B);
R = sqrt(X(1)*X(1)+X(2)*X(2)-X(3));

% Initialize matrix Di, size (n x 1). Stores value of distance from % center of circle to the point
di = zeros(nop,1);

counter=1;
for counter = 1:10
% Calculate distance from center of the circle to nth point in set % and also calculate Jacobian
for n=1:nop
    di(n,1)=sqrt((P(n,1)-X(1, 1))^2+(P(n,2)-X(2, 1))^2) - R;
    J(n,1) = -(P(n,1)-X(1)) / (sqrt((P(n,1)-X(1, 1))^2 + ... (P(n,2)-X(2, 1))^2));
    J(n,2) = -(P(n,2)-X(2)) / (sqrt((P(n,1)-X(1, 1))^2 + ... (P(n,2)-X(2, 1))^2));
    J(n,3) = -1;
end

% Calculate update value
U = (J'*J)/(J'*(-di));
X = X + U;                    % Update center coordinate
R = R + U(3, 1);             % Update radius

% Condition of required precision in result
if (double(abs(U(1)))<1*1e-10)    
    break
end

sum_sq_err = 0;               % Sum of Square of errors (di^2)
for n = 1:ncp
    sum_sq_err = sum_sq_err + di(n,1)^2;
end

Circularity = max(di) - min(di);

tElapsed = toc(tStart);       % Stop clock to measure performance

% Display the result
disp_str_1 = sprintf('Coordinate of center of circle is: %0.4f
%0.4f', X(1,1), X(2,1));
disp_str_2 = sprintf('Radius of circle is: %0.4f', R);
disp_str_3 = sprintf('Sum of square of error is: %0.6f',
sum_sq_err);
disp_str_4 = sprintf('Circularity error is: %0.6f', Circularity);
disp_str_5 = sprintf('The elapsed time is : %0.6f', tElapsed);
disp(disp_str_1);          
disp(disp_str_2);          
disp(disp_str_3);          
disp(disp_str_4);

% Plot circle
u = 0:0.01:2*pi;
XofC = X(1)+R*cos(u);
YofC = X(2)+R*sin(u);
plot(double(XofC), double(YofC));
hold on
n = 1:ncp;
plot(P(n,1),P(n,2),'r*');
3(c) A MATLAB Program to evaluate the circularity error using Least Square Method (Trust-Region Strategy – Dogleg Method).

% This program uses the Dogleg method, a method based on trust
% region strategy to fit set of data points in circle. The data
% points should be stored in comma separated value file (csv). The
% first column in csv file should be x coordinate and second column
% would be y coordinate of the point. This file is read by the
% program.

close all     % Remove all figures
clear all     % Remove all items from workspace, freeing up
              % system memory
clc           % Clear command window

% Prompts for selecting file
prompt = 'Enter the filename to fit circle (*.csv): ';
str = input(prompt, 's');

% The data points are read in matrix P
P=csvread(str);
digits(16)
tStart = tic;  % Start clock to measure performance

nop = size(P, 1);         % Number of points in set
DeltaZero1 = 0;
alpha = 0;
L=0;
X=[0;0;0];
J = zeros(nop,3);         % jacobian matrix of the first order
derivatives              % of objective function is initialized
rbarX=zeros(nop,1);
rbarY=zeros(nop,1);

for counter=1:nop
    rbarX(counter,1) = sqrt((P(counter, 1) - X(1,1))^2 +
                            (P(counter, 2) - X(2,1))^2) - X(3,1);
end

% selection of trust region radius deltazero.
DeltaBarMax = max(rbarX);
DeltaBarMin = min(rbarX);
DeltaZero = 0.5*(DeltaBarMin+DeltaBarMax); %*rand;
Eta = 0.24;

Pbar1 = zeros(3,1);
Pbar2 = zeros(3,1);
Pbar3 = zeros(3,1);
Pbar  = zeros(3,1);
m = 0;

% iterations start to find the centre and radius of the best fit
% circle.
for k = 1:10
    X1 = X;
    % rbarX and the Jacobians are calculating at each point.
    for counter=1:nop
rbarX(counter, 1) = sqrt((P(counter, 1) - X(1, 1))^2 + (P(counter, 2) - X(2, 1))^2 - X(3, 1));
J(counter, 1) = -(P(counter, 1) - X(1, 1))/(rbarX(counter, 1) + X(3, 1));
J(counter, 2) = -(P(counter, 2) - X(2, 1))/(rbarX(counter, 1) + X(3, 1));
J(counter, 3) = -1;
end
% calculations of parameters required for the method to run.
A=J'*rbarX;
B=J'*J; % approximation to second derivative matrix
% (called Hessian matrix) that is taken in this % method
C=A'*B*A;
D=A'*A;
F=-D/C; % D/(deltaZero*C)
% Calculation of the step using steepest descent direction
PU=F*A;
% Calculation of full step direction solution
PB=inv(B)*(-A);
NormPU=sqrt(PU'*PU);
NormPB=sqrt(PB'*PB);
NormPBPU=sqrt((PB-PU)'*(PB-PU));
% Application of selection of path using the Dogleg method
% If full step is taking the solution beyond the trust region
if (NormPB>DeltaZero)
% If steepest descent step also taking the solution beyond the % trust region.
if (NormPU>=DeltaZero)
Pbar1 = -(DeltaZero/NormPU)*PU;
else
a = NormPBPU^2;
b = dot((2*PU),(PB-PU));
c = NormPU^2 - DeltaZero^2;
d = sqrt(b^2 - 4*a*c);
a1 = (-b+d)/2*a;
a2 = (-b-d)/2*a;
if( 0 < a1) && (a1 < 1 )
alpha = a1;
elseif( 0 < a2) && (a2 < 1 )
alpha = a2;
else
alpha = 0;
end
Pbar2=PU+alpha*(PB-PU);
% then contain the solution to within trust region or on the % boundary of the region. To do so the path selected in finding % Pbar has the shape of Dog's leg and hence the method is called the % Dogleg method.
end
else
Pbar3=PB;
end

166
if (NormPB>DeltaZero)
    if (NormPU>=DeltaZero)
        Pbar = Pbar1;
    else
        Pbar = Pbar2;
    end
else
    Pbar = Pbar3;
end

NormPbar=sqrt(Pbar'*Pbar);
Y = X + Pbar;  % next iteration
fX = (0.5)*((rbarX)'*(rbarX));

for counter=1:nop
    rbarY(counter,1) = sqrt((P(counter, 1)-Y(1,1))^2+(P(counter, 2)-Y(2,1))^2)-Y(3,1);
end

fY = (0.5)*((rbarY)'*(rbarY));
E = Pbar' * A;
G = Pbar'*B*Pbar;
H = 0.5 * G;
mDiff = E + H;
rho = (fY - fX)/mDiff;
% decision taking parameter. it is the ratio of change in values of % function at new and old X to the change in values of trust region % model function there.

if (rho < 0.25) % selection criteria
    DeltaZero = 0.25*NormPbar;
elseif(rho > 0.75) && (NormPbar>=DeltaZero)
    DeltaZero1 = 2.0*DeltaZero;
    L = [DeltaZero1 DeltaBarMax];
    DeltaZero = min(L);
end

if (rho>Eta)
    X = Y;
end

NormX1=sqrt(X1'*X1);
NormX=sqrt(X'*X);
% if (NormX1==NormX)
%     break
% end
    m = m+1;
end

AMD_H_Avg=0;
sum_sq_err = 0;  % Sum of Square of errors (di^2)
di = zeros(nop,1);
for n=1:nop
    di(n,1)=sqrt(((P(n,1) - X(1,1))^2 + (P(n,2) - X(2,1))^2)) - X(3,1);
    sum_sq_err = sum_sq_err + di(n,1)^2;
end

167
Emax = max(di);
Emin = min(di);
Circularity = abs(Emax) + abs(Emin);

tElapsed = toc(tStart); % Stop clock to measure performance

% Display the result
disp_str_1 = sprintf('Coordinate of center of circle is: %0.4f 
%0.4f', X(1,1), X(2,1));
disp_str_2 = sprintf('Radius of circle is: %0.4f', X(3,1));
disp_str_3 = sprintf('Sum of square of error is: %0.6f', 
sum_sq_err);
disp_str_4 = sprintf('Circularity error is: %0.6f', Circularity);
disp_str_5 = sprintf('The elapsed time is : %0.6f', tElapsed);
disp(disp_str_1);
disp(disp_str_2);
disp(disp_str_3);
disp(disp_str_4);

% Plot circle
u = 0:0.01:2*pi;
XofC = X(1)+R*cos(u);
YofC = X(2)+R*sin(u);
plot(double(XofC), double(YofC));
hold on
n = 1:nop;
plot(P(n,1),P(n,2),'r*');
4(a) A MATLAB Program to evaluate the sphericity error using Maximum Distance Point Strategy.

% This program uses the Maximum Distance Point Strategy (MDPS) to fit set of data points in sphere. The data points should be stored in comma separated value file (csv). The first column in csv file represents x coordinate, second column represents y coordinate and third column represents of the point. This file is read by the program.

close all % Remove all figures
clear all % Remove all items from workspace, freeing up % system memory
clc % Clear command window

% Prompts for selecting file
prompt = 'Enter the filename to fit sphere (*.csv): ';
str = input(prompt, 's');

% The data points are read in matrix P
P=csvread(str);

tStart = tic; % Start clock to measure performance
nop = size((P), 1); % Number of points in set

% Initialize distance matrix dist_m, size(n x n)
Dist_m = zeros(nop, nop);

% Calculates distance between two points
for col=1:nop
  for row=1:nop
    Dist_m(row, col)=sqrt((P(col,1)-P(row,1))^2 + (P(col,2)-P(row,2))^2 + (P(col,3)-P(row,3))^2);
  end
end

% Initialize quadruplets matrix, size(n-1 x 3)
quadruplets = zeros(nop,4);

% In each quadruplets the first point is fixed by selecting each % point once
for row=1:nop
  quadruplets(row,1)=row;
end

% Finds indices of the maximum values of each column of matrix % Dist_m and returns them in output vector I. If there are several % identical maximum values, the index of the first one found % is returned. So matrix 'I' is row matrix (size 1 x n) and % represents index of each maximum value in the column.
[C, I] = max(Dist_m);

% Second point in quadruplets is selected based on index value % present in matrix 'I'. Distance for selected point is set to zero % to avoid it's selection as third point in the quadruplets.
for col=1:nop
  quadruplets(col,2)=I(col);
  Dist_m(I(col),col)=0;
end
% Initialize normal distance matrix AP, size(n x n)
AP = zeros(nop, nop);

% Calculate normal distance between line joined by first two points
% selected in quadruplets and each point in data set using eq. 4.6
for col=1:nop
    for row=1:size(P,1)
        if Dist_m(row,col)== 0
            % Do nothing
        else
            x1=P(quadruplets(col,1),1);
            y1=P(quadruplets(col,1),2);
            z1=P(quadruplets(col,1),3);
            x2=P(quadruplets(col,2),1);
            y2=P(quadruplets(col,2),2);
            z2=P(quadruplets(col,2),3);
            x3=P(row,1);
            y3=P(row,2);
            z3=P(row,3);
            AP(row,col)=((x1-x3)^2 + (y1-y3)^2 + (z1-z3)^2)*((x2-x1)^2 + (y2-y1)^2 + (z2-z1)^2) - (x2-x1)*(x1-x3) + (y2-y1)*(y1-y3) + (z2-z1)*(z1-z3))^2;
        end
    end
end

[C, I] = max(AP);
% Third point in quadruplets is selected based on index value
% present in matrix 'I'.
for col=1:nop
    quadruplets(col,3)=I(col);
    Dist_m(I(col),col)=0;
end

xi=zeros(3,3);

% Initialize normal distance matrix DP, size(n x n)
DP = zeros(nop, nop);

% Calculate normal distance index between plane passing through
% first three points selected in quadruplets and each point in data
% using eq. 4.7
for col=1:nop
    for row=1:nop
        if Dist_m(row,col)== 0
            DP(row,col)=0;
        elseif P(quadruplets(col,1),3)==P(row,3) || ...
            P(quadruplets(col,2),3)==P(row,3) || ...
            P(quadruplets(col,3),3)==P(row,3)
            DP(row,col)=0;
        else
            x1=P(quadruplets(col,1),1);
            y1=P(quadruplets(col,1),2);
            z1=P(quadruplets(col,1),3);
            x2=P(quadruplets(col,2),1);
            y2=P(quadruplets(col,2),2);
            z2=P(quadruplets(col,2),3);
            x3=P(quadruplets(col,3),1);
            y3=P(quadruplets(col,3),2);
            z3=P(quadruplets(col,3),3);
        end
    end
end
x4=P(row,1);
y4=P(row,2);
z4=P(row,3);
xi(1,1)=x4-x1;
xi(1,2)=y4-y1;
xi(1,3)=x4-z1;
xi(2,1)=x2-x1;
xi(2,2)=y2-y1;
xi(2,3)=x2-z1;
xi(3,1)=x3-x1;
xi(3,2)=y3-y1;
xi(3,3)=z3-y1;
DP(row,col)=abs(vpa(det(xi)));
end
end

% Fourth point in quadruplets is selected based on index value
% present in matrix 'I'.
[C, I] = max(DP);
for col=1:nop
    quadruplets(col,4)=I(col);
    Dist_m(I(col),col)=0;
end

% Initialize matrix for centers (size: n x 4),
% Store x coordinate (1st column), y coordinate (second column),
% z coordinate (third column) and radius (forth column)
Centers = zeros(nop,4);
X1 = zeros(3,1);
X2 = zeros(3,3);

for row=1:size(quadruplets,1)
    X1(1)= P(quadruplets(row,2),1)^2 - P(quadruplets(row,1),1)^2 +
    P(quadruplets(row,2),2)^2 - P(quadruplets(row,1),2)^2 +
    P(quadruplets(row,2),3)^2 - P(quadruplets(row,1),3)^2;
    X1(2)= P(quadruplets(row,3),1)^2 - P(quadruplets(row,1),1)^2 +
    P(quadruplets(row,3),2)^2 - P(quadruplets(row,1),2)^2 +
    P(quadruplets(row,3),3)^2 - P(quadruplets(row,1),3)^2;
    X1(3)= P(quadruplets(row,4),1)^2 - P(quadruplets(row,1),1)^2 +
    P(quadruplets(row,4),2)^2 - P(quadruplets(row,1),2)^2 +
    P(quadruplets(row,4),3)^2 - P(quadruplets(row,1),3)^2;
    X2(1,1)= 2*P(quadruplets(row,2),1) - 2*P(quadruplets(row,1),1);
    X2(1,2)= 2*P(quadruplets(row,2),2) - 2*P(quadruplets(row,1),2);
    X2(1,3)= 2*P(quadruplets(row,2),3) - 2*P(quadruplets(row,1),3);
    X2(2,1)= 2*P(quadruplets(row,3),1) - 2*P(quadruplets(row,1),1);
    X2(2,2)= 2*P(quadruplets(row,3),2) - 2*P(quadruplets(row,1),2);
    X2(2,3)= 2*P(quadruplets(row,3),3) - 2*P(quadruplets(row,1),3);
    X2(3,1)= 2*P(quadruplets(row,4),1) - 2*P(quadruplets(row,1),1);
    X2(3,2)= 2*P(quadruplets(row,4),2) - 2*P(quadruplets(row,1),2);
    X2(3,3)= 2*P(quadruplets(row,4),3) - 2*P(quadruplets(row,1),3);
    X = X2\X1;
    R= sqrt(((X(1,1) - P(quadruplets(row,1),1))^2 + (X(2,1) -
    P(quadruplets(row,1),2))^2) + (X(3,1) -
    P(quadruplets(row,1),3))^2);
    Centers(row, 1) = X(1,1);
    Centers(row, 2) = X(2,1);
    Centers(row, 3) = X(3,1);
    Centers(row, 4) = R;
end
% Initialize matrix for sum of square of error for each circle
% (size: n x 1)
sum_sq_err_m = zeros(nop,1);

% Calculate sum of square of error for each sphere
for row=1:nop
    for counter=1:size(P,1)
        sum_sq_err_m(row,1) = sum_sq_err_m(row,1) +
        (sqrt((P(counter,1) - Centers(row,1))^2 + (P(counter,2) -
            Centers(row,2))^2 + (P(counter,3) - Centers(row,3))^2) -
            Centers(row,4))^2;
    end
end

MAD=mean(sum_sq_err_m);  % mean of sum of square of errors
SD=std(sum_sq_err_m);

Prev_Avg_Center = zeros(1,4);
Sum_Center=zeros(1,4);

% Find first sphere with average of center of coordinates of each
% sphere and average of radii
for counter=1:nop
    Sum_Center(1,1) = Sum_Center(1,1)+Centers(counter,1);
    Sum_Center(1,2) = Sum_Center(1,2)+Centers(counter,2);
    Sum_Center(1,3) = Sum_Center(1,3)+Centers(counter,3);
    Sum_Center(1,4) = Sum_Center(1,4)+Centers(counter,4);
end

k = nop;  % Initially all spheres are selected
Avg_Center = zeros(1,4);
Avg_Center(1,1) = vpa(Sum_Center(1,1)/k, 6);
Avg_Center(1,2) = vpa(Sum_Center(1,2)/k, 6);
Avg_Center(1,3) = vpa(Sum_Center(1,3)/k, 6);
Avg_Center(1,4) = vpa(Sum_Center(1,4)/k, 6);

sum_sq_err_avg = 0;  % sum of squared error from found average
spheres

% Initialize matrix di, size (n x 1). Stores value of distance from
% center of circle to the point
di = zeros(nop,1);

for counter=1:nop
    di(counter, 1) = sqrt((P(counter,1) - Avg_Center(1,1))^2 +
        (P(counter,2) - Avg_Center(1,2))^2 + (P(counter,3) -
            Avg_Center(1,3))^2) - Avg_Center(1,4);
    sum_sq_err_avg = sum_sq_err_avg + di(counter, 1)^2;
end

% Calculate sphericity and store as previous
Prev_Ht = abs(max(di)) + abs(min(di));
Prev_Avg_Center(1,1) = Avg_Center(1,1);
Prev_Avg_Center(1,2) = Avg_Center(1,2);
Prev_Avg_Center(1,3) = Avg_Center(1,3);
Prev_Avg_Center(1,4) = Avg_Center(1,4);

% Follow heuristics algorithm discussed in 4.3.2 
while (1)
    k=0;  % Counts numbers of selected circle
    ...
m=0;
Sum_A=0;
Sum_Center=zeros(1,4);
for counter=1:size(Centers,1)
    if le(sum_sq_err_m(counter,1), MAD)
        Sum_Center(1,1)= Sum_Center(1,1)+Centers(counter,1);
        Sum_Center(1,2)= Sum_Center(1,2)+Centers(counter,2);
        Sum_Center(1,3)= Sum_Center(1,3)+Centers(counter,3);
        Sum_Center(1,4)= Sum_Center(1,4)+Centers(counter,4);
        k=k+1;
        A(counter,1)=sum_sq_err_m(counter,1);
    else
        Sum_Center(1,1)= Sum_Center(1,1);
        Sum_Center(1,2)= Sum_Center(1,2);
        Sum_Center(1,3)= Sum_Center(1,3);
        Sum_Center(1,4)= Sum_Center(1,4);
        sum_sq_err_m(counter,1)=MAD;
        A(counter,1)=0;
    end
end
for counter=1:size(Centers,1)
    if A(counter,1)>0
        m=m+1;
        Sum_A=Sum_A+A(counter,1);
    end
end
MAD=Sum_A/m;
SD=std(sum_sq_err_m);
Avg_Center = zeros(1,4);
Avg_Center(1,1) = vpa(Sum_Center(1,1)/k, 6);
Avg_Center(1,2) = vpa(Sum_Center(1,2)/k, 6);
Avg_Center(1,3) = vpa(Sum_Center(1,3)/k, 6);
Avg_Center(1,4) = vpa(Sum_Center(1,4)/k, 6);

sum_sq_err_avg = 0;
for counter=1:nop
    di(counter, 1) = sqrt((P(counter,1) - Avg_Center(1,1))^2 +
                (P(counter,2) - Avg_Center(1,2))^2 +
                (P(counter,3) -
                Avg_Center(1,3))^2) -
                Avg_Center(1,4);
    sum_sq_err_avg = sum_sq_err_avg + di(counter, 1)^2;
end
Ht = abs(max(di)) + abs(min(di));
if Prev_Ht <= Ht
    break
else
    Prev_Ht = Ht;
    Prev_Avg_Center(1,1) = Avg_Center(1,1);
    Prev_Avg_Center(1,2) = Avg_Center(1,2);
    Prev_Avg_Center(1,3) = Avg_Center(1,3);
    Prev_Avg_Center(1,4) = Avg_Center(1,4);
end

sum_sq_err_avg = 0;
for counter=1:nop
    di(counter, 1) = sqrt((P(counter,1) - Prev_Avg_Center(1,1))^2 +
                (P(counter,2) - Prev_Avg_Center(1,2))^2 +
                (P(counter,3) -
                Prev_Avg_Center(1,3))^2) -
                Prev_Avg_Center(1,4);
sum_sq_err_avg = sum_sq_err_avg + di(counter, 1)^2;
end

tElapsed = toc(tStart); % Stop clock to measure performance

% Display the result
disp_str_1 = sprintf('Coordinate of center of sphere is: %0.4f %0.4f %0.4f', Prev_Avg_Center(1,1), Prev_Avg_Center(1,2), Prev_Avg_Center(1,3));
disp_str_2 = sprintf('Radius of sphere is: %0.4f', Prev_Avg_Center(1,4));
disp_str_3 = sprintf('Sum of square of error is: %0.6f', sum_sq_err_avg);
disp_str_4 = sprintf('Sphericity error is: %0.6f', Prev_Ht);
disp_str_5 = sprintf('The elapsed time is: %0.6f', tElapsed);
disp(disp_str_1);
disp(disp_str_2);
disp(disp_str_3);
disp(disp_str_4);
disp(disp_str_5);

% Plot sphere
[x,y,z] = ellipsoid(Prev_Avg_Center(1,1), Prev_Avg_Center(1,2), Prev_Avg_Center(1,3), Prev_Avg_Center(1,4),
                    Prev_Avg_Center(1,4),25);
surf(double(x),double(y),double(z))
colormap Gray
axis equal
hold on
n = 1:nop;
plot3(P(n,1),P(n,2),P(n,3),'r*');
4(b) A MATLAB Program to evaluate the sphericity error using Least Square Method (Line-Search Strategy – Gauss-Newton Method).

% This program uses the Gauss-Newton Algorithm to fit set of data
% points in sphere. The data points should be stored in comma
% separated value file (csv). The first column in csv file
% represents x coordinate, second column represents y coordinate
% and third column represents of the point. This file is read by
% the program.

close all    % Remove all figures
clear all    % Remove all items from workspace, freeing up
             % system memory
clc          % Clear command window

% Prompts for selecting file
prompt = 'Enter the filename to fit sphere (*.csv): ';
str = input(prompt, 's');

% The data points are read in matrix P
P = csvread(str);

%Start clock to measure performance
%Start clock to measure performance

n0 = size(P, 1);    % Number of points in set
A = zeros(n0, 4);   % Initialization of matrix A, size (n x 3)
B = zeros(n0, 1);   % Initialization of matrix B, size (n x 1)
J = zeros(n0, 4);   % Initialization of matrix J, size (n x 3)

% Calculates A and B
for n=1:size(A,1)
    A(n,1) = 2*P(n,1);
    A(n,2) = 2*P(n,2);
    A(n,3) = 2*P(n,3);
    A(n,4) = -1;
    B(n) = P(n,1)*P(n,1)+P(n,2)*P(n,2)+P(n,3)*P(n,3);
end

% Calculates initial estimate of coordinate of center and radius
X = (A'*A)/(A'*B);
R = sqrt(X(1)*X(1)+X(2)*X(2)+X(3)*X(3)-X(4));

% Initialize matrix Di, size (n x 1). Stores value of distance from
% center of circle to the point
Di = zeros(n0, 1);

% Initialize matrix Ri, size (n x 1).
Di = zeros(n0, 1);

counter = 1;

for counter = 1:10
    % Calculate distance from center of the circle to nth point in
% set and also calculate Jacobian
    % for n=1:n0
        Di(n, 1) = sqrt((P(n,1) - X(1, 1))^2 + (P(n,2) - X(2,1))^2 + (P(n,3) - X(3))^2) - R;
    % end

    for n=1:size((P),1)
\[ J(n,1) = -(P(n,1)-X(1))/Di(n); \]
\[ J(n,2) = -(P(n,2)-X(2))/Di(n); \]
\[ J(n,3) = -(P(n,3)-X(3))/Di(n); \]
\[ J(n,4) = -1; \]
end

\[ Z = (J'*J)\backslash (J'*(-Di)); \]
\[ X = X + Z; \]
\[ R = R+Z(4); \]
end

\[ \text{sum\_sq\_err} = 0; \quad \% \text{Sum of Square of errors (di}^2) \]
for \( n = 1:nop \)
    \[ \text{sum\_sq\_err} = \text{sum\_sq\_err} + Di(n,1)^2; \]
end
\[ \text{Sphericity} = \max(Di) - \min(Di); \]
\[ \text{tElapsed} = \text{toc(tStart);} \quad \% \text{Stop clock to measure performance} \]
\[ \% \text{Display the result} \]
\[ \text{disp\_str\_1} = \text{sprintf('Coordinate of center of sphere is: \%0.4f \%0.4f', X(1,1), X(2,1), X(3,1));} \]
\[ \text{disp\_str\_2} = \text{sprintf('Radius of sphere is: \%0.4f', R)}; \]
\[ \text{disp\_str\_3} = \text{sprintf('Sum of square of error is: \%0.4f', sum\_sq\_err)}; \]
\[ \text{disp\_str\_4} = \text{sprintf('Circularity error is: \%0.6f', Sphericity)}; \]
\[ \text{disp\_str\_5} = \text{sprintf('The elapsed time is : \%0.6f', tElapsed)}; \]
\[ \text{disp(disp\_str\_1);} \]
\[ \text{disp(disp\_str\_2);} \]
\[ \text{disp(disp\_str\_3);} \]
\[ \text{disp(disp\_str\_4);} \]
\[ \text{disp(disp\_str\_5);} \]
\[ \% \text{Plot Sphere} \]
\[ [x,y,z]=\text{ellipsoid}(X(1,1),X(2,1),X(3,1),R,R,R,25); \]
\[ \text{surf(double(x),double(y),double(z))} \]
\[ \text{colormap Gray} \]
\[ \text{axis equal} \]
\[ \text{hold on} \]
\[ n = 1:nop; \]
\[ \text{plot(P(n,1),P(n,2),P(n,3),'r*');} \]
\[ \text{plot3(double(X(1)),double(X(2)),double(X(3)),'g*')} \]