CME 307 Optimization Project 1

March 19, 2018

Group: Austin Wang, Joe Higgins, Lawrence Moore

0.1 Introduction

The sensor network localization (SNL) problem is easy to understand yet surprisingly complex to solve. Given the distance between a set of anchors and sensors, with the true location of the anchors being known, the goal is to locate the sensors. Yet despite the straightforward statement of the problem, localizing the sensors in a wide variety of configurations presents a major challenge. In this project, we dive deep into two major components of the SNL problem under the assumption that all distance information is known:

- How the number of dimensions and anchors along with the starting point and learning rate affect the performance of the algorithms over a large number of randomly generated points.
- What algorithmic approach is overall best suited for the problem. To be explicit, we wrote extensible implementations of
  - Second Order Conic Problem (SOCP)
  - Semi-definite Programming (SDP), with and without noise
  - Simple non-linear least squares (NLLS)
  - Steepest Descent Method (SDM) with three different schemes for enforcing semi-definiteness

After running 50 simulations for each configuration of dimensions, anchors, and sensors, we found that SDP is the best approach once the number of anchors exceeds the number of dimensions - it always found the sensors. In the case that the number of anchors is less than the number of dimensions, NLLS slightly outperforms the other two. SOCP performs the worst under almost all circumstances.

Additionally, we investigated two other aspects of the SNL problem. One was introducing noise to the SDP formulation. Either the noise would be set to zero at the optimal point, or the resulting incorrect estimate did not lead NLLS to the optimal solution when initialized to that point. For the SDM, only the eigenvalue decomposition leads to an optimal point. Using only the top 6 eigenvalues did not lead to convergence. The LDL decomposition fails when the solution falls out of the feasible region.

0.2 Question 1: Comparing Average Performance of SOCP, SDP, and NLLS

In previous homeworks, our analysis of SOCP and SDP was mainly concerned with the relative positions of the sensors in the anchor-sensor system. As opposed to focusing on which specific

configurations allow the sensors to be localized for the three different methods, in this part of the project we explore the effects of dimensionality and number of anchors on the success rates of the methods.

0.2.1 Methodology

Our first step was providing an extensible implementation of the SOCP, SDP, and NLLS for a given dimension-number of anchors combination (d, a), where $d \in [1, 2, 3]$ and $a \in [2, 3, 4]$. We fix the number of sensors at 10. The functions run_socp, run_sdp, and run_nlls execute a single run of the methods for given input sensors and input anchors, and return the number of sensors that were located correctly.

Then, at the beginning of each of 50 trials, we randomly generate sensors and anchors whose coordinates are chosen from $(-2, 2)$ for a given dimension and a given number of anchors. Finally, we count the number of correctly located sensors for each method and display the average number of successes per iteration in the bar graphs that follow.

```
In [57]: # Import necessary modules
         %matplotlib inline
         import scipy as sp
         import cvxpy as cvx
         import numpy as np
         from scipy import random
         from scipy import linalg
         import matplotlib.pyplot as plt
         from scipy.linalg import eigh
```

0.2.2 Helper functions

General Helpers

```
In [5]: # Used for generating random anchors and sensors whose coordinates are in [-2,2]
        def generate_points(num, dim):
            return np.matrix(4 * np.random.random((num, dim)) - 2)

        # For generating the sensor-sensor distances and sensor-anchor distances
        def generate_ss_distances(sensors):
            return list(map(lambda s1:
                        list(map(lambda s2: np.linalg.norm(s1 - s2), sensors))
                    , sensors))

        def generate_sa_distances(sensors, anchors):
            return list(map(lambda s:
                        list(map(lambda a: np.linalg.norm(a - s), anchors))
                    , sensors))

        # For defining the the matrix-matrix dot product
        def sum_elem_product(A,B):
            return cvx.sum_entries(cvx.mul_elemwise(A, B))
```

```
        # For calculating the number of sensors that are correctly located
        def sensor_acc(x, true_sensors):
            num_correct = 0
            for i in range(true_sensors.shape[0]):
                if abs(np.linalg.norm(x[i,:]-true_sensors[i,:])) <= 10**-2:
                    num_correct += 1
            return num_correct
```

SDP Helpers

```
In [6]: #Make first set of constraint matrices (look like identity)
        def enforce_id(sensors):
            dim = np.shape(sensors)[1]
            num_sensors = np.shape(sensors)[0]
            matrices = []
            rhs = []
            for i in range(dim):
                new_matrix = np.zeros((dim+num_sensors, dim+num_sensors))
                new_matrix[i,i] = 1
                matrices.append(new_matrix)
                rhs.append(1)

            return (matrices, rhs)

        #Make second set of constraint matrices (symmetric holders)
        def enforce_id2(sensors):
            dim = np.shape(sensors)[1]
            num_sensors = np.shape(sensors)[0]
            matrices = []
            rhs = []
            for i in range(dim):
                for j in range(dim):
                    new_matrix = np.identity(dim)
                    if(j > i):
                        new_matrix[i,j] = 1
                        new_matrix[j,i] = 1
                        big_matrix = np.zeros((dim+num_sensors, dim+num_sensors))
                        big_matrix[0:dim,0:dim] = new_matrix
                        matrices.append(big_matrix)
                        rhs.append(dim)

            return (matrices, rhs)


        #Make third set of constraint matrices (anchors to sensors)
        def sensor_constraints(sensors):
            dim = np.shape(sensors)[1]
            num_sensors = np.shape(sensors)[0]
```

```
        d_ss = generate_ss_distances(sensors)
        matrices = []
        rhs = []
        zero_vec_dim = np.zeros(dim)
        for i in range(num_sensors):
            for j in range(i+1, num_sensors):
                zero_vec_num_s = np.zeros(num_sensors)
                zero_vec_num_s[i] = 1
                zero_vec_num_s[j] = -1

                new_vec = np.matrix(np.append(zero_vec_dim, zero_vec_num_s))

                new_matrix = np.dot(np.transpose(new_vec), new_vec)

                matrices.append(new_matrix)
                rhs.append(d_ss[i][j]**2)

        return (matrices, rhs)

    #Make fourth set of constraint matrices (sensors to sensors)
    def anchor_constraints(sensors, anchors):
        num_sensors = np.shape(sensors)[0]
        num_anchors = np.shape(anchors)[0]
        d_sa = generate_sa_distances(sensors, anchors)
        matrices = []
        rhs = []
        for i in range(num_anchors):
            for j in range(num_sensors):
                zero_vec_num_s = np.zeros(num_sensors)
                zero_vec_num_s[j] = -1

                new_vec = np.append(np.array(anchors[i,:]), np.array(zero_vec_num_s))
                new_vec = np.matrix(new_vec)
                new_matrix = np.dot(np.transpose(new_vec), new_vec)
                matrices.append(new_matrix)
                rhs.append(d_sa[j][i]**2)

        return (matrices, rhs)
```

NLLS Helpers

```
In [7]: # Gradient of objective
    def grad(sensors, anchors, sens2sens_dist, sens2anc_dist):
        dim = sensors.shape[1]
        sensor_distance_sum = np.zeros((1,dim))
        anchor_distance_sum = np.zeros((1,dim))
        gradient = np.zeros(sensors.shape)
```

```
            for i, sensor_i in enumerate(sensors):
                for j, sensor_j in enumerate(sensors):
                    if(i != j):
                        sensor_distance_sum += (np.linalg.norm(sensor_i - sensor_j)**2 - \
                                                np.linalg.norm(sens2sens_dist[i][j])**2) * \
                                               (sensor_i - sensor_j)

                for k, anchor_k in enumerate(anchors):
                    anchor_distance_sum += (np.linalg.norm(anchor_k - sensor_i)**2 - \
                                            np.linalg.norm(sens2anc_dist[i][k])**2) * \
                                           (sensor_i - anchor_k)

                gradient[i,:] = 8*sensor_distance_sum + 4*anchor_distance_sum

            return gradient
```

Individual Method Run Helpers

```
In [8]: # Returns the number of correctly located sensors for SOCP
        def run_socp(sensors, anchors):
            d_ss = generate_ss_distances(sensors)
            d_sa = generate_sa_distances(sensors, anchors)

            dim = sensors.shape[1]
            num_sensors = 10
            num_anchors = anchors.shape[0]
            x = cvx.Variable(num_sensors, dim)
            objective = cvx.Minimize(0)

            constraints  = []
            for i in range(num_sensors):
                x_i = x[i, :]
                for j in range(num_anchors):
                    constraints.append(cvx.norm(x_i - anchors[j]) <= d_sa[i][j])
                for j in range(num_sensors):
                    if i < j:
                        constraints.append(cvx.norm(x_i - sensors[j]) <= d_ss[i][j])

            prob = cvx.Problem(objective, constraints)
            result = prob.solve(solver = 'MOSEK')
            return sensor_acc(x.value, sensors)
```

```
In [9]: # Returns the number of correctly located sensors for SDP
        def run_sdp(sensors, anchors):

            # Make constraints
            A = enforce_id(sensors)
            B = enforce_id2(sensors)
```

```
    C = sensor_constraints(sensors)
    D = anchor_constraints(sensors, anchors)

    # Parameters
    dim = sensors.shape[1]
    num_sensors = 10
    num_anchors = anchors.shape[0]
    Z = cvx.Semidef(num_sensors + dim)

    constraints = []
    for id_constraint, rhs in zip(A[0], A[1]):
        constraints.append(sum_elem_product(id_constraint, Z) == rhs)
    for id_constraint2, rhs in zip(B[0], B[1]):
        constraints.append(sum_elem_product(id_constraint2, Z) == rhs)
    for sensor_constraint, rhs in zip(C[0], C[1]):
        constraints.append(sum_elem_product(sensor_constraint, Z) == rhs)
    for anchor_constraint, rhs in zip(D[0], D[1]):
        constraints.append(sum_elem_product(anchor_constraint, Z) == rhs)

    objective = cvx.Minimize(0)
    prob = cvx.Problem(objective, constraints)
    result = prob.solve(solver = 'MOSEK')
    sol = np.transpose(Z[0:dim, dim:dim+num_sensors].value)

    return sensor_acc(sol, sensors)
```

```
In [10]: def run_nlls(sensors, anchors):
    check = 1000
    max_iter = 10000
    k = 0

    # Parameters and initial data
    dim = sensors.shape[1]
    num_sensors = 10
    num_anchors = anchors.shape[0]
    d_ss = generate_ss_distances(sensors)
    d_sa = generate_sa_distances(sensors, anchors)

    # Initial sensors guess

    # RESULTS QUITE SENSITIVE TO INITIAL GUESS AS WELL
    sensors_0 = np.zeros(sensors.shape)
    sensors_k = sensors_0


    # Iteration
    alpha = .0001 # Result quite sensitive to alpha; chose from empirical evidence
    while check > 10**-6 and k < max_iter:
```

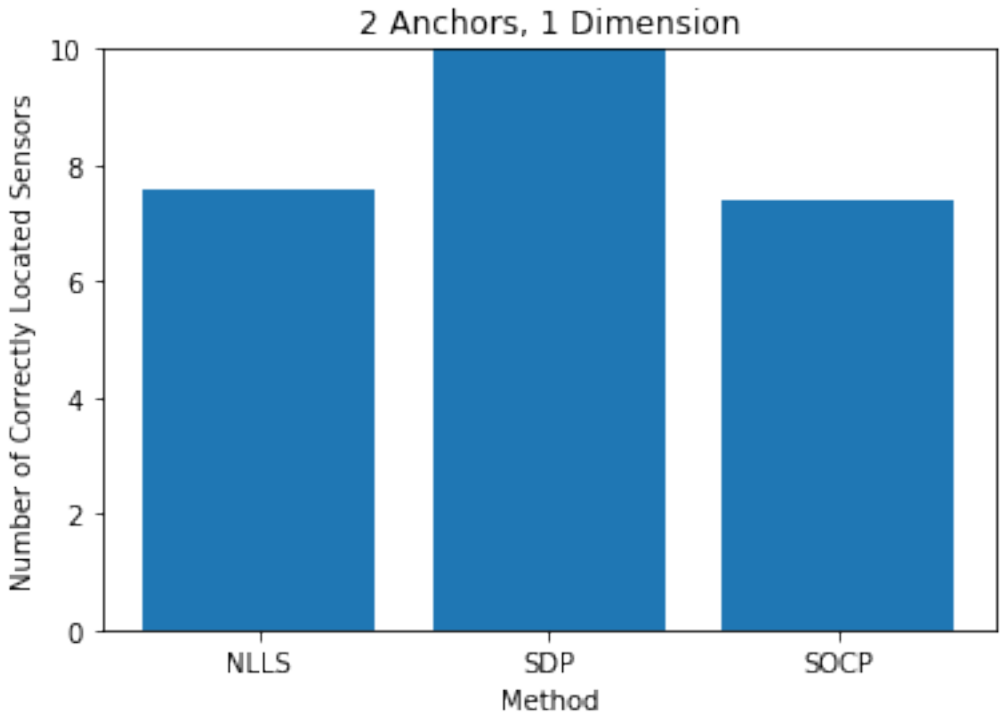
```
            sensors_k1 = sensors_k - alpha * grad(sensors_k, anchors, d_ss, d_sa)
            check = np.linalg.norm(sensors_k1 - sensors_k)
            sensors_k = sensors_k1
            k = k+1
        return sensor_acc(sensors_k1, sensors)
```

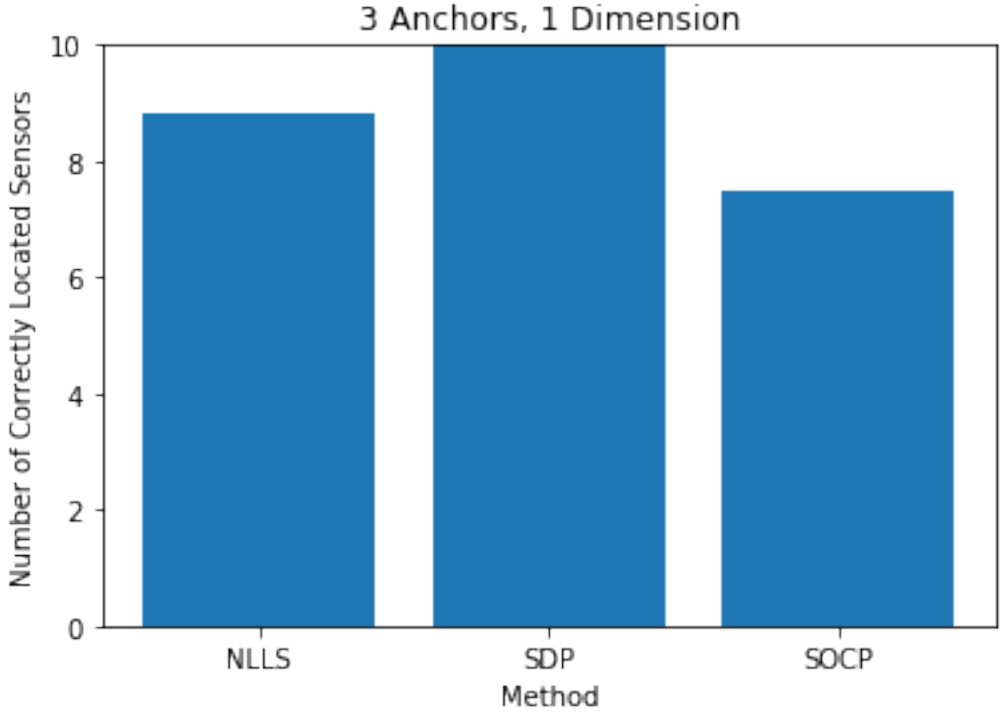
In [11]:
```
# Function that takes in a given dimension dim in [1,2,3] and outputs
# a tuple of 3 lists of lists. Each of these three outer lists corresponds
# to the data for SOCP, SDP, and Nonlinear Least Squares. The inner lists each
# correspond to the number of anchors in [2,3,4]. The values in each of these inner
# lists are the average number of sensors out of the 10 that were located correctly
# out of 50 trials
def execute_solvers(dim):
    SOCP_vals = []
    SDP_vals = []
    NLLS_vals = []
    for anc_i in num_anchors:
        socp_sum = 0
        sdp_sum = 0
        nlls_sum = 0
        for j in range(50):
            # Initialize data for a given trial
            sensors = generate_points(10, dim)
            anchors = generate_points(anc_i, dim)

            socp_sum += run_socp(sensors, anchors)
            sdp_sum += run_sdp(sensors, anchors)
            nlls_sum += run_nlls(sensors, anchors)
        SOCP_vals.append(socp_sum/50)
        SDP_vals.append(sdp_sum/50)
        NLLS_vals.append(nlls_sum/50)
        print('Finished One Anchor Iteration')

    return (SOCP_vals, SDP_vals, NLLS_vals)
```

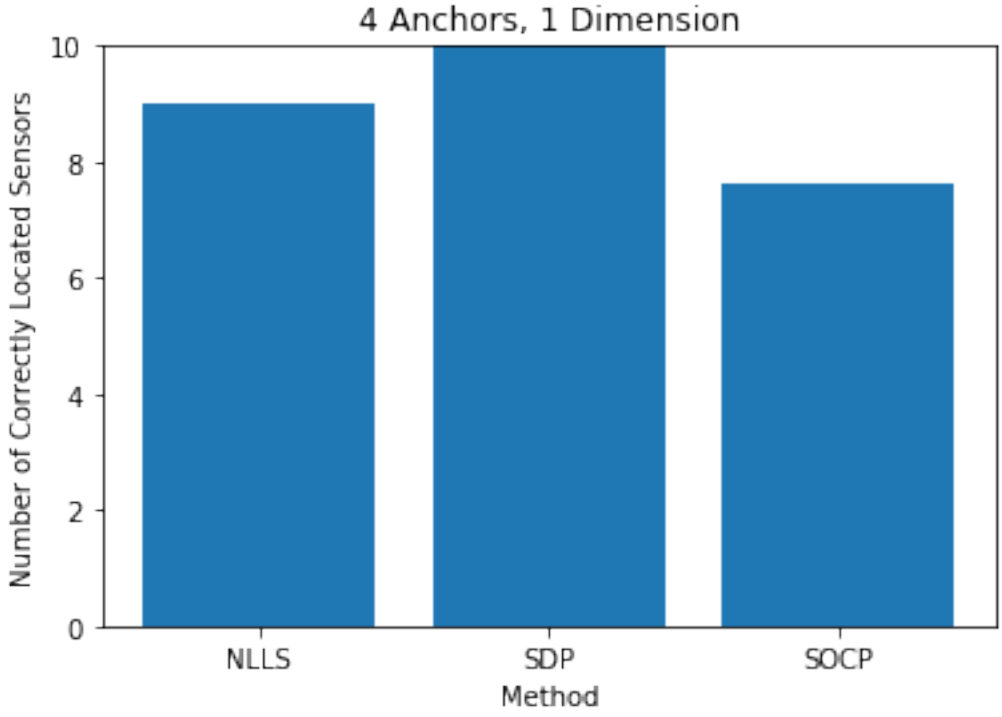
In [10]:
```
num_anchors = [2,3,4]

# Results - uncomment the following cells to run solvers
# dim1 = execute_solvers(1)
# dim2 = execute_solvers(2)
# dim3 = execute_solvers(3)

# Results of our simulation (if you don't want to rerun code)
dim1 = ([7.38, 7.5, 7.62], [10.0, 10.0, 10.0], [7.6, 8.8, 9.0])
dim2 = ([2.76, 3.06, 3.6], [0.06, 10.0, 10.0], [3.82, 7.0, 7.82])
dim3 = ([0.62, 0.94, 1.16], [0.0, 0.16, 10.0], [0.0, 2.68, 4.5])
```

In [47]:
```
# 1 Dimension
```

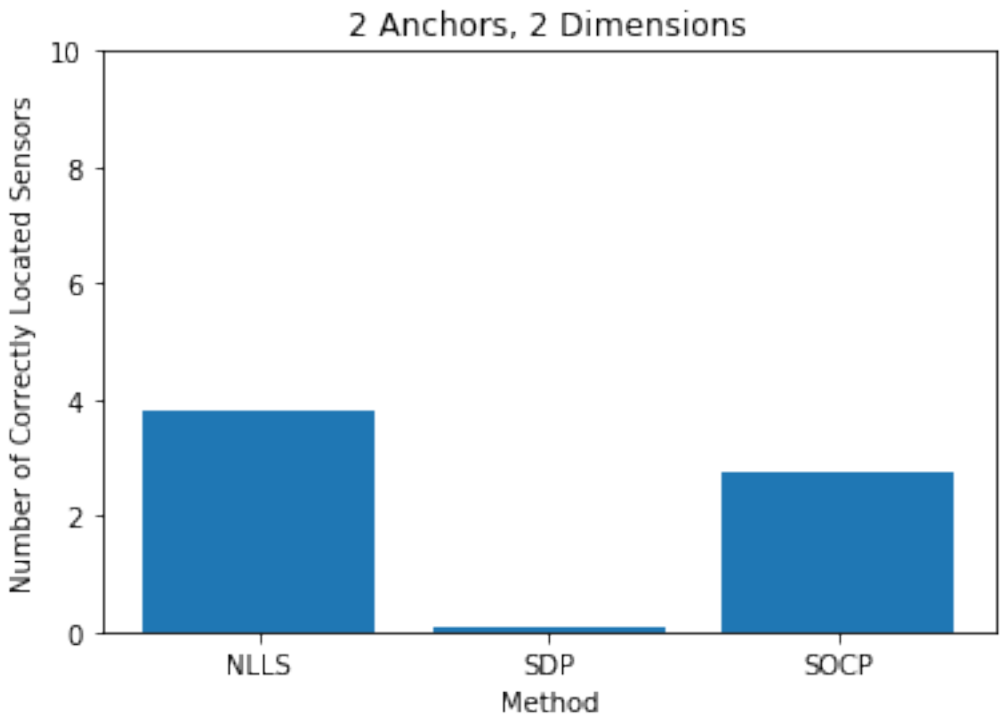
```
# 2 anchors
plt.figure()
plt.bar(['SOCP','SDP','NLLS'],[i[0] for i in dim1])
plt.title('2 Anchors, 1 Dimension')
plt.xlabel('Method')
plt.ylabel('Number of Correctly Located Sensors')
plt.ylim((0,10))
plt.savefig('anc2_dim1.jpg')

# 3 anchors
plt.figure()
plt.bar(['SOCP','SDP','NLLS'],[i[1] for i in dim1])
plt.title('3 Anchors, 1 Dimension')
plt.xlabel('Method')
plt.ylabel('Number of Correctly Located Sensors')
plt.ylim((0,10))
plt.savefig('anc3_dim1.jpg')

# 4 anchors
plt.figure()
plt.bar(['SOCP','SDP','NLLS'],[i[2] for i in dim1])
plt.title('4 Anchors, 1 Dimension')
plt.xlabel('Method')
plt.ylabel('Number of Correctly Located Sensors')
plt.ylim((0,10))
plt.savefig('anc4_dim1.jpg')
```

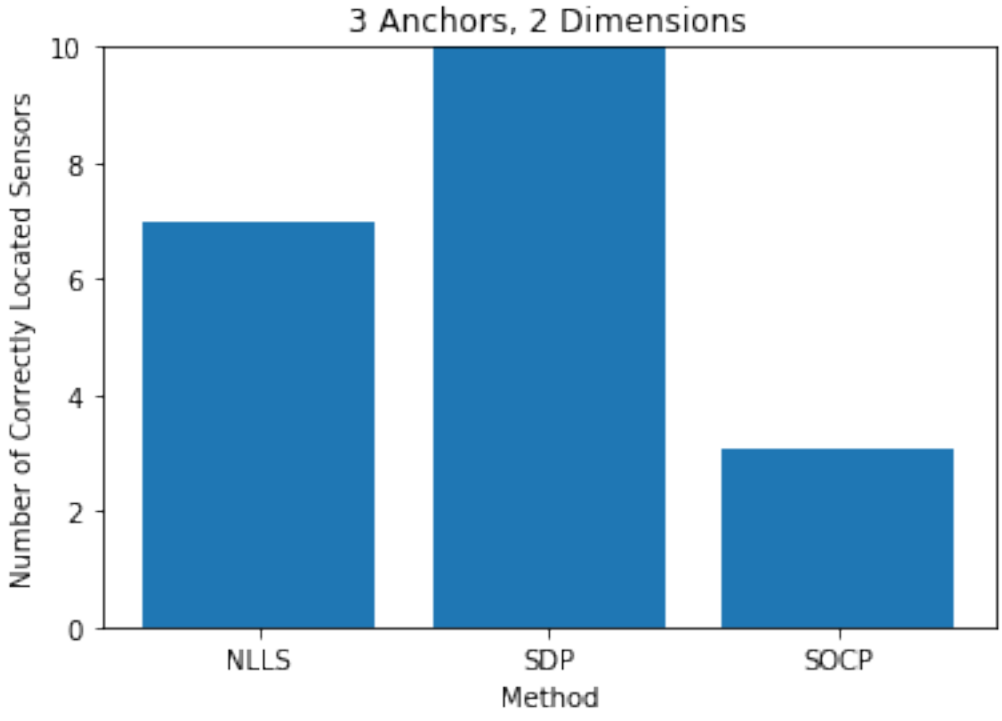


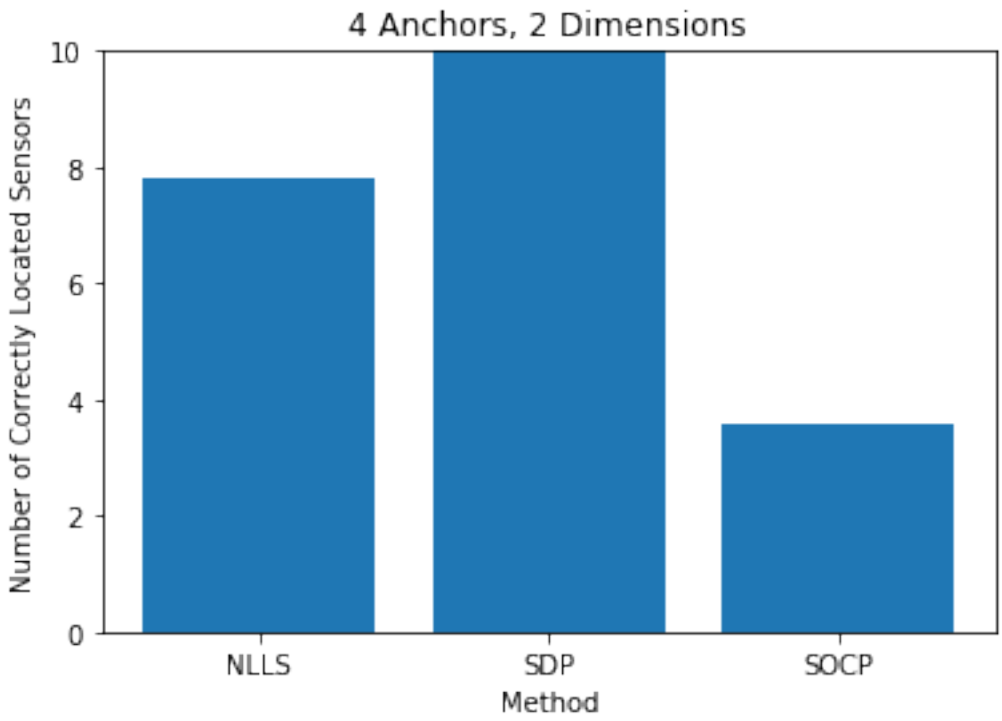

3 Anchors, 1 Dimension
Number of Correctly Located Sensors
0
2
4
6
8
10
NLLS
SDP
SOCP
Method


4 Anchors, 1 Dimension
Number of Correctly Located Sensors
0
2
4
6
8
10
NLLS
SDP
SOCP
Method

In [48]:
```
# 2 Dimensions

    # 2 anchors
    plt.figure()
    plt.bar(['SOCP','SDP','NLLS'],[i[0] for i in dim2])
    plt.title('2 Anchors, 2 Dimensions')
    plt.xlabel('Method')
    plt.ylabel('Number of Correctly Located Sensors')
    plt.ylim((0,10))
    plt.savefig('anc2_dim2.jpg')

    # 3 anchors
    plt.figure()
    plt.bar(['SOCP','SDP','NLLS'],[i[1] for i in dim2])
    plt.title('3 Anchors, 2 Dimensions')
    plt.xlabel('Method')
    plt.ylabel('Number of Correctly Located Sensors')
    plt.ylim((0,10))
    plt.savefig('anc3_dim2.jpg')

    # 4 anchors
    plt.figure()
    plt.bar(['SOCP','SDP','NLLS'],[i[2] for i in dim2])
    plt.title('4 Anchors, 2 Dimensions')
    plt.xlabel('Method')
    plt.ylabel('Number of Correctly Located Sensors')
    plt.ylim((0,10))
    plt.savefig('anc4_dim2.jpg')
```

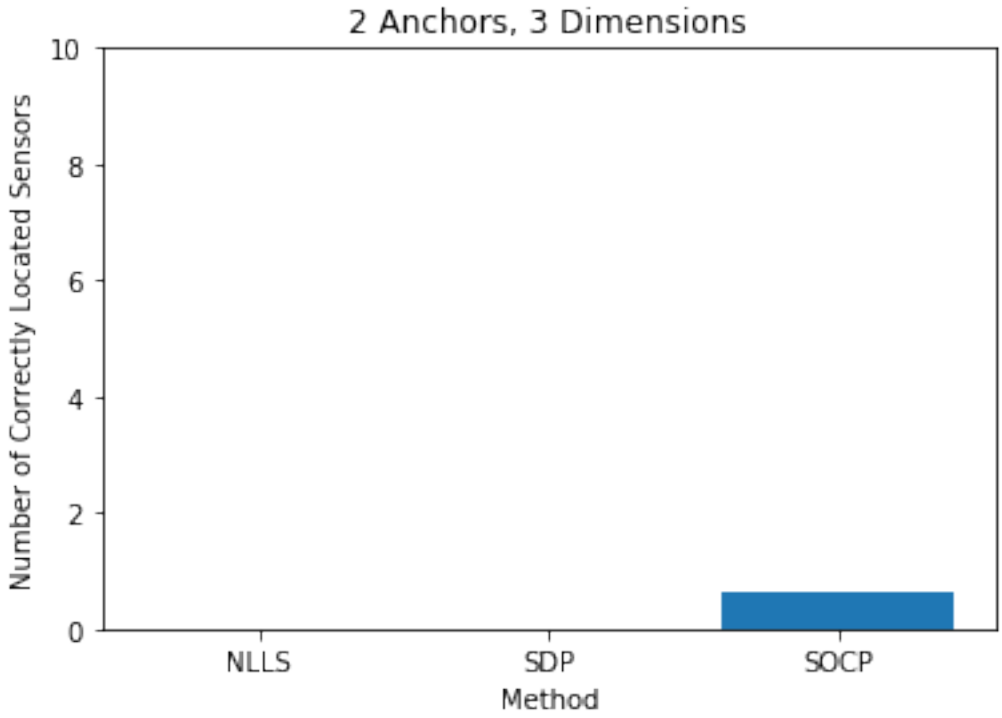

2 Anchors, 2 Dimensions
Number of Correctly Located Sensors
0
2
4
6
8
10
NLLS
SDP
SOCP
Method

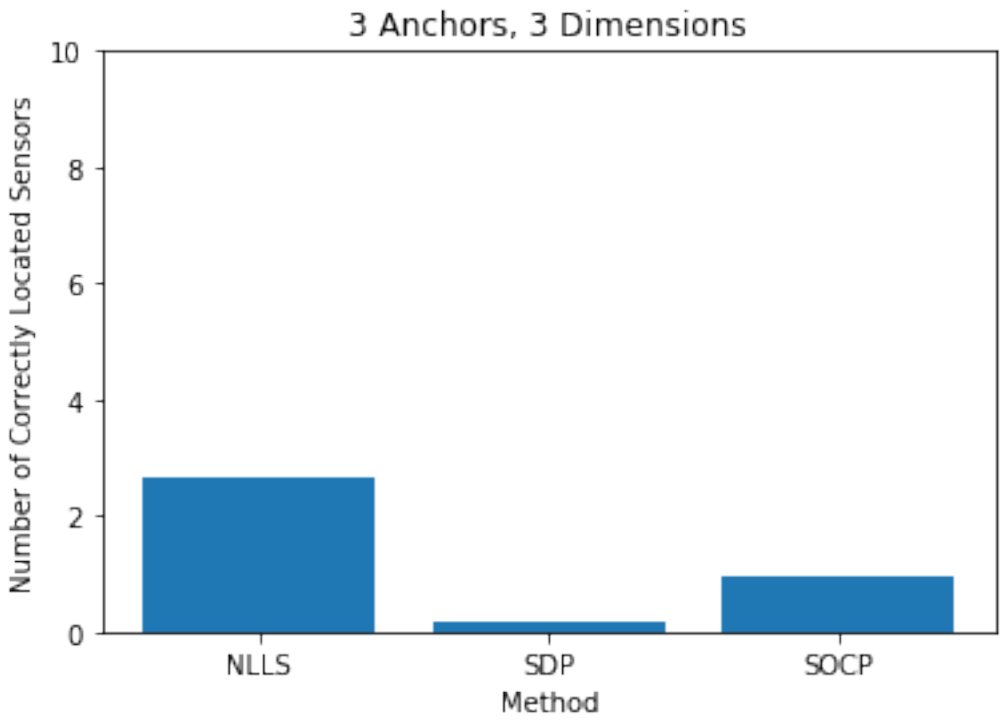

3 Anchors, 2 Dimensions
Number of Correctly Located Sensors
0
2
4
6
8
10
NLLS
SDP
SOCP
Method

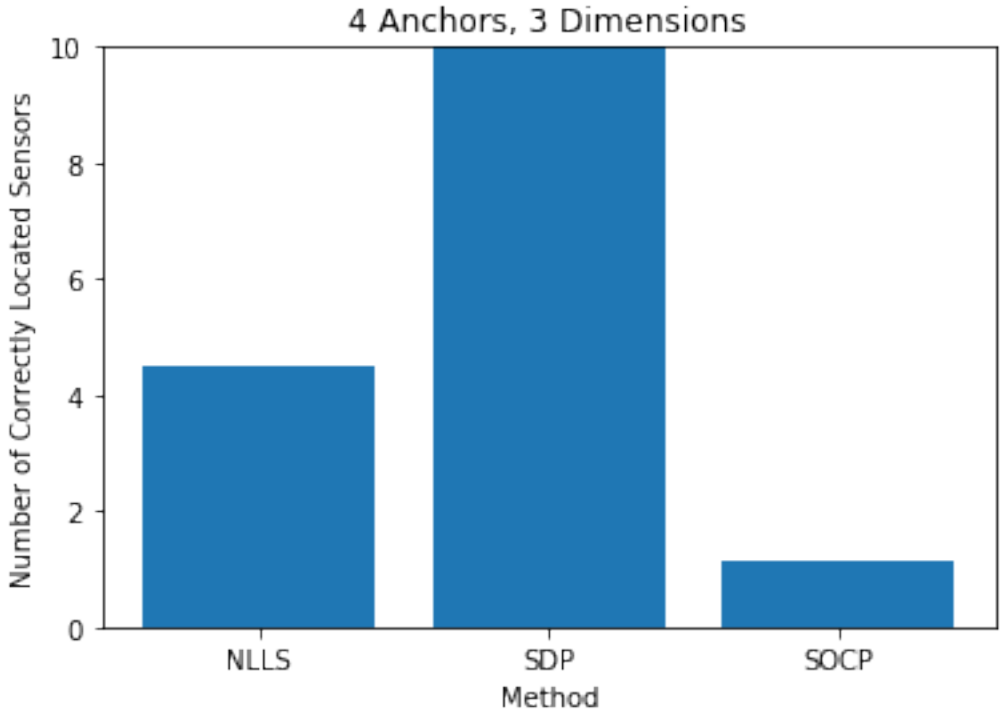


In [49]:
```
# 3 Dimensions

# 2 anchors
plt.figure()
plt.bar(['SOCP','SDP','NLLS'],[i[0] for i in dim3])
plt.title('2 Anchors, 3 Dimensions')
plt.xlabel('Method')
plt.ylabel('Number of Correctly Located Sensors')
plt.ylim((0,10))
plt.savefig('anc2_dim3.jpg')

# 3 anchors
plt.figure()
plt.bar(['SOCP','SDP','NLLS'],[i[1] for i in dim3])
plt.title('3 Anchors, 3 Dimensions')
plt.xlabel('Method')
plt.ylabel('Number of Correctly Located Sensors')
plt.ylim((0,10))
plt.savefig('anc3_dim3.jpg')

# 4 anchors
plt.figure()
plt.bar(['SOCP','SDP','NLLS'],[i[2] for i in dim3])
plt.title('4 Anchors, 3 Dimensions')
```

```
plt.xlabel('Method')
plt.ylabel('Number of Correctly Located Sensors')
plt.ylim((0,10))
plt.savefig('anc4_dim3.jpg')
```




3 Anchors, 3 Dimensions
Number of Correctly Located Sensors
0
2
4
6
8
10
NLLS
SDP
SOCP
Method


4 Anchors, 3 Dimensions
Number of Correctly Located Sensors
0
2
4
6
8
10
NLLS
SDP
SOCP
Method

0.2.3 Discussion

For a given dimension, the number of anchors heavily influences which algorithm performs the best. When the number of anchors is less than the number of dimensions, SDP performs the worst, with NLLS performing the best under this condition. However, once the number of anchors exceeds the number of dimensions, SDP finds all ten sensors in every case. Note that we generated the sensors and anchors randomly in a range of 4 across each dimension, which captures instances of the sensors falling inside the convex hull of the anchors in addition to falling outside. SOCP's performance remains essentially constant across the number of anchors, as it would seem the number of points that fall within the convex hull remains roughly the same. NLLS does better with more anchors, though the increase in performance is not as drastic as with SDP.

As the number of dimensions increased, the performance of all three algorithms decreased. This was particularly the case with SOCP, where the average number of anchors found for dimension 3 was only around 1. As we saw in the previous homeworks, the convex hull of the anchors plays a critical role in whether the location of the sensors is found for SOCP. Therefore, it makes sense that the increase in dimensions leads to more sensors outside the convex hull, and thus a worse performance overall.

Changing the focus from dimensionality to initial conditions, we found that the success rate of NLLS is highly dependent on the starting point and the learning rate. In our particular simulations, we used a constant starting point of the origin. We saw that for a given run of the NLLS gradient descent algorithm, there were three possible cases. In the first case, the NLLS was able to exactly locate all 10 sensors. In the second case, the norm of the gradient became quite close to 0, but the algorithm did not converge to the true sensor locations. In the final case, the norm of the gradient blew up. Which case we encountered appears to be directly dependent on the starting point and the learning rate. Although we used the origin as a fixed starting point in our simulations, we also tested starting points that were very near the true sensor locations and very far. For starting points close to the true sensor location, we almost always encountered case 1 if we chose our learning rate small enough. As the starting points became further and further from the true sensor locations, we tended to encounter case 2 more often. In addition, we needed to increase the learning rate to achieve a reasonable convergence speed, but this could lead to an explosion of the gradient, given that we used a constant learning rate in our simulations (i.e., no backtracking). To achieve a reasonable convergence rate using a constant learning rate, and to increase the frequency of convergence to the true sensor locations, we found that a good learning rate was 0.0001.

0.3 Visualization of Points

```
In [12]: def get_SOCP_result(sensors, anchors):
             d_ss = generate_ss_distances(sensors)
             d_sa = generate_sa_distances(sensors, anchors)

             x = cvx.Variable(num_sensors, dim)
             objective = cvx.Minimize(0)

             constraints  = []
             for i in range(num_sensors):
                 x_i = x[i, :]
                 for j in range(num_anchors):
```

```
                    constraints.append(cvx.norm(x_i - anchors[j]) <= d_sa[i][j])
                for j in range(num_sensors):
                    if i < j:
                        constraints.append(cvx.norm(x_i - sensors[j]) <= d_ss[i][j])

        prob = cvx.Problem(objective, constraints)
        result = prob.solve(solver = 'MOSEK')
        #print("SOCP complete")
        return x.value
```

In [14]:
```
def get_SDP_result(sensors, anchors):

    A = enforce_id(sensors)
    B = enforce_id2(sensors)
    C = sensor_constraints(sensors)
    D = anchor_constraints(sensors, anchors)

    Z = cvx.Semidef(num_sensors + dim)

    constraints = []
    for id_constraint, rhs in zip(A[0], A[1]):
        constraints.append(sum_elem_product(id_constraint, Z) == rhs)
    for id_constraint2, rhs in zip(B[0], B[1]):
        constraints.append(sum_elem_product(id_constraint2, Z) == rhs)
    for sensor_constraint, rhs in zip(C[0], C[1]):
        constraints.append(sum_elem_product(sensor_constraint, Z) == rhs)
    for anchor_constraint, rhs in zip(D[0], D[1]):
        constraints.append(sum_elem_product(anchor_constraint, Z) == rhs)

    objective = cvx.Minimize(0)
    prob = cvx.Problem(objective, constraints)
    result = prob.solve(solver = 'MOSEK')

    #print("SDP complete")
    return np.transpose(Z[0:dim, dim:dim+num_sensors].value)
```

In [15]:
```
def get_NLLS_matches(sensors, anchors):
    check = 1000
    max_iter = 10000
    k = 0

    # Initial sensors guess
    sensors_0 = generate_points(num_sensors, dim)
    sensors_k = sensors_0

    # iteration
    alpha = .0001
    while check > 10**-8 and k < max_iter:
```

```
            sensors_k1 = sensors_k - alpha * grad(sensors_k)
            check = np.linalg.norm(sensors_k1 - sensors_k)
            sensors_k = sensors_k1
        #    if(k % 2000 == 0):
        #        print("NLLS k:" + str(k))
        #    k = k+1
        #
        #print("NLLS complete")
        return sensors_k

    def grad(X):
        sensor_distance_sum = np.zeros((1,X.shape[1]))
        anchor_distance_sum = np.zeros((1,X.shape[1]))
        gradient = np.zeros(X.shape)

        for i, sensor_i in enumerate(X):
            for j, sensor_j in enumerate(X):
                if(i != j):
                    sensor_distance_sum += (np.linalg.norm(sensor_i - sensor_j)**2 - \
                                            np.linalg.norm(sensors[i,:] - sensors[j,:])**
                                           (sensor_i - sensor_j)

            for k, anchor_k in enumerate(anchors):
                anchor_distance_sum += (np.linalg.norm(anchor_k - sensor_i)**2 - \
                                        np.linalg.norm(anchors[k,:] - sensors[i,:])**2) *
                                       (sensor_i - anchor_k)

            gradient[i,:] = 8*sensor_distance_sum + 4*anchor_distance_sum

        return gradient
```

In [16]:
```
def get_sensor_matches(sensors, results):
    matches = []
    for sensor, result in zip(sensors, results):
        matches.append(np.linalg.norm(sensor - result) < 10**(-4))
    return np.array(matches)
```

In [17]:
```
def plot_results(anchors, sensors, results):
    anchor_x = np.array(anchors[:,0])
    anchor_y = np.array(anchors[:,1])

    matches = get_sensor_matches(sensors,results)

    matched_sensors_x = np.array(sensors[matches,0])
    matched_sensors_y = np.array(sensors[matches,1])
    unmatched_sensors_x = np.array(sensors[~matches,0])
    unmatched_sensors_y = np.array(sensors[~matches,1])
```

```
    fig, out_plot = plt.subplots()

    out_plot.scatter(x=anchor_x, y=anchor_y, c="g", alpha=0.5)
    out_plot.scatter(x=matched_sensors_x, y=matched_sensors_y, c="b", alpha=0.5)
    out_plot.scatter(x=unmatched_sensors_x, y=unmatched_sensors_y, c="r", alpha=0.5)
```

In [18]:
```
# General, hardcoded setup
dim = 2
num_anchors = 4
num_sensors = 10
```

0.3.1 Structure of individual trials

We now present plots showing which sensors were able to be located and which were not able to be located for several static configurations of anchors. This is presented to give an intuition for what's going on under the hood of the averages of each method (SDP, SOCP, NLLS) in the discussion above. We will present 2 examples with 4 anchors in 2 dimensions.

- Green: anchor
- Blue: sensor located by method
- Red: sensor not located by method

Example 1

- SOCP: able to find sensors located within the convex hull of the anchors
- SDP: able to find all sensors
- NLLS: able to find all sensors

Example 2

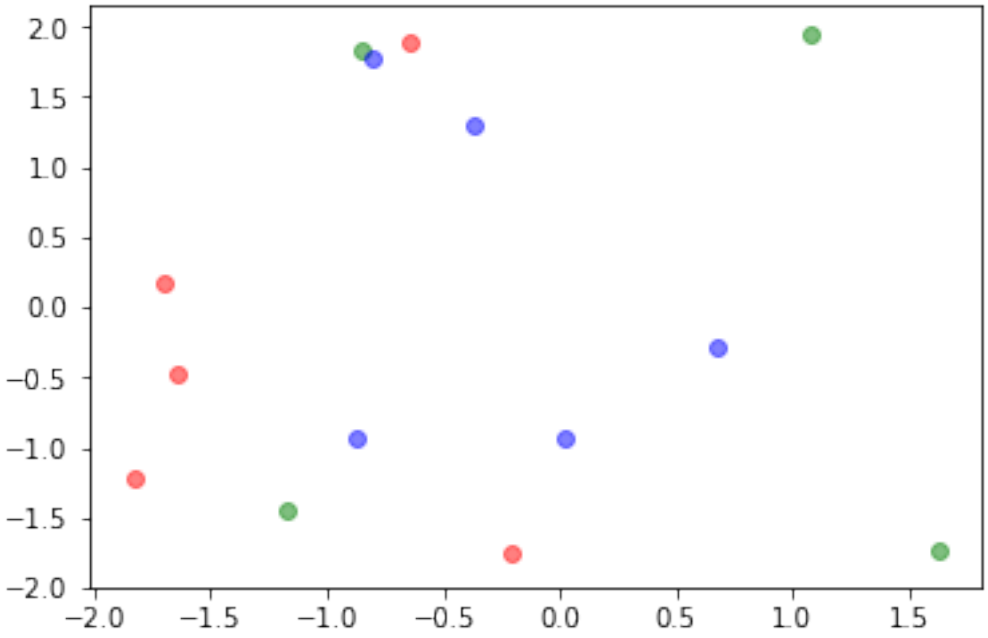
- SOCP: able to find sensors located within the convex hull of the anchors, and one outside
- SDP: able to find all sensors
- NLLS: unable to converge and finds no sensors

0.3.2 Conclusion

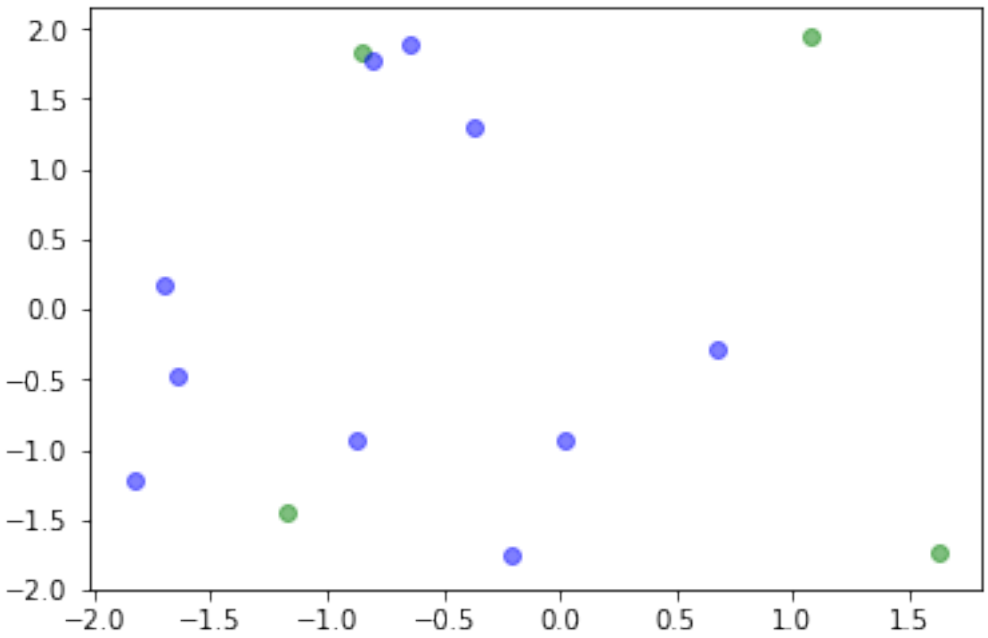
- SOCP: typically finds a subset of the points based on the configuration of the sensors. It will generally locate all the sensors within the convex hull of the anchors, and occasionally locate some sensors outside of the convex hull
- SDP: always locates all sensors, regardless of their location when the number of anchors is greater than the dimension
- NLLS: the average performance is between finding every sensor and finding no sensors. However, it is not because it typically finds that many sensors: the average we observe is due to combining many trials where the problem either finds all of the sensors, or none of the sensors.

In [20]:
```
#Example 1: SOCP inside convex hull border, SDP everywhere, NLLS everywhere
np.random.seed(seed=31)
anchors = generate_points(num_anchors, dim)
```

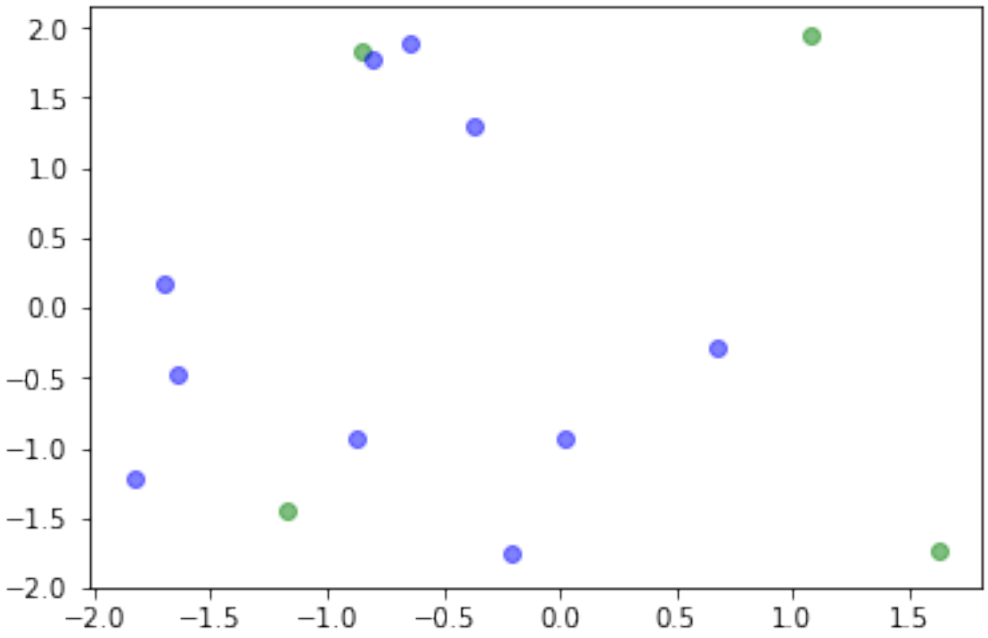
```
sensors = generate_points(num_sensors, dim)

SOCP_result = get_SOCP_result(sensors, anchors)
SDP_result = get_SDP_result(sensors, anchors)
NLLS_result = get_NLLS_matches(sensors, anchors)
plot_results(anchors, sensors, SOCP_result)
```



In [23]:
```
plot_results(anchors, sensors, SDP_result)
```

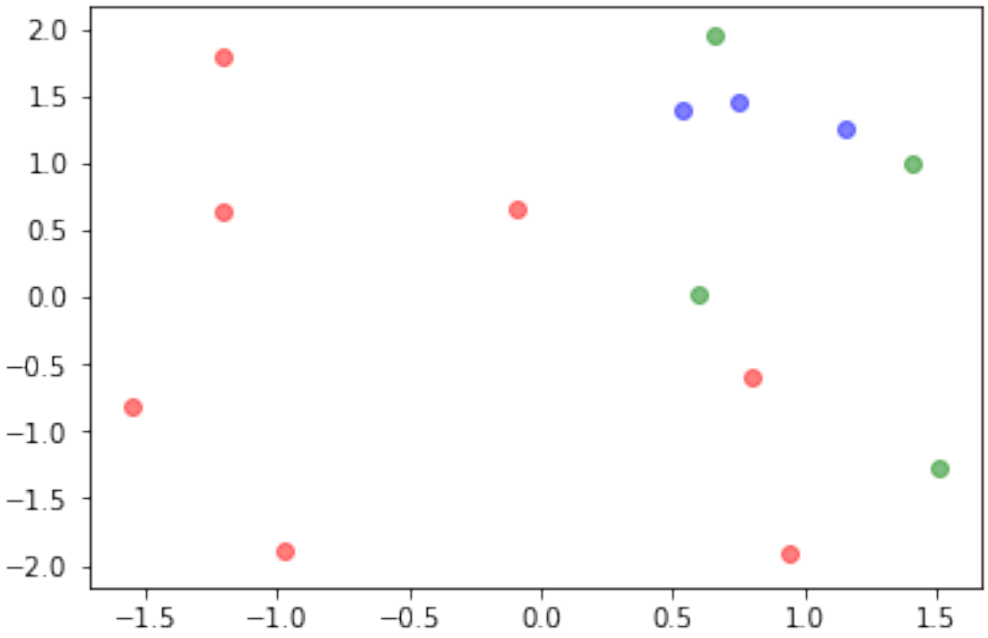


In [24]:
```
plot_results(anchors, sensors, NLLS_result)
```

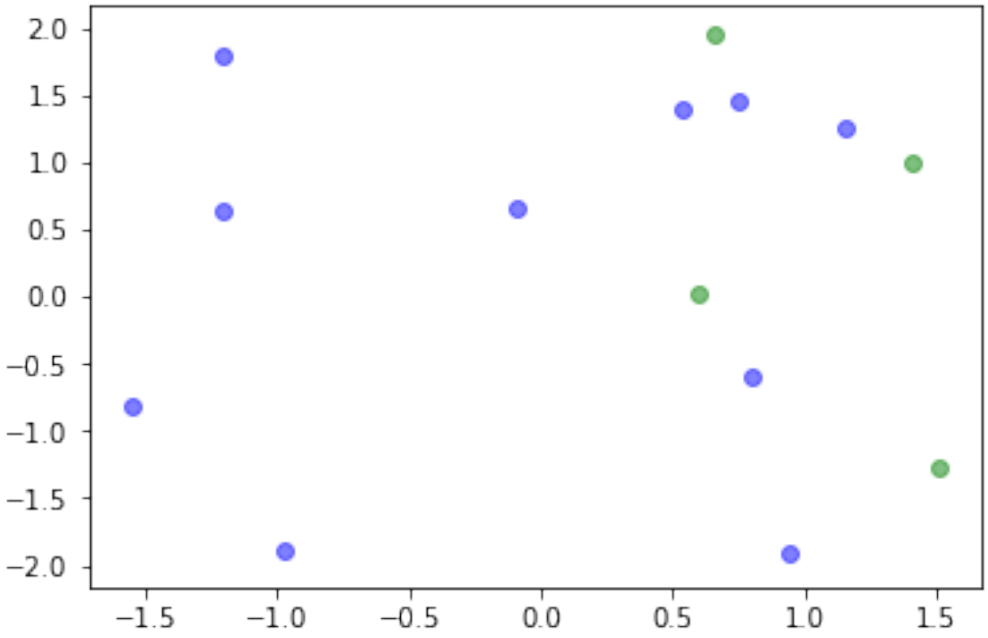


In [25]:
```
#Example 2: SOCP near convex hull border, SDP everywhere, NLLS not converge
np.random.seed(seed=18)
```

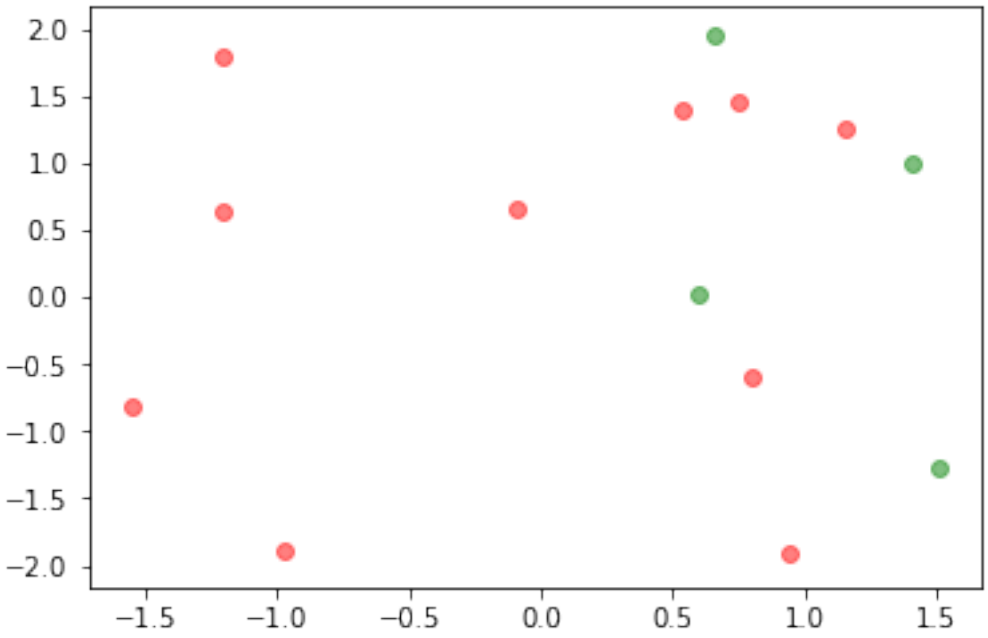
```
anchors = generate_points(num_anchors, dim)
sensors = generate_points(num_sensors, dim)

SOCP_result = get_SOCP_result(sensors, anchors)
SDP_result = get_SDP_result(sensors, anchors)
NLLS_result = get_NLLS_matches(sensors, anchors)
plot_results(anchors, sensors, SOCP_result)
```



In [26]:
```
plot_results(anchors, sensors, SDP_result)
```



In [27]: `plot_results(anchors, sensors, NLLS_result)`



0.4 Question 2: SDP with Noise

In the event that the sensor readings are noisy, it is useful to introduce a corresponding representation of the noise to our objective function and attempt to minimize it. This makes the process slightly more robust. In our case, when no noise was present, the noise terms were exactly set to zero as expected.

```
In [34]: num_sensors = 10
         num_anchors = 4
         dim = 2

         Z = cvx.Semidef(num_sensors + dim)
         anchors = generate_points(num_anchors, dim)
         sensors = generate_points(num_sensors, dim)

In [35]: # order of indexing: (1,2), (1,3), (1,4)... (9,10)
         delta_p_ss = cvx.Variable((int) (num_sensors*(num_sensors-1)/2))
         delta_pp_ss = cvx.Variable((int) (num_sensors*(num_sensors-1)/2))

         # order of indexing: (anchor, sensor), primary sort on anchors
         delta_p_sa = cvx.Variable(num_anchors * num_sensors)
         delta_pp_sa = cvx.Variable(num_anchors * num_sensors)

In [36]: constraints = []

         A = enforce_id(sensors)
         B = enforce_id2(sensors)
         C = sensor_constraints(sensors)
         D = anchor_constraints(sensors, anchors)

         constraints.append(delta_p_ss >= 0)
         constraints.append(delta_pp_ss >= 0)
         constraints.append(delta_p_sa >= 0)
         constraints.append(delta_pp_sa >= 0)

         # add the noise through the presence of dp and dpp
         for id_constraint, rhs in zip(A[0], A[1]):
             constraints.append(sum_elem_product(id_constraint, Z) == rhs)
         for id_constraint2, rhs in zip(B[0], B[1]):
             constraints.append(sum_elem_product(id_constraint2, Z) == rhs)
         for sensor_constraint, rhs, dp, dpp in zip(C[0], C[1], delta_p_ss, delta_pp_ss):
             constraints.append(sum_elem_product(sensor_constraint, Z) + \
                                dp - dpp == rhs)
         for anchor_constraint, rhs, dp, dpp in zip(D[0], D[1], delta_p_sa, delta_pp_sa):
             constraints.append(sum_elem_product(anchor_constraint, Z) + \
                                dp - dpp == rhs)

In [37]: objective = cvx.Minimize(cvx.sum_entries(delta_p_ss) + \
                                  cvx.sum_entries(delta_pp_ss) + \
```

```
                                    cvx.sum_entries(delta_p_sa) + \
                                    cvx.sum_entries(delta_pp_sa))

        # First find some solution given the presence of the noise terms
        prob = cvx.Problem(objective, constraints)
        result = prob.solve(solver = 'MOSEK')
```

In [38]:
```
        print("Real sensors: ")
        print(sensors)
        print("Generated Sensors: ")
        generated = np.transpose(Z[0:dim, dim:dim+num_sensors].value)
        print(generated)
```

```
Real sensors:
[[-1.73149206  1.14908005]
 [-0.4842959  -0.11751045]
 [-1.68359731 -1.38633418]
 [-0.49126324 -0.59498631]
 [-1.80997962 -1.05069925]
 [-1.21422021 -0.45419742]
 [ 1.3351632   1.09978512]
 [ 1.21413695 -1.23406885]
 [ 0.63491031  0.14586037]
 [-0.0958941   0.58956201]]
Generated Sensors:
[[-1.73145986  1.14902039]
 [-0.4842856  -0.11751239]
 [-1.68355751 -1.3863104 ]
 [-0.4912512  -0.5949727 ]
 [-1.80993831 -1.05068806]
 [-1.21419356 -0.45419768]
 [ 1.33513145  1.09976654]
 [ 1.2141154  -1.23401217]
 [ 0.63489648  0.14586385]
 [-0.09589422  0.58954174]]
```

In [39]:
```
        # Now use generate sensor locations as the initial points for the
        # nonlinear least squares approach.
        check = 1000
        max_iter = 10000
        k = 0

        # Initial sensors guess
        sensors_0 = generated
        sensors_k = sensors_0

        # D iteration
```

```
        alpha = .000001
        while check > 10**-8 and k < max_iter:
            sensors_k1 = sensors_k - alpha * grad(sensors_k)
            check = np.linalg.norm(sensors_k1 - sensors_k)
            sensors_k = sensors_k1
            k = k+1

        print("Real sensors: ")
        print(sensors)
        print("Generated Sensors: ")
        print(sensors_k)
```

```
Real sensors:
[[-1.73149206  1.14908005]
 [-0.4842959  -0.11751045]
 [-1.68359731 -1.38633418]
 [-0.49126324 -0.59498631]
 [-1.80997962 -1.05069925]
 [-1.21422021 -0.45419742]
 [ 1.3351632   1.09978512]
 [ 1.21413695 -1.23406885]
 [ 0.63491031  0.14586037]
 [-0.0958941   0.58956201]]
Generated Sensors:
[[-1.73148875  1.1490604 ]
 [-0.48429807 -0.11750761]
 [-1.68358494 -1.3863605 ]
 [-0.49124967 -0.59501914]
 [-1.80995413 -1.05073905]
 [-1.21420435 -0.45423144]
 [ 1.33516547  1.09977821]
 [ 1.21415295 -1.23406684]
 [ 0.63492001  0.14583267]
 [-0.09588785  0.58954863]]
```

0.4.1 Question 4 : Steepest Descent and Projection Method

For a first order Steepest Descent method for the SNL problem, one of the main issues is ensuring that the semi-definite constraint is enforced at each iteration. In this project, we try 3 different approaches:

- Eigenvalue decomposition, ensuring that none of the eigenvalues are negative.
- LDL decomposition, ensuring that none of the pivots in D are negative.
- Eigenvalue decomposition, keeping only the k largest positive eigenvalues.

We find that the eigenvalue decomposition works consistently for ten sensors, four anchors, and two dimensions. For the LDL decomposition, we realized this approach is ill-suited to maintaining positive semi-definiteness. This is becase the LDL decomposition requires the matrix to be

positive definite in the first place in order to run. This leads the algorithm to fail once the point falls out of the feasible region. For the approach using only k of the largest positive eigenvalues, we predictably succeed if k = n, as this simply becomes the first approach outlined. However, as soon as k < n, the algorithm never converges on the correct sensor locations.

```
In [44]: num_sensors = 10
         num_anchors = 4
         dim = 2
         range_mult = 1

         anchors = generate_points(num_anchors, dim) * range_mult
         sensors = generate_points(num_sensors, dim)

         d_sa = list(map(lambda s: list(map(lambda a: np.linalg.norm(a - s), anchors)), sensors

         d_ss = list(map(lambda s1:
                 list(map(lambda s2: np.linalg.norm(s1 - s2), sensors))
             , sensors))

In [45]: A = enforce_id(dim, num_sensors)
         B = enforce_id2(dim, num_sensors)
         C = sensor_constraints(dim, num_sensors)
         D = anchor_constraints(num_anchors, num_sensors)

In [46]: def sum_elem_product_noncvx(A,B):
             return np.sum(np.multiply(A,B))

         def make_AX_less_b(X):
             output = []
             for id_constraint, rhs in zip(A[0], A[1]):
                 output.append(sum_elem_product_noncvx(id_constraint, X) - rhs)
             for id_constraint2, rhs in zip(B[0], B[1]):
                 output.append(sum_elem_product_noncvx(id_constraint2, X) - rhs)
             for sensor_constraint, rhs in zip(C[0], C[1]):
                 output.append(sum_elem_product_noncvx(sensor_constraint, X) - rhs)
             for anchor_constraint, rhs in zip(D[0], D[1]):
                 output.append(sum_elem_product_noncvx(anchor_constraint, X) - rhs)

             return output

         def scriptAt_y(y):
             output = np.zeros((dim+num_sensors,dim+num_sensors)) #start sum of y_i*A_i
             i = 0
             for id_constraint, rhs in zip(A[0], A[1]):
                 output += np.multiply(id_constraint, y[i])
                 i = i + 1
             for id_constraint2, rhs in zip(B[0], B[1]):
                 output += np.multiply(id_constraint2, y[i])
```

```
            i = i + 1
        for sensor_constraint, rhs in zip(C[0], C[1]):
            output += np.multiply(sensor_constraint, y[i])
            i = i + 1
        for anchor_constraint, rhs in zip(D[0], D[1]):
            output += np.multiply(anchor_constraint, y[i])
            i = i + 1
        return output

    def grad(X):
        return scriptAt_y(make_AX_less_b(X))
```

In [47]:
```
def eigen_decomp_project(X_k1):
    [eigenvalues, V] = np.linalg.eig(X_k1)
    eigenvalues[eigenvalues < 0] = 0
    Lambda = np.diag(eigenvalues)
    return np.dot(np.dot(V, Lambda), np.transpose(V))
```

In [48]:
```
def ldl_decomp_project(X_k1):
    def ldl_decomp(A):
        A = np.matrix(A)

        S = np.diag(np.diag(A))
        Sinv = np.diag(1/np.diag(A))
        D = np.matrix(S.dot(S))
        Lch = np.linalg.cholesky(A)
        L = np.matrix(Lch.dot(Sinv))
        return L, D
    [L, D] = ldl_decomp(X_k1)
    D[D < 0] = 0
    D_psd = np.diag(D)
    return np.dot(np.dot(L, D), np.transpose(L))
```

In [55]:
```
def pca_proj(X):
    [evalues, evectors] = sp.linalg.eigh(X)

    idx = evalues.argsort()[::-1]
    evalues = evalues[idx]
    evectors = evectors[:,idx]

    matrix = np.zeros(X.shape)
    width = evalues.shape[0]
    for i in range(9):
        index = width - i - 1
        if index >= 0 and evalues[index] > 0:
            matrix += evalues[index] * np.outer(evectors[:, index], evectors[:, index]
    return matrix
```

In [50]:
```
Beta = 100
def run_solver(method_name, projection_func):
```

```
    thing = random.rand(dim + num_sensors, dim + num_sensors)
    X_0 = np.dot(thing,thing.transpose())
    X_k = X_0
    k = 0
    check = 1000
    max_iter = 20000

    while check > 10**-8 and k < max_iter:
        # Get next iterate from descent
        X_k1 = X_k - (1/Beta) * grad(X_k)

        # Project back into cone with eigen decomp
        X_k1 = projection_func(X_k1)

        #set up for next iteration
        check = np.linalg.norm(X_k1 - X_k)
        X_k = X_k1

        k = k+1

    print("Real sensors: ")
    print(sensors)
    print("Generated Sensors for method " + method_name)
    generated = np.transpose(X_k[0:dim, dim:dim+num_sensors])
    print(generated)
```

```
In [51]: run_solver("Eigenvalue Decomposition", eigen_decomp_project)

Real sensors:
[[ 0.11911412  1.38362663]
 [-0.69435684 -1.40503445]
 [ 1.13409721  1.52692982]
 [ 1.2775626  -0.002812  ]
 [-0.27034573 -0.688677  ]
 [ 0.12282072  1.8767791 ]
 [ 1.72091239  1.49248362]
 [-0.64155136  1.25286935]
 [-0.48007333 -1.96914533]
 [-1.54460461  0.76727871]]
Generated Sensors for method Eigenvalue Decomposition
[[ 0.11911417  1.3836265 ]
 [-0.69435669 -1.40503436]
 [ 1.13409716  1.52692969]
 [ 1.27756256 -0.00281201]
 [-0.27034562 -0.68867696]
 [ 0.12282075  1.87677892]
 [ 1.7209123   1.49248349]
 [-0.64155124  1.25286923]
```

```
 [-0.48007319 -1.96914519]
 [-1.54460441  0.76727863]]
```

```
In [ ]: # Diverges with error because the matrix eventually becomes non positive-definite
        # run_solver("LDL Decomposition", ldl_decomp_project)
```

```
In [58]: run_solver("Top 6 Eigenavalues", pca_proj)
```

```
Real sensors:
[[ 0.11911412  1.38362663]
 [-0.69435684 -1.40503445]
 [ 1.13409721  1.52692982]
 [ 1.2775626  -0.002812  ]
 [-0.27034573 -0.688677  ]
 [ 0.12282072  1.8767791 ]
 [ 1.72091239  1.49248362]
 [-0.64155136  1.25286935]
 [-0.48007333 -1.96914533]
 [-1.54460461  0.76727871]]
Generated Sensors for method Top 6 Eigenavalues
[[ 0.03689537  0.0165328 ]
 [ 0.1522006   0.37062384]
 [-0.02979418  0.01039562]
 [-0.15137448 -0.04655928]
 [ 0.04253337  0.11605903]
 [ 0.01166932 -0.07747883]
 [-0.19841848 -0.09613416]
 [-0.06465546 -0.16185471]
 [-0.24062498 -0.01725277]
 [-0.16560825 -0.312057  ]]
```