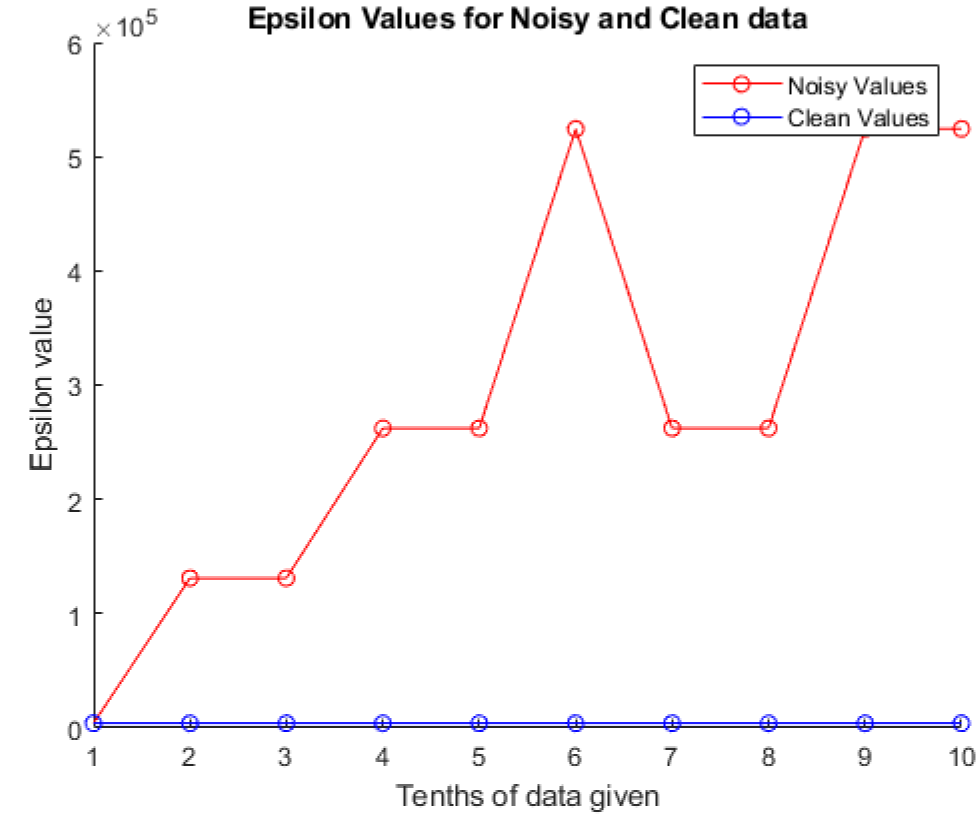
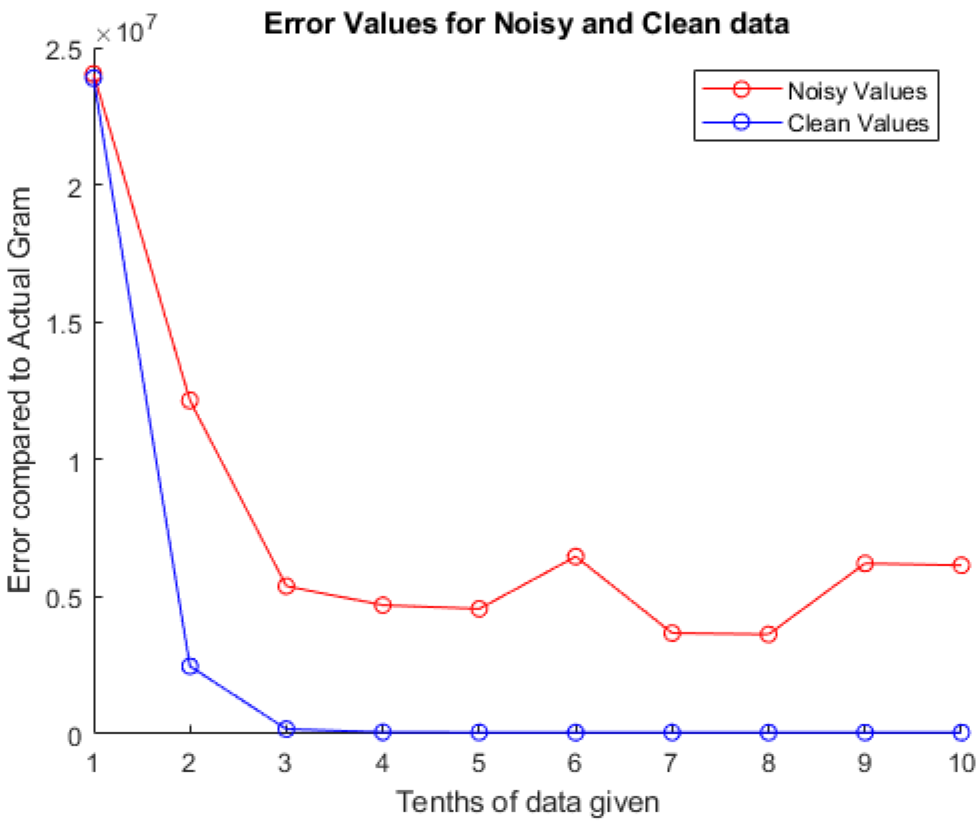
```
R_exact = readmatrix('Nodes1thru57_dist_exact.txt');
S_exact = R_exact.^2;
N = size(R_exact, 1);
[~,~,G_clean] = algo_1(R_exact, S_exact, N);

R_rounded = readmatrix('Nodes1thru57_dist.txt');
S_rounded = R_rounded.^2;
[~,~,G_noisy] = algo_1(R_rounded, S_rounded, N);


names = ['Noisy' + string(1:10) + 'Nodes1to57kn57_dist', string(1:10)
 + 'kn57Nodes1to57_exactdist'];
graphs = size(names, 2);
clean_errors = zeros([1, graphs]);
noisy_errors = zeros([1, graphs]);
epsilons = zeros([1, graphs]);
for j=1:graphs
    % Putting 2048 here because it's somewhat close to the epsilon
    % values that CVX deems 'good'
    [clean_errors(j), noisy_errors(j), epsilons(j)] = get_error(names(j),
 G_noisy, G_clean, 4096);
end

figure
hold on
plot(epsilons(1:10), 'ro-')
plot(epsilons(11:20), 'bo-')
title('Epsilon Values for Noisy and Clean data')
xlabel('Tenths of data given')
ylabel('Epsilon value')
legend(['Noisy Values'; 'Clean Values'])
hold off

figure
hold on
plot(noisy_errors(1:10), 'ro-')
plot(clean_errors(11:20), 'bo-')
title('Error Values for Noisy and Clean data')
xlabel('Tenths of data given')
ylabel('Error compared to Actual Gram')
legend(['Noisy Values'; 'Clean Values'])
hold off


m_list = 1./(((1:10)/10) * N*(N-1)/2);
b_noisy = (m_list' * m_list) \ (m_list' * noisy_errors(1:10));
b_clean = (m_list' * m_list) \ (m_list' * clean_errors(11:20));
figure
hold on
loglog(noisy_errors(1:10), 'ro-')
loglog(clean_errors(11:20), 'bo-')
axis tight
```

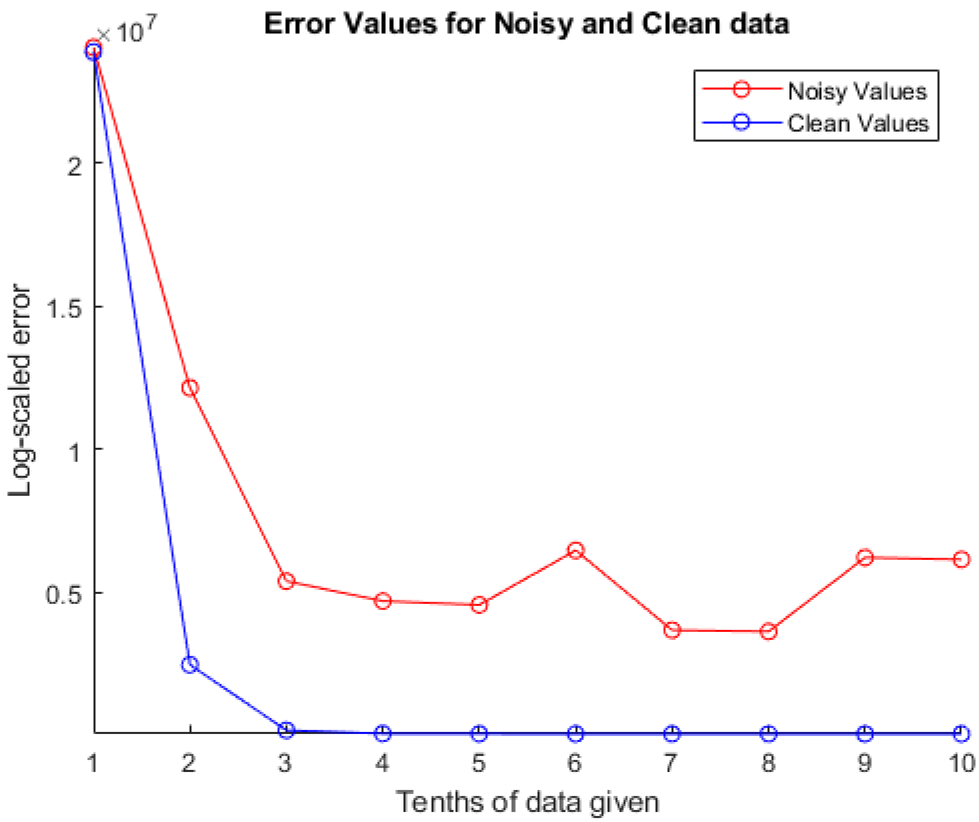
```
title('Error Values for Noisy and Clean data')
xlabel('Tenths of data given')
ylabel('Log-scaled error')
legend(['Noisy Values'; 'Clean Values'])
hold off

figure
hold on
semilogx(noisy_errors(1:10), 'ro-')
semilogx(clean_errors(11:20), 'bo-')
axis tight
title('Error Values for Noisy and Clean data')
xlabel('Tenths of data given')
ylabel('SemiLog-scaled error')
legend(['Noisy Values'; 'Clean Values'])
hold off


% The epsilon looks like it has an error rate similar to 1/m^alpha instead of
 e^-bm as the rate seems to be significantly higher at X=1 than at X=2.
% Epsilon seems to make a significant difference; when more and more
 restrictions are placed on the program it makes sense that it becomes harder
 to fully conform to every restriction. When epsilon is much larger, it
 becomes far more feasible.
% The feasiblity vs accuracy tradeoff is a massive one. As mentioned before,
 the fact that epsilon has such a large difference on the program is evident
 given the disparity between epsilons where we've obtained results.
% When epsilon is low, the program has a more difficult time finding a
 solution; that is, the program determines that a solution would not be found
 in any reasonable amount of time.

function [Error_clean, Error_noisy, epsilon] = get_error(name, G_noisy,
 G_clean, init_epsilon)
[N, ~, ~, E_ijs, D_tilde] = prep_distance('SparseGraphs\Sparse' + name
 + '.txt');

G = NaN(N);
epsilon = init_epsilon / 2;

while (isnan(G))
    epsilon = epsilon * 2;
    % TODO: FIX THIS PART FOR LATER
    % Want to get down the rest of the code before coming back
    G = perform_cvx(N, D_tilde, E_ijs, epsilon);
end

Error_clean = norm(G - G_clean, 'fro');
Error_noisy = norm(G - G_noisy, 'fro');
end

function G = perform_cvx(N, D_tilde, E_ijs, epsilon)
m = size(D_tilde, 1);
cvx_begin sdp quiet
variable G(N,N) semidefinite symmetric;
```

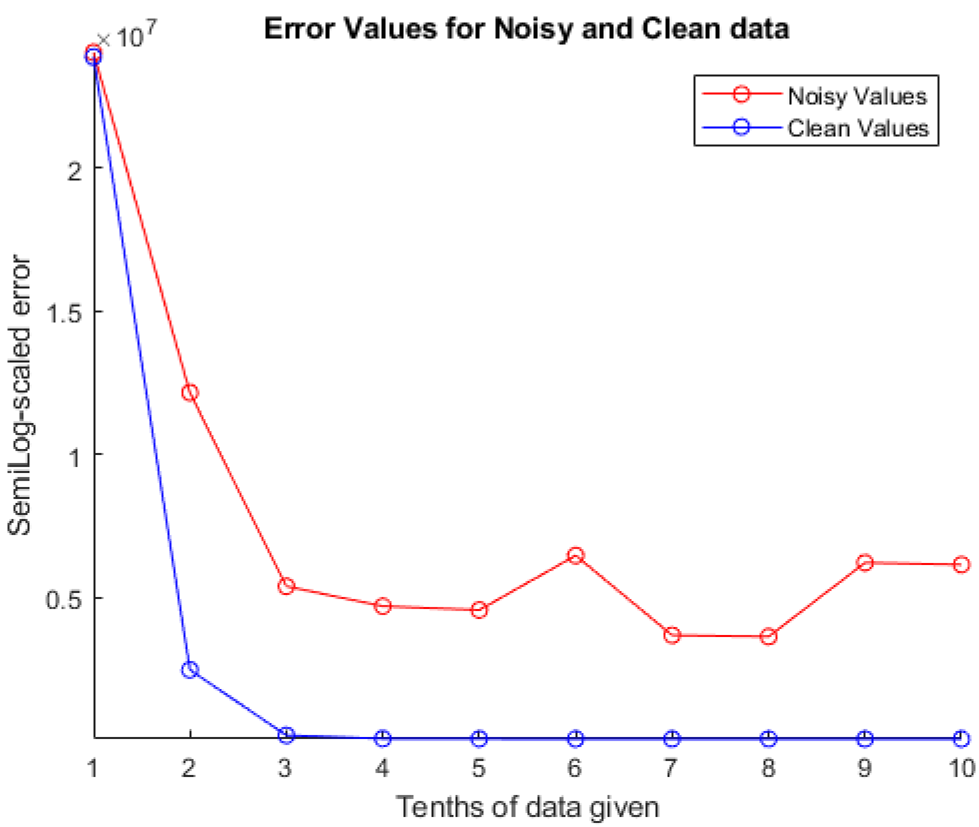
```
minimize trace(G);
subject to
G*ones(N, 1) == 0;
% We're looking to minimize the inner product of Ge_ij and e_ij and
% the random d distances
% e_ij = 1 at i, -1 at j, 0 else
% E_ijs should be this M times for each possible edge

% Need to make sure it only counts constraints that make sense
% That is, we need to ensure it only counts constraints that we have data
% for
% Otherwise, we get that the distance between two points has to be zero (or
% close enough to zero as to be zero)

% By doing this, we're making sure that we're only getting the distances
% that are non-zero
abs(diag(E_ijs'*G*E_ijs) - D_tilde') <= epsilon * ones(m, 1);

cvx_end
end

function [E, D_tilde] = fetch_vals(N, D)
% First get the number of possible edges
m = (N*N - N) / 2;
% Initialize E to all zeros
E = zeros([N, m]);
% Create a variable to store the distances
D_tilde = zeros([1, m]);
j = 1;
index = 1;

% Do it all with a single for loop (makes D easier)
for k=1:m
    % Add 1 to J and loop if it's at N + 1
    j = mod(j + 1, N + 1);
    % If it's zero, we've hit the end of a column
    if (j == 0)
        index = index + 1;
        j = 1;
    end

    % We don't really want all the duplicate i-j pairs, so remove those by
    % only making the first pair of i-j matter (instead of j-i)
    % This way, we only get the right number of i-j
    while (j <= index)
        j = mod(j + 1, N + 1);
    end
    % At the end, we then fill the values we want to fill
    % i is 1 and j is 0
    E(index, k) = 1;
    E(j, k) = -1;

    % Then, we assign the right distance from our data to the distance
    % array
```

```
    D_tilde(k) = D(j, index);
end
end


% Below are setup functions we don't really need to worry about.
function [N, M, arr] = read_to_arr(filename)
T = readtable(filename);
N = table2array(T(1, 1));
M = table2array(T(1, 2));
arr = table2array(T(2:M+1, :));
end

% Initially assign D to zero to allocate space and to make it easy to tell
% when a point has no value given (0 distances!??)
function [N, M, D, E, y] = prep_distance(filename)
[N, M, arr] = read_to_arr(filename);
D = zeros(N);
for index=1:M
    point = arr(index, 1:2);
    D(point(1), point(2)) = arr(index, 3);
    E(point(1), index) = 1;
    E(point(2), index) = -1;
    y(index) = arr(index, 3)^2;
end
D = D + D';
end

function [rho, upsilon, G] = algo_1(R, S, n)
% Use Algorithm 1 to create Gram G
rho = (1/(2 * n)) * sum(sum(S));
upsilon = (1/n) *  (S - rho*eye(n)) * ones(n, 1);
G = 0.5*upsilon*ones(n, 1)' + 0.5*ones(n, 1)*upsilon' - 0.5*S;
end
```


Epsilon Values for Noisy and Clean data
$\times 10^5$
Noisy Values
Clean Values
Epsilon value
Tenths of data given
0
1
2
3
4
5
6
7
8
9
10


Error Values for Noisy and Clean data
$\times 10^7$
Noisy Values
Clean Values
Error compared to Actual Gram
Tenths of data given
0
0.5
1
1.5
2
2.5
1
2
3
4
5
6
7
8
9
10


Error Values for Noisy and Clean data
×10^7
Noisy Values
Clean Values
2
1.5
1
0.5
Log-scaled error
1
2
3
4
5
6
7
8
9
10
Tenths of data given


Error Values for Noisy and Clean data
×10^7
Noisy Values
Clean Values
2
1.5
1
0.5
SemiLog-scaled error
1
2
3
4
5
6
7
8
9
10
Tenths of data given

Published with MATLAB® R2022b