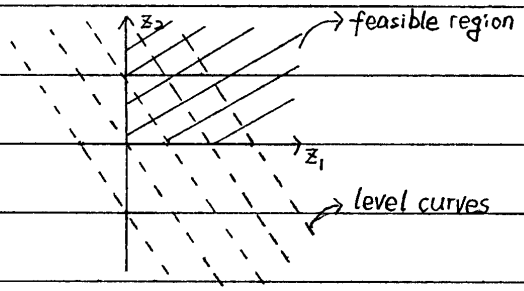
ME231 HW2

1. (a) $\min_{z_1, z_2} \; 3z_1 + 2z_2$

s.t. $z_1 \geq 0$

$z_2 \geq 0$



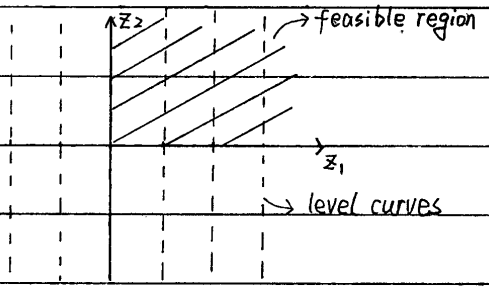
traits: feasible, bounded

has a unique optimum

(b) $\min_{z_1, z_2} \; z_1$

s.t. $z_1 \geq 0$

$z_2 \geq 0$



traits: feasible, bounded

has multiple optima

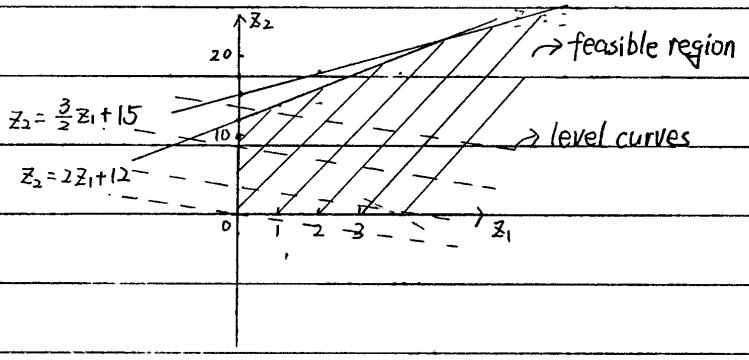
(c) $\min_{z_1, z_2} \; -5z_1 - 7z_2$

s.t. $-3z_1 + 2z_2 \leq 30$

$-2z_1 + z_2 \leq 12$

$z_1 \geq 0$

$z_2 \geq 0$



[redacted], traits : unbounded

(d) $\min_{z_1, z_2} \; z_1 - z_2$

s.t. $z_1 - z_2 \geq 2$

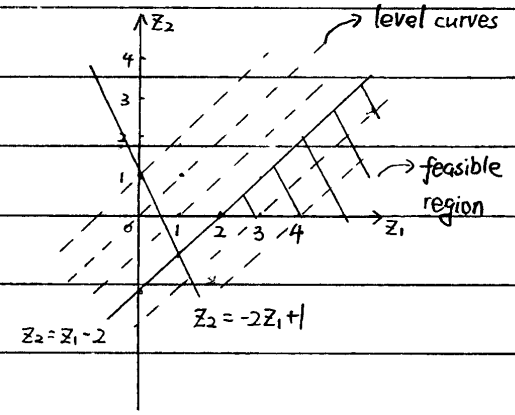
$2z_1 + z_2 \geq 1$

$z_1 \geq 0$

$z_2 \geq 0$

traits: feasible, bounded

has multiple optima



(e) $\min_{z_1, z_2} \; 3z_1 + z_2$

s.t. $z_1 - z_2 \le 1$

$3z_1 + 2z_2 \le 12$

$2z_1 + 3z_2 \le 3$

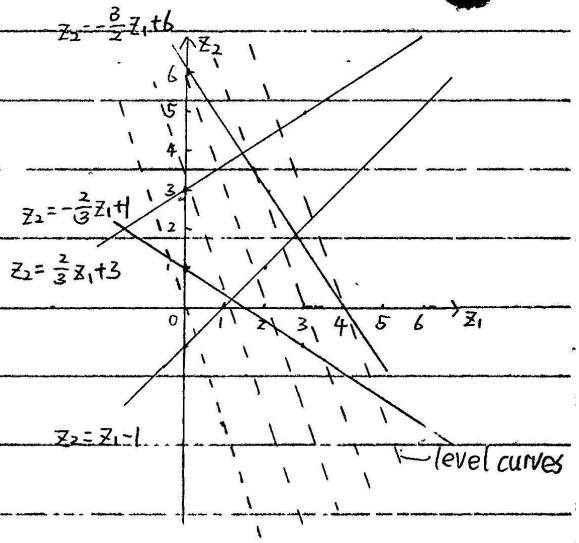
$-2z_1 + 3z_2 \ge 9$

$z_1 \ge 0$

$z_2 \ge 0$

traits: infeasible, unbounded

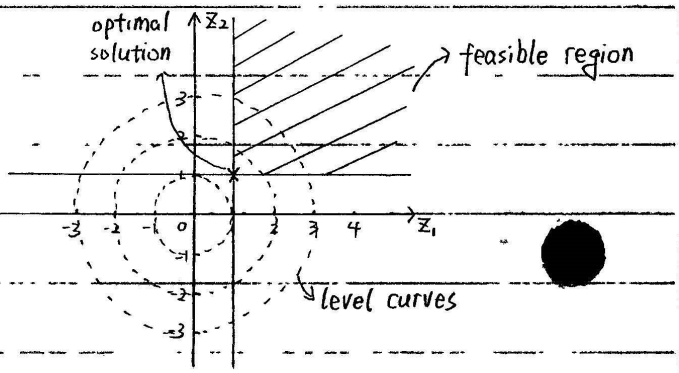
has no optimum



2. (a) $\min_{z_1, z_2} \; z_1^2 + z_2^2$

s.t. $z_1 \ge 1$ (active)

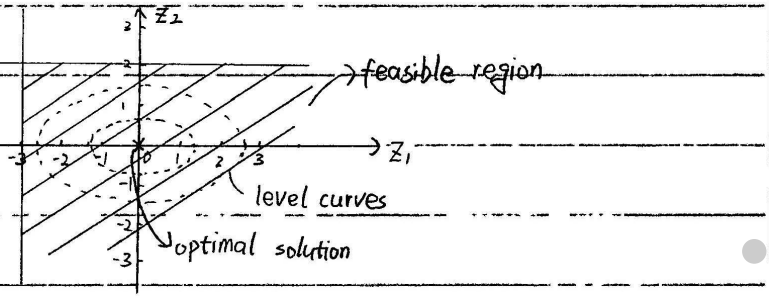
$z_2 \ge 1$ (active)



(b) $\min_{z_1, z_2} \; 2z_1^2 + 7z_2^2$

s.t. $z_1 \ge -3$ inactive)

$z_2 \ge z_2$ inactive)

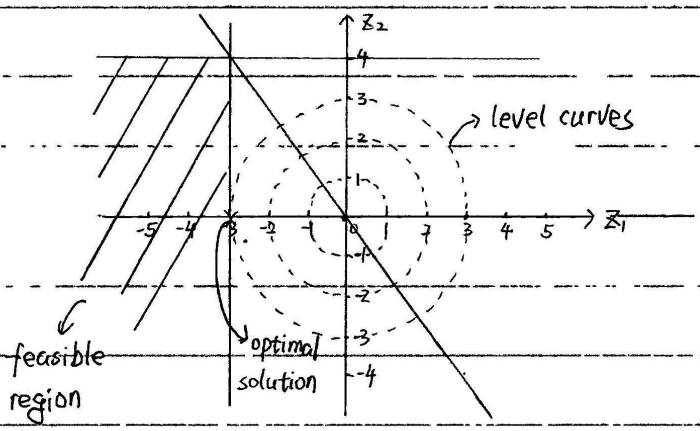


(c) $\min_{z_1, z_2} \; z_1^2 + z_2^2$

s.t. $z_1 \le -3$ (active)

$z_2 \le 4$ inactive)

$0 \ge 4z_1 + 3z_2$ (inactive)



Assignment 2: Optimization 1 (Solution)

University of California Berkeley

ME C231A, EE C220B, Experiential Advanced Control I

Question 1. Linear programming, by hand

Question 2. Quadratic programming, by hand

The written solutions for questions 1, 2 can be found in the attached pdf file.

Question 3. Approximately verifying optimality, by hand

Part (a) Evaluate the cost at for example (z1=0.6, z2=0.4, z3=0), also at (z1=0.46, z2=0.55, z3=0) and some other points to see the given point is a local minimum.

Part (b) This optimization problem has a quadratic cost and linear constraints, so it is a convex quadratic program (QP). For a convex optimization problem, every locally optimal solution is globally optimal. Thus, z^* is also a global minimum.

---

4. Using `linprog` to solve LPs:

In [1]:
```
import numpy as np
import scipy as cp
from scipy.optimize import linprog
```

Read about the features and syntaxes of **`scipy.optimize.linprog`** package here: https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.linprog.html (https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.linprog.html). Then use it to solve the following problem. Let $x, y, z \in \mathbb{R}$.

$$\begin{aligned} \min_{x,y,z} \quad & x + y + z \\ \text{subject to } & 2 \leq x \\ & -1 \leq y \\ & -3 \leq z \\ & x - y + z \geq 4 \end{aligned}$$

To get you started, here is one form of a solution.

In [2]:
```
f = np.ones((3,))   # 1D array
A = np.array([[-1, 0, 0], [0, -1, 0], [0, 0, -1], [-1, 1, -1]])   # 2D array
b = np.array([-2,1,3,-4])   # 1D array
# default bound is (0, None), so here we need to specify bound as (None, Non
e),
# since all the bounds all already considered in the inequality constraints
opt1 = cp.optimize.linprog(c=f, A_ub=A, b_ub=b, bounds=(None, None), method='s
implex')
print('x*=', opt1.x)
print('J*=', opt1.fun)
```

```
x*= [ 6. -1. -3.]
J*= 2.0
```

You can also use the elementwise bounds to solve the same problem in a slightly different way.

In [3]:
```
f = np.ones((3,))
A = np.array([[-1, 1, -1]])
b = np.array([-4])
x_bounds = (2, None)
y_bounds = (-1, None)
z_bounds = (-3, None)
# opt2 = cp.optimize.linprog(c=f, A_ub=A, b_ub=b, bounds=[x_bounds, y_bounds,
 z_bounds], method='simplex')
opt2 = cp.optimize.linprog(c=f, A_ub=A, b_ub=b, bounds=[x_bounds, y_bounds, z_
bounds])
print('x*=', opt2.x)
print('J*=', opt2.fun)
```

```
x*= [ 4.00000001 -1.         -0.99999999]
J*= 2.0000000134883065
```

You might get different values of x, y, z but the same cost for each solution. Why?

The solver provides one optimal solution among many optimal solutions. Hence, depnding on the choice of solver the optimizer can be different, but the optimal value is the same.

---

5. Using `cvxopt` to solve QPs:

Read about the features and syntaxes of **`cvxopt`** online. Constrained least-squares problems are useful when your decision parameters are known to reside in a constrained set. Here, we show how to solve a constrained least-squares as a quadratic program. The constrained least-squares problem is of the form

$$\min_x \|Ax - b\|_2^2$$
$$\text{subject to } l_i \leq x_i \leq u_i$$

Observe that

$$\|Ax - b\|_2^2 = (x^T A^T - b^T)(Ax - b) = x^T A^T Ax - 2b^T Ax + b^T b$$

We can drop the constant term $b^T b$ in the optimization program and add it back later. The constrained least-squares problem becomes

$$\min_x \ \frac{1}{2} x^T \left(2A^T A\right) x + (-2A^T b)^T x$$
$$\text{subject to } \ l_i \leq x_i \leq u_i,$$

which is a quadratic program. For the following A and b for $X \in \mathbb{R}^5$:

In [4]:
```
n = 4 # dimenstion of x
# A = np.random.randn(n, n)
# b = np.random.randn(n)
A = np.array([[-0.54, -1.81,  0.25, -0.46],   # A and b matrices are generated
using np.random.randn and kept fixed for the homework.
 [-0.38,  0.37, -2.48, -0.68],
 [-1.31,  0.74, -1.57,  0.28],
 [-0.31, -0.02,  0.75,  0.20]])
b = np.array([ 0.40, -2.45, -0.23, 0.98])
l_i = -0.5
u_i = 0.5
```

use the **`cvxopt`** to solve the constrained least-squares problem.

In [5]:
```
import cvxopt
from cvxopt import matrix, solvers

P = 2*A.T@A
q = -2*A.T@b
G = np.concatenate([np.eye(n), -np.eye(n)], axis=0)
h = np.concatenate([u_i*np.ones((n,)), -l_i*np.ones((n,))], axis=0)

P = cvxopt.matrix(P, tc='d')
q = cvxopt.matrix(q, tc='d')
G = cvxopt.matrix(G, tc='d')
h = cvxopt.matrix(h, tc='d')
sol = cvxopt.solvers.qp(P,q,G,h)

print('x*=', sol['x'])
print('p*=', sol['primal objective'] + b.T@b)
```

```
     pcost       dcost       gap    pres   dres
 0: -6.7988e+00 -1.2012e+01  2e+01  2e+00  1e-16
 1: -6.3888e+00 -9.0588e+00  4e+00  3e-01  4e-17
 2: -6.0963e+00 -6.3893e+00  3e-01  3e-03  6e-16
 3: -6.1599e+00 -6.1690e+00  9e-03  4e-05  1e-16
 4: -6.1606e+00 -6.1607e+00  1e-04  4e-07  4e-17
 5: -6.1606e+00 -6.1606e+00  1e-06  4e-09  1e-16
Optimal solution found.
x*= [-3.00e-01]
[-2.47e-01]
[ 5.00e-01]
[ 5.00e-01]

p*= 1.0152149348586423
```

Compare the result to analytical solution with projection: standard least-squares, followed by variable clipping to enforce constraints after-the-fact.

In [6]:
```
xstar = (np.linalg.inv((A.T @ A))) @ (A.T.dot(b))
xstar[xstar > u_i] = u_i  # projection
xstar[xstar < l_i] = l_i  # projection
print('x_analytical:', xstar)
print('x_cvxopt:', sol['x'])
print(np.linalg.norm(A @ (np.reshape(sol['x'],((4),))) - b))
print(np.linalg.norm(A @ xstar - b))
```

```
x_analytical: [-0.5        -0.20267715  0.5         0.5       ]
x_cvxopt: [-3.00e-01]
[-2.47e-01]
[ 5.00e-01]
[ 5.00e-01]

1.0075787487132912
1.0561293728400345
```

Finding the analytical solution and then projecting the solution on the feasible set results in approximated solution. The exact solution must be computed by calling a solver such as `cvxopt`.

---

6. System Identification of a Building Zone Temperature Model:

In Homework 1, you computed a discrete time model of a building zone temperature that looks like the following equation:

$$T(k+1) = (1 - p_1 u_1(k))T(k) + p_2 u_1(k) u_2(k) + q(k)$$

where p_1 and p_2 are now parameters of the model that we want to identify using a least-squares approach. In the accompanying `etch2169.mat` file, you have data for the following measurements of the state and inputs on a certain day:

1. zone temperature T in array `RoomTempData`
2. air mass flow rate u_1 in array `FanData`
3. supply air temperature u_2 in array `SupplyTempData`

For each of the arrays, the first column represents the actual date and time that the measurement (in the second column) is taken. Note that we have access to fan speed data as opposed to air mass flow rate data. Assuming that they are equal (up to a constant) is an approximation which does not hold in general, but is okay for our purposes. To determine the parameters p_1 and p_2, we use a weekend day and assume that the heat load $q(k)$ is zero (when there are typically few graduate students in the lab to add heat to the system). Specifically, follow these steps:

The dates and times that are stored in the first column of each array mentioned above are in the format of $serial\ date\ numbers$. Serial date numbers represent a calendar date as the number of days that have passed since a fixed base date of January 1, 0000 A.D.. Note that the measurements were not taken at the exact same times; they are each off by a few seconds from another. Now we use the function `datenum` to convert the date character back into the serial date number format. Note that 1 minute corresponds to the real number 0.00069444440305233. Furthermore, we use `datenum` to determine a sampling time `TS` of 1 minute in serial date number format. Finally, we create a vector of serial date numbers from 10:01:00 AM to 3:59:00 PM on Sep. 9, 2018. The downloaded data contains the day's measurements from between 10 AM and 4 PM.

In [7]:
```
from scipy.io import loadmat

Data = loadmat('etch2169.mat')
RoomTempData = Data['RoomTempData']
FanData = Data['FanData']
SupplyTempData = Data['SupplyTempData']

#print the first data point  just to understand the data type
time_room_temp = RoomTempData[0,0]
time_fan  = FanData[0,0]
time_sup_temp = SupplyTempData[0,0]

print(time_room_temp)
print(time_room_temp)
print(time_fan)
print(RoomTempData[0,1])
print(FanData[0,1])
print(SupplyTempData[0,1])
```

```
737312.4170802135
737312.4170802135
737312.4171149357
74.9000015258789
54.400001525878906
69.0999984741211
```

In [8]:
```
from datetime import date
from datetime import datetime as dt

def datenum(d):
    return 366 + d.toordinal() + (d - dt.fromordinal(d.toordinal())).total_sec
onds()/(24*60*60)

d_start = dt.strptime('2018-9-9 10:1','%Y-%m-%d %H:%M')
d_end = dt.strptime('2018-9-9 15:59','%Y-%m-%d %H:%M')
d_start_plus_onemin = dt.strptime('2018-9-9 10:2','%Y-%m-%d %H:%M')

TS = datenum(d_start_plus_onemin) - datenum(d_start)
print(TS)
TimeQuery = np.arange(start=datenum(d_start), stop=datenum(d_end), step=TS)
```

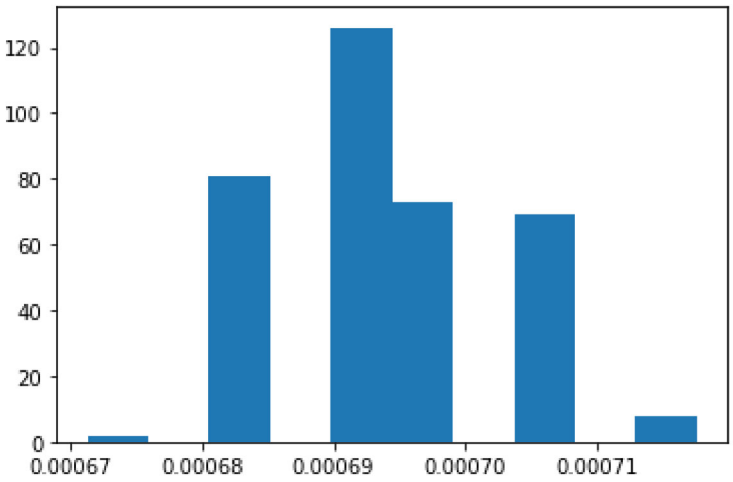
```
0.00069444440305233
```

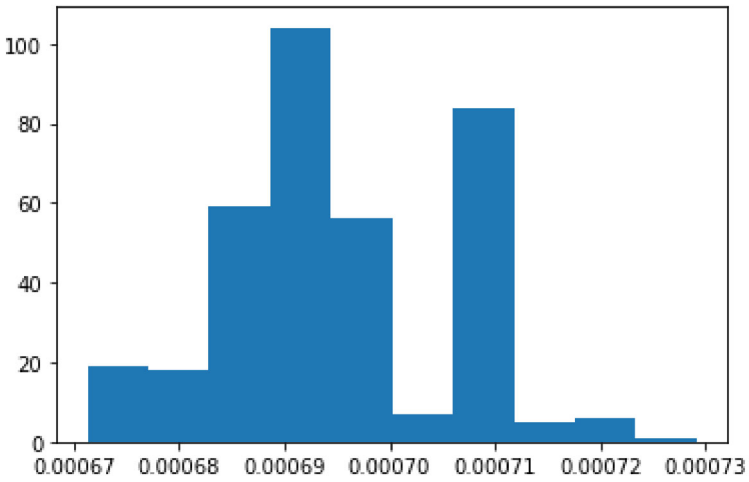
Part (a)

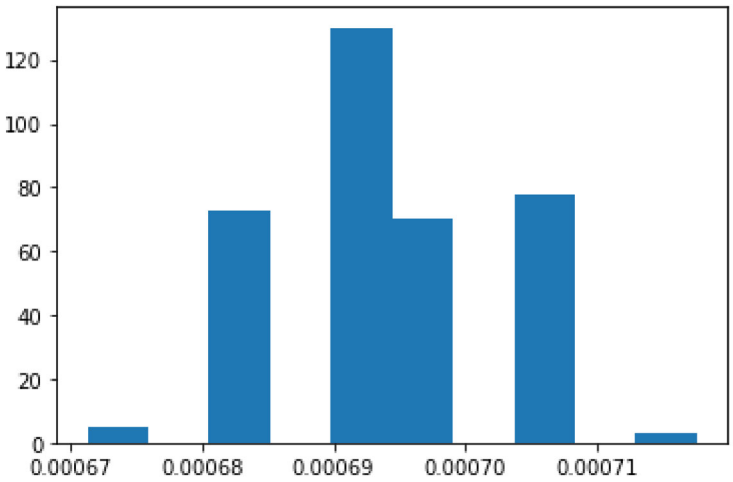
Verify using *np.diff* and *np.histogram* that the data is nearly collected with uniform sample time, but that there is a few percent variability.

In [9]:

```
import matplotlib.pyplot as plt

counts, bins = np.histogram(np.diff(RoomTempData[:,0]))
plt.hist(bins[:-1], bins, weights=counts)
plt.show()

counts, bins = np.histogram(np.diff(FanData[:,0]))
plt.hist(bins[:-1], bins, weights=counts)
plt.show()

counts, bins = np.histogram(np.diff(SupplyTempData[:,0]))
plt.hist(bins[:-1], bins, weights=counts)
plt.show()
# RoomTempData.shape
```





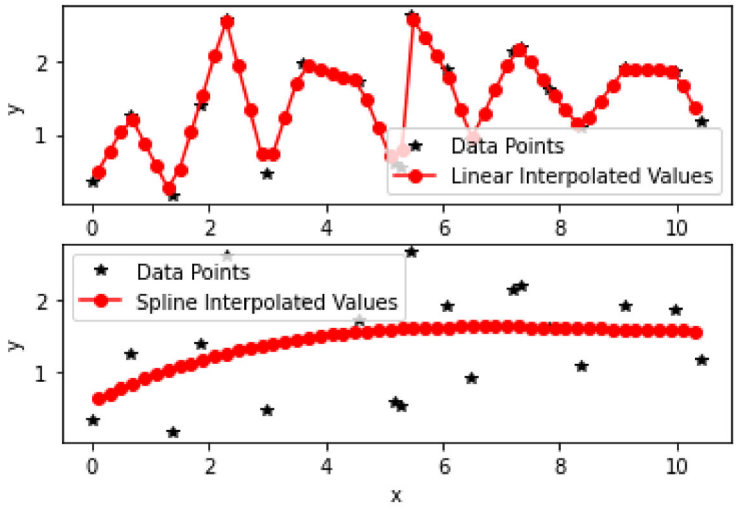


Part (b)

Next we are going to interpolate the data points to obtain two effects 1- remove outlies/filter the data, 2- call the interpolated fucntion so that the new data is sampled exactly every minute (start at 10:01 AM, then 10:02 AM, etc. till 3:59 PM). (note that python has many ways of resampling data-- this is just one approach) **Hint:** Use the `scipy` package `interpolate` and the commands `interpolate.interp1d` and `interpolate.UnivariateSpline`. As an illustration, understand the example below:

In [10]:
```
from scipy import interpolate
import matplotlib.pyplot as plt

xData = np.cumsum(np.hstack([np.zeros((1,)), 0.1+np.random.rand(19,)]))
yTmp = np.hstack([np.sort(3*np.random.rand(10,1)), np.fliplr(np.sort(3*np.rand
om.rand(10,1)))])
yData = yTmp.flatten()
xQuery = np.arange(start=0.1, stop=np.max(xData), step=0.2)

f_interp = interpolate.interp1d(xData, yData, 'linear')
yInterpLinear = f_interp(xQuery)

f_spline = interpolate.UnivariateSpline(xData, yData)
yInterpSpline = f_spline(xQuery)

plt.subplot(2,1,1)
plt.plot(xData, yData, 'k*', xQuery, yInterpLinear, '-or')
plt.legend(['Data Points', 'Linear Interpolated Values'])
plt.ylabel('y')
plt.subplot(2,1,2)
plt.plot(xData, yData, 'k*', xQuery, yInterpSpline, '-or')
plt.legend(['Data Points', 'Spline Interpolated Values'])
plt.xlabel('x')
plt.ylabel('y')
```

Out[10]:
```
Text(0, 0.5, 'y')
```

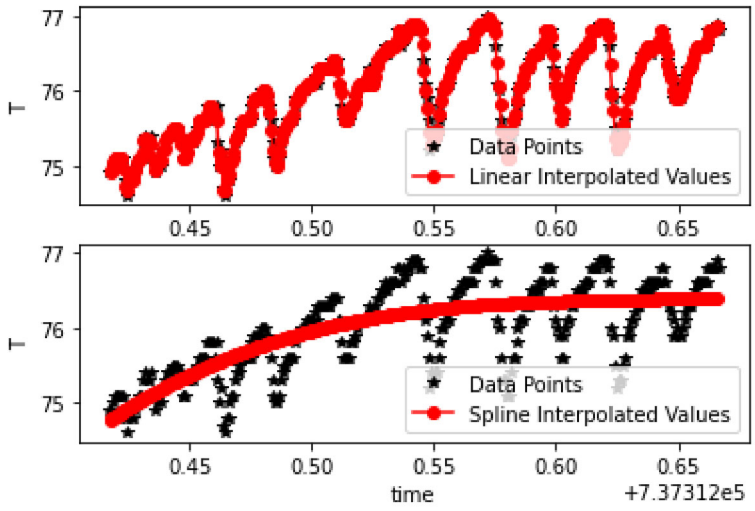


Plot and compare the linear- and spline-interpolated values to the actual data. Choose a method and justify your choice. Use a different interpolation method if you see fit.

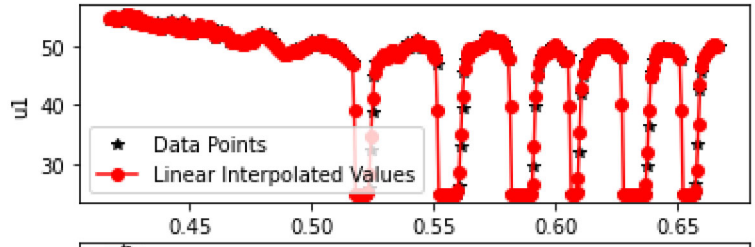
In [11]:

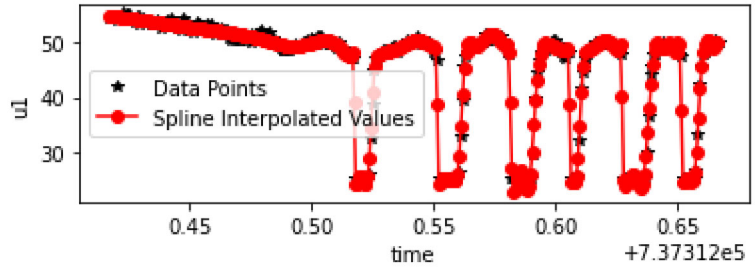
```
# xData
time_T = RoomTempData[:,0]
time_u1 = FanData[:,0]
time_u2 = SupplyTempData[:,0]

# yData
data_T = RoomTempData[:,1]
data_u1 = FanData[:,1]
data_u2 = SupplyTempData[:,1]

# linear
InterpLinear_T_mid = interpolate.interp1d(time_T, data_T, 'linear')
InterpLinear_T = InterpLinear_T_mid(TimeQuery)

InterpLinear_u1_mid = interpolate.interp1d(time_u1, data_u1, 'linear')
InterpLinear_u1 = InterpLinear_u1_mid(TimeQuery)

InterpLinear_u2_mid = interpolate.interp1d(time_u2, data_u2, 'linear')
InterpLinear_u2 = InterpLinear_u2_mid(TimeQuery)


# Spline
InterpSpline_T_mid = interpolate.UnivariateSpline(time_T, data_T)
InterpSpline_T = InterpSpline_T_mid(TimeQuery)

InterpSpline_u1_mid = interpolate.UnivariateSpline(time_u1, data_u1)
InterpSpline_u1 = InterpSpline_u1_mid(TimeQuery)

InterpSpline_u2_mid = interpolate.UnivariateSpline(time_u2, data_u2)
InterpSpline_u2 = InterpSpline_u2_mid(TimeQuery)

# T Plot
plt.subplot(2,1,1)
plt.plot(time_T, data_T, 'k*')
plt.plot(TimeQuery, InterpLinear_T, '-or')
plt.legend(['Data Points', 'Linear Interpolated Values'])
plt.xlabel('time')
plt.ylabel('T')
plt.subplot(2,1,2)
plt.plot(time_T, data_T, 'k*')
plt.plot(TimeQuery, InterpSpline_T, '-or')
plt.legend(['Data Points','Spline Interpolated Values'])
plt.xlabel('time')
plt.ylabel('T')
plt.show()
# u1 Plot
plt.subplot(2,1,1)
plt.plot(time_u1, data_u1, 'k*')
plt.plot(TimeQuery, InterpLinear_u1, '-or')
plt.legend(['Data Points', 'Linear Interpolated Values'])
plt.xlabel('time')
plt.ylabel('u1')
plt.subplot(2,1,2)
plt.plot(time_u1, data_u1, 'k*')
plt.plot(TimeQuery, InterpSpline_u1, '-or')
plt.legend(['Data Points','Spline Interpolated Values'])
```

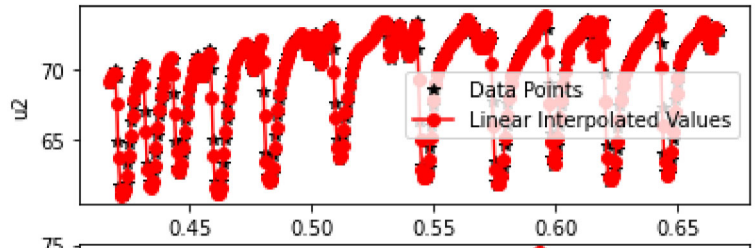
```
plt.xlabel('time')
plt.ylabel('u1')
plt.show()
#u2 Plot
plt.subplot(2,1,1)
plt.plot(time_u2, data_u2, 'k*')
plt.plot(TimeQuery, InterpLinear_u2, '-or')
plt.legend(['Data Points', 'Linear Interpolated Values'])
plt.xlabel('time')
plt.ylabel('u2')
plt.subplot(2,1,2)
plt.plot(time_u2, data_u2, 'k*')
plt.plot(TimeQuery, InterpSpline_u2, '-or')
plt.legend(['Data Points','Spline Interpolated Values'])
plt.xlabel('time')
plt.ylabel('u2')
plt.show()
```

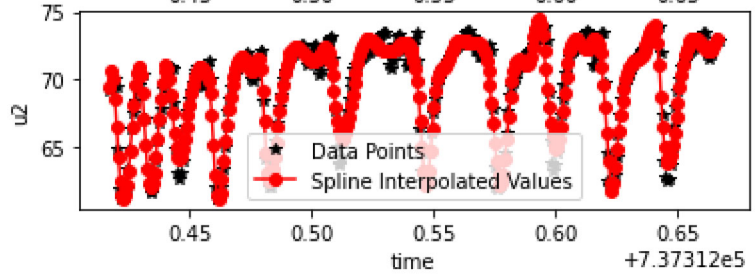

T
77
76
75
Data Points
Linear Interpolated Values
0.45
0.50
0.55
0.60
0.65

T
77
76
75
Data Points
Spline Interpolated Values
0.45
0.50
0.55
0.60
0.65
time
+7.37312e5


u1
50
40
30
Data Points
Linear Interpolated Values
0.45
0.50
0.55
0.60
0.65


u1
50
40
30
Data Points
Spline Interpolated Values
0.45
0.50
0.55
0.60
0.65
time
+7.37312e5


u2
70
65
Data Points
Linear Interpolated Values
0.45
0.50
0.55
0.60
0.65


u2
75
70
65
Data Points
Spline Interpolated Values
0.45
0.50
0.55
0.60
0.65
time
+7.37312e5

To do least squares parameter estimation, we want to solve the following problem:

$$\min_{p_1,p_2,e} \sum_{k=1}^{N} \|e(k)\|_2^2$$
$$\text{s.t. } T_{data}(k+1) = (1 - p_1 u_1(k))T_{data}(k) + p_2 u_1(k)u_2(k) + q(k) + e(k)$$
$$\forall k = \{0, \ldots, N-1\}$$

where N is the number of data samples used, T_{data}, u_1, and u_2 represent the resampled/interpolated data, and $e(k)$ is the one step model estimation error of the temperature, whose norm (over time) is minimized by proper choice of parameters. Populate the appropriate matrices A and b using the known data such that

$$\begin{bmatrix} e(1) \\ \vdots \\ e(N) \end{bmatrix} = Ap - b$$

where $A \in \mathbb{R}^{N\times 2}$, $b \in \mathbb{R}^N$ and p is the 2×1 unknown parameter vector. Perform the least squares estimation of the parameters using **`cvxopt.solvers.qp`**. Implement this code in a function

In [12]:
```
def bldgIdentification(Tdata, u1Seq, u2Seq):
    return estParm
```

In [13]:
```
from cvxopt import matrix, solvers
N = 360

def bldgIdentification(Tdata, u1Seq, u2Seq):
    b = -np.diff(Tdata)
    a2 = -np.multiply(u1Seq, u2Seq)
    a1 = np.multiply(u1Seq, Tdata[0:-1])
    A = np.concatenate([np.reshape(a1,(len(a1),1)), np.reshape(a2,(len(a2),1
))], axis = 1)
    P_mid = 2*A.T @ A
    q_mid = -2*A.T @ b
    P = cvxopt.matrix(P_mid, tc='d')
    q = cvxopt.matrix(q_mid, tc='d')
    sol = cvxopt.solvers.qp(P, q)
    estParm = sol['x']
    return estParm

Tdata = RoomTempData[:,1]
u1Seq = FanData[0:-1,1]
u2Seq = SupplyTempData[0:-1,1]

Param = bldgIdentification(Tdata, u1Seq, u2Seq)
print(Param)
```

```
[ 4.02e-04]
[ 4.41e-04]
```

The input arguments **`u1Seq`** and **`u2Seq`** are of length N, while **`Tdata`** has length $N+1$. The code should formulate and solve the least squares problem, and return the parameter estimate as a 2×1 vector.

See how good are thre model preedictions over a long horizon. On a single figure, plot the day's actual Temperature data versus time. Then, compute and plot T_{est} by propagating the the model forward as follows:

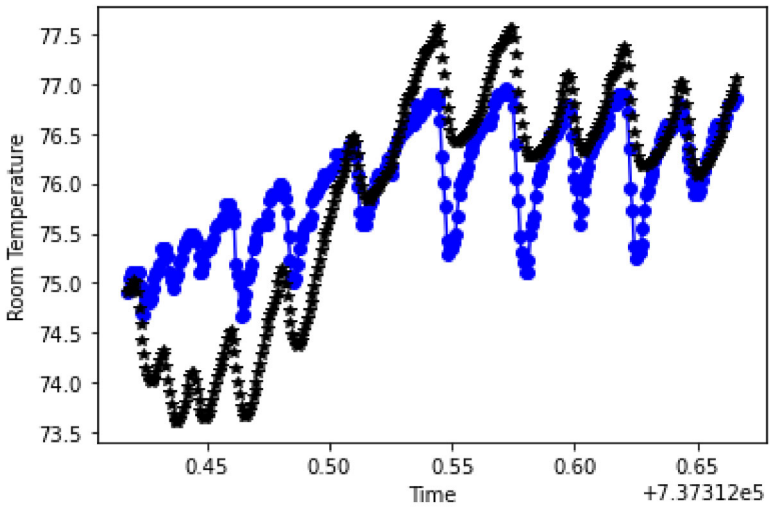
$$T_{est}(0) = T_{actual}(0)$$
$$T_{est}(k+1) = (1 - p_1 u_1(k))T_{est}(k) + p_2 u_1(k)u_2(k) + q(k)$$

using the values for p_1 and p_2 as obtained by the least-squares solution.

In [14]:
```
Tdata = InterpLinear_T
u1Seq = InterpLinear_u1[0:-1]
u2Seq = InterpLinear_u2[0:-1]

Param = bldgIdentification(Tdata, u1Seq, u2Seq)
print(Param)

N = len(Tdata)
T_est = np.zeros((N,1))
T_intit = Tdata[0]
T_est[0] = T_intit
for i in range(N-1):
    T_est[i+1] = (1 - Param[0]*u1Seq[i])*T_est[i] + Param[1]*u1Seq[i]*u2Seq[i]


plt.plot(TimeQuery, Tdata, '-bo')
plt.plot(TimeQuery, T_est, 'k*')
plt.xlabel('Time')
plt.ylabel('Room Temperature')
# plt.legend('Actual Data','Predictive Model')
```

```
[ 3.99e-04]
[ 4.38e-04]
```

Out[14]:
```
Text(0, 0.5, 'Room Temperature')
```



7. Linear Classification:

In this problem, a population P (often people, but not necessary) refers to a collection of distinct objects, called its *members*. Associated with each member m_k is a vector v_k of values of known, delineated traits of that member, called the member's $features$. The dimension of each feature vector is $n_F \times 1$. Hence, using this feature vector, each member of the population can be represented as a point in n_F dimensional space. Note though, it is possible for two different members to have identical features.

Suppose the the population P is divided into two distinct groups, G_1 and G_2. The goal of *linear classification*, is to find a linear function, $L(v) := c^T v + b$, such that

$$c^T v_k + b > 0 \ \text{ for all } v_k \in G_1$$

and

$$c^T v_k + b < 0 \ \text{ for all } v_k \in G_2$$

In other words, the linear inequalities $c^T v + b > 0$ and $c^T v + b < 0$ split the n_F-dimensional space into two nonintersecting halfspaces, and one group of the population lies entirely in one halfspace, while the other group lies in the other halfspace. The linear function $L(v) := c^T v + b$ is said to *classify* the population into the two groups. **Such a function is often used as a predictor for another person, attempting to predict which group it belongs to, based on the value of $L(v)$, where v is the known vector of this person's traits.**

A concrete example will make the ideas clear. Suppose $n_F = 2$, and the traits (features) are the 2-dimensional vector

$$v = \begin{bmatrix} \text{age} \\ \text{number of action movies seen this year} \end{bmatrix}$$

The population consists of n_P people who have just viewed a pre-release showing of a new movie, called MovieX. The two groups consists of the people who liked (based on a short exit survey) MovieX (G_1), and those people who did not like MovieX (G_2). Using this data, the producers of MovieX want to build a linear classifier (based on *age* and *number of action movies seen this year*) which will predict if another person (outside this population set) will like the movie, so they can target their promotions accordingly (eg., if they discover that older viewers who don't see many action movies prefer MovieX, then they might do a pamphlet-handout to middle-aged people buying non-action movie tickets a week before MovieX is released).

The generalization is as follows: Given two sets of points

$$G_1 = (v_1, v_2, \ldots, v_N), \quad G_2 = (v_{N+1}, v_{N+2}, \ldots v_P)$$

where each $v_k \in \mathbf{R}^{n_F}$. The task is to find a linear constraint that "separates" them.
Specifically, find $c \in \mathbf{R}^{n_F}$ and $b \in \mathbf{R}$ such that

$$c^T v_k + b > 0 \text{ for all } k \leq N$$

and

$$c^T v_k + b < 0 \text{ for all } k > N.$$

Note that if this is possible, then simply by scaling c and b, it follows that

$$c^T v_k + b \geq 1 \text{ for all } k \leq N$$

and

$$c^T v_k + b \leq -1 \text{ for all } k > N.$$

A serious problem is that there may be some outliers in the data so that the two sets G_1 and G_2 simply can't be separated by a hyperplane (ie., there is some discrete overlap in the points). For those points, you need to add/subtract a nonnegative slack variable, $t_k \geq 0$ to make it work, so perhaps we adjust the requirement to

$$c^T v_k + b \geq 1 - t_k \text{ for all } k \leq N$$

and

$$c^T v_k + b \leq -(1 - t_k) \text{ for all } k > N.$$

But, hopefully **most points don't need this adjustment**, so you force the use of such t_k to a minimum by minimizing $\sum_{k=1}^{P} t_k$. This results in a linear program of the form

$$\min_{c,b,t_1,\ldots,t_P} \quad t_1 + t_2 + \cdots + t_P$$

subject to

$$t_1 \geq 0, t_2 \geq 0, \ldots, t_P \geq 0$$

and

$$c^T v_k + b \geq 1 - t_k \text{ for all } k \leq N$$

and

$$c^T v_k + b \leq -(1 - t_k) \text{ for all } k > N.$$

In [15]:
```
def buildLinClass(G1, G2):
    return c,b,t
```

The input arguments G_1 and G_2 are real-values arrays of dimension $n_F \times p_1$ and $n_F \times p_2$, respectively. Each column is a feature vector of a particular person in that group.

The output arguments are a $n_F \times 1$ vector c and a scalar value b, such that the linear function $L(v) := c^T v + b$ approximately classifies the two groups, with $L(v) > 0$ for members of G_1 and $L(v) < 0$ for members in G_2.
Of course, because of outliers, the classification is not necessarily exact, but the linear program minimization has minimized the total amount of slack variables used to create the separation. The 3rd output argument is the vector of required corrections t, and is of dimension $(p_1 + p_2) \times 1$.

You can try your function out on synthetic data (which has no outliers) below. Here there are only 2 features, so we can visually see the classification (could also do in 3-dimenions). Note that in practice, the number of features is usually much larger than 2, and hence cannot be visualized.

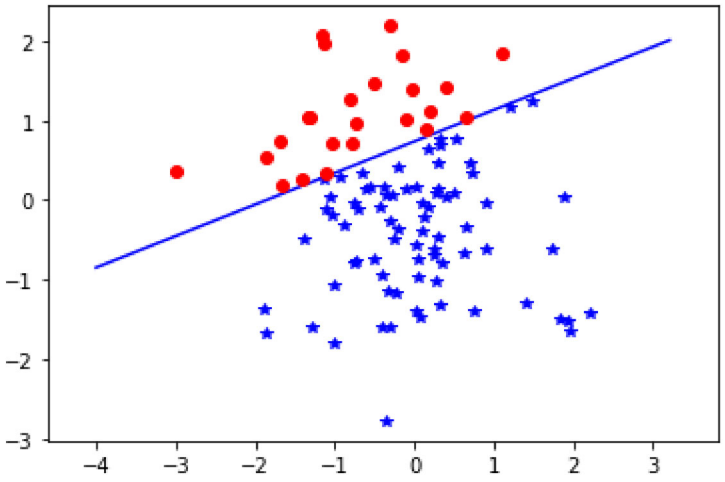
In [16]:

```
import numpy as np
import scipy as cp
# import cvxpy as cv
from scipy.optimize import linprog
import matplotlib.pyplot as plt
from cvxopt import matrix, solvers
def buildLinClass(G1, G2):
    nf, p1 = np.shape(G1)
    n, p2 = np.shape(G2)
    f = np.concatenate([np.zeros((nf+1,)), np.ones((p1+p2,))], axis = 0)

    A = np.concatenate([np.concatenate([np.zeros((p1+p2,nf+1)), -np.eye(p1+p2
)], axis = 1),
                        np.concatenate([-G1.T, -np.ones((p1,1)), -np.eye(p1),
np.zeros((p1,p2))], axis = 1),
                        np.concatenate([G2.T, np.ones((p2,1)), np.zeros((p2,p1
)), -np.eye(p2)], axis = 1)], axis = 0)

    b = np.concatenate([np.zeros((p1+p2,)), -np.ones((p1,)), -np.ones((p2,))],
axis = 0)

    f = cvxopt.matrix(f, tc='d')
    A = cvxopt.matrix(A, tc='d')
    b = cvxopt.matrix(b, tc='d')

    sol = cvxopt.solvers.lp(f,A,b)
    xOpt = sol['x']
    J = sol['primal objective']

    c = np.array(xOpt[0:nf]).flatten()
    b = np.array(xOpt[nf]).flatten()
    t = np.array(xOpt[nf+1:]).flatten()

    return c, b, t


nF = 2
nP = 100
cTrue = np.random.randn(nF,1)
bTrue = np.random.randn(1,1)
Pop = np.random.randn(nF, nP)

LPop =  cTrue.T@Pop + bTrue
idx_pos = np.argwhere(LPop>0)
idx_neg = np.argwhere(LPop<0)

G1 = Pop[:, idx_pos[:,1]]  # create the populations based on their L-value
G2 = Pop[:, idx_neg[:,1]]  # create the populations based on their L-value

[cEst, bEst, tAdjust] =  buildLinClass(G1,G2)

# max(abs(tAdjust))      # should be 0 (or very close)
f1Min = np.min(Pop[0,:])  # minimum age
f1Max = np.max(Pop[0,:])  # maximum age
f2Min = np.min(Pop[1,:])  # minimum number of movies
f2Max = np.max(Pop[1,:])  # maximum number of movies
```

```
plt.plot(np.array([f1Min-1, f1Max+1]), -(cEst[0]*np.array([f1Min-1, f1Max+1])+
bEst)/cEst[1],'b')
plt.plot(G1[0,:],G1[1,:],'b*')
plt.plot(G2[0,:],G2[1,:],'ro')
plt.xlim([f1Min-0.1, f1Max+0.1])
plt.ylim([f2Min-0.1, f2Max+0.1])
plt.axis('equal')
```

```
     pcost       dcost       gap    pres   dres   k/t
 0:  1.7618e+01  1.6518e+02  6e+02  2e+00  1e+01  1e+00
 1:  2.2760e+01  6.7828e+01  1e+02  7e-01  4e+00  2e+00
 2:  1.9495e+01  3.5980e+01  4e+01  2e-01  1e+00  7e-01
 3:  1.5927e+01  2.5486e+01  3e+01  1e-01  8e-01  5e-01
 4:  1.3028e+01  1.9291e+01  2e+01  9e-02  5e-01  3e-01
 5:  1.1424e+01  1.6121e+01  2e+01  7e-02  4e-01  3e-01
 6:  9.6467e+00  1.2916e+01  2e+01  5e-02  3e-01  2e-01
 7:  8.4769e+00  1.0780e+01  1e+01  3e-02  2e-01  1e-01
 8:  8.0335e+00  9.9601e+00  2e+01  3e-02  2e-01  1e-01
 9:  6.8884e+00  8.2197e+00  1e+01  2e-02  1e-01  6e-02
10:  5.9138e+00  6.9770e+00  1e+01  2e-02  9e-02  5e-02
11:  4.7633e+00  5.5146e+00  1e+01  1e-02  6e-02  4e-02
12:  4.0980e+00  4.6701e+00  1e+01  8e-03  5e-02  4e-02
13:  3.6011e+00  4.0977e+00  1e+01  7e-03  4e-02  4e-02
14:  2.0353e+00  2.2795e+00  5e+00  3e-03  2e-02  2e-02
15:  1.8269e+00  2.0455e+00  4e+00  3e-03  2e-02  2e-02
16:  1.7032e+00  1.8837e+00  4e+00  2e-03  1e-02  2e-02
17:  1.4293e+00  1.5772e+00  4e+00  2e-03  1e-02  2e-02
18:  1.1331e-01  1.2460e-01  3e-01  2e-04  9e-04  1e-03
19:  1.1557e-03  1.2710e-03  3e-03  2e-06  9e-06  1e-05
20:  1.1558e-05  1.2711e-05  3e-05  2e-08  9e-08  1e-07
21:  1.1558e-07  1.2711e-07  3e-07  2e-10  9e-10  1e-09
22:  1.1558e-09  1.2711e-09  3e-09  2e-12  9e-12  1e-11
Optimal solution found.
```

Out[16]:
```
(-4.363525303919237, 3.5720022196115613, -3.027886548781723, 2.44164216257022
7)
```



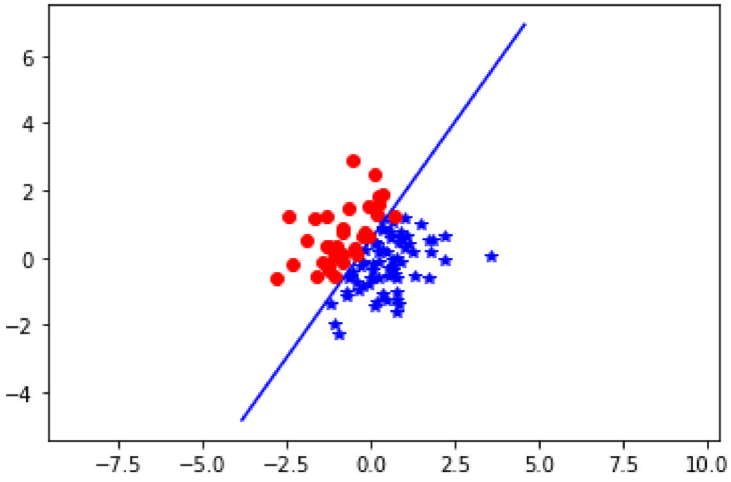
We can add some random noise to 10\% of the data, so that outliers appear. Now the classification will likely not be perfect, but will still make good sense. Run this several times to gain intuition.

In [17]:

```
nF = 2
nP = 100
nOut = np.int(0.1*nP)
cTrue = np.random.randn(nF,1)
bTrue = np.random.randn(1,1)
Pop = np.random.randn(nF, nP)
Noise = np.asarray([np.random.randn(1) if i < nOut else 0.0 for i in range(nP
)], dtype=object)

LPop =  cTrue.T@Pop + bTrue + Noise  #corrupt some L-values with noise
idx_pos = np.argwhere(LPop>0)
idx_neg = np.argwhere(LPop<0)

G1 = Pop[:, idx_pos[:,1]]  # create the populations based on their L-value
G2 = Pop[:, idx_neg[:,1]]  # create the populations based on their L-value

[cEst, bEst, tAdjust] =  buildLinClass(G1,G2)

max(abs(tAdjust))      # should be 0 (or very close)
f1Min = min(Pop[0,:])  # minimum age
f1Max = max(Pop[0,:])  # maximum age
f2Min = min(Pop[1,:])  # minimum number of movies
f2Max = max(Pop[1,:])  # maximum number of movies

plt.plot(np.array([f1Min-1, f1Max+1]), -(cEst[0]*np.array([f1Min-1, f1Max+1])+
bEst)/cEst[1],'b')
plt.plot(G1[0,:],G1[1,:],'b*')
plt.plot(G2[0,:],G2[1,:],'ro')
plt.xlim([f1Min-0.1, f1Max+0.1])
plt.ylim([f2Min-0.1, f2Max+0.1])
plt.axis('equal')
```

```
     pcost       dcost       gap    pres   dres   k/t
 0:  1.6634e+01  1.8488e+02  7e+02  2e+00  1e+01  1e+00
 1:  2.0021e+01  7.3037e+01  1e+02  7e-01  4e+00  2e+00
 2:  1.9783e+01  5.7388e+01  1e+02  5e-01  3e+00  2e+00
 3:  1.5621e+01  2.3849e+01  2e+01  1e-01  6e-01  5e-01
 4:  1.3577e+01  1.9743e+01  2e+01  8e-02  5e-01  4e-01
 5:  1.1008e+01  1.5159e+01  2e+01  6e-02  3e-01  3e-01
 6:  8.4101e+00  1.0396e+01  1e+01  3e-02  2e-01  1e-01
 7:  6.7602e+00  7.3460e+00  3e+00  8e-03  5e-02  3e-02
 8:  6.0899e+00  6.1665e+00  4e-01  1e-03  6e-03  3e-03
 9:  6.0024e+00  6.0135e+00  6e-02  2e-04  9e-04  5e-04
10:  6.0030e+00  6.0130e+00  7e-02  1e-04  8e-04  5e-04
11:  5.9917e+00  5.9981e+00  5e-02  9e-05  5e-04  3e-04
12:  5.9791e+00  5.9797e+00  5e-03  8e-06  5e-05  2e-05
13:  5.9773e+00  5.9773e+00  5e-05  9e-08  5e-07  2e-07
14:  5.9773e+00  5.9773e+00  5e-07  9e-10  5e-09  2e-09
Optimal solution found.
```

Out[17]:
```
(-4.238493470792102, 4.976850120593393, -5.4135893475254235, 7.50302595968994
3)
```



8. Regression with $\|\cdot\|_1$, $\|\cdot\|_\infty$, and linear constraints:

Suppose $A_1 \in \mathbf{R}^{m\times n}$ and $b_1 \in \mathbf{R}^m$, while $A_\infty \in \mathbf{R}^{p\times n}$ and $b_\infty \in \mathbf{R}^p$. Finally $A_c \in \mathbf{R}^{q\times n}$ and $b_c \in \mathbf{R}^q$. Consider the optimization problem

$$\begin{aligned} \min_{x\in\mathbf{R}^n} \ & \|A_1 x - b_1\|_1 + \|A_\infty x - b_\infty\|_\infty \\ \text{s.t. } & A_c x \leq b_c \end{aligned}$$

Part (a)

Use `cvxopt` to write a function `reg1Inf`, with function declaration line

In [18]:
```
from cvxopt import matrix, solvers
def reg1Inf(A1, b1, Ainf, binf, Ac, bc):
    return xOpt, J
```

In [19]:
```
from cvxopt import matrix, solvers

def reg1Inf(A1, b1, Ainf, binf, Ac, bc):

    c = np.concatenate([np.zeros(np.size(Ac,1)), np.ones(np.size(A1,0)), np.ar
ray((1,))],  axis = 0)
    A = np.concatenate([np.concatenate([A1, -np.eye(np.size(A1,0)), np.zeros((
np.size(A1,0),1))], axis = 1),
         np.concatenate([-A1, -np.eye(np.size(A1,0)), np.zeros((np.size(A1,0),
1))], axis = 1),
         np.concatenate([Ainf, np.zeros((np.size(Ainf,1), np.size(A1,1))), -np
.ones((np.size(Ainf,0),1))], axis = 1),
         np.concatenate([-Ainf, np.zeros((np.size(Ainf,0), np.size(A1,0))), -n
p.ones((np.size(Ainf,0),1))], axis = 1),
         np.concatenate([Ac, np.zeros((np.size(Ac,0), np.size(A1,0))), np.zero
s((np.size(Ac,0),1))], axis = 1)],  axis = 0)
    b = np.concatenate([b1, -b1, binf, -binf, bc], axis = 0)

    c = cvxopt.matrix(c, tc='d')
    A = cvxopt.matrix(A, tc='d')
    b = cvxopt.matrix(b, tc='d')


    sol = cvxopt.solvers.lp(c,A,b)
    xOpt = sol['x']
    J = sol['primal objective']

    return xOpt, J
```

which reformulates (using slack variables) into a linear program, and calls **`cvxopt.solvers.lp`** to get the solution. If the problem is infeasible, then **`xOpt`** should return empty and `J` should return as **`inf`**. **Hint:** Rewatch the videos on linear programming, especially the last few, which talk about $\|\cdot\|_1$ and $\|\cdot\|_\infty$ regression problems, and how to reformulate as LPs. This problem combines the two ideas, as well as imposing additional linear inequality constraints.

Part (b)

Test your code on the following small example

$$\begin{aligned} \min \quad & \| \begin{bmatrix} z_1 \\ z_2 + 5 \end{bmatrix} \|_1 + \| \begin{bmatrix} z_1 - 2 \\ z_2 \end{bmatrix} \|_\infty \\ \text{subject to} \quad & 3z_1 + 2z_2 \leq -3 \\ & 0 \leq z_1 \leq 2 \\ & -2 \leq z_2 \leq 3 \end{aligned}$$

In [20]:
```
A1 = np.eye(2)
b1 = np.array([0, -5])
Ainf = np.eye(2)
binf = np.array([2, 0])
Ac = np.array([[3, 2], [1, 0], [0, 1], [-1, 0], [0, -1]])
bc = np.array([-3, 2, 3, 0, 2])
# Calling the function
xOpt, J = reg1Inf(A1, b1, Ainf, binf, Ac, bc)
print('x*=', xOpt)
print('J*=', J)
```

```
     pcost       dcost       gap    pres   dres   k/t
 0:  0.0000e+00 -4.0000e+00  7e+01  2e+00  4e+00  1e+00
 1:  5.0224e+00  4.9463e+00  4e+00  1e-01  3e-01  4e-01
 2:  5.0369e+00  5.0287e+00  3e-01  1e-02  3e-02  3e-02
 3:  4.9988e+00  4.9985e+00  1e-02  6e-04  1e-03  2e-03
 4:  5.0000e+00  5.0000e+00  1e-04  6e-06  1e-05  2e-05
 5:  5.0000e+00  5.0000e+00  1e-06  6e-08  1e-07  2e-07
 6:  5.0000e+00  5.0000e+00  1e-08  6e-10  1e-09  2e-09
Optimal solution found.
x*= [ 8.18e-10]
[-2.00e+00]
[ 1.16e-10]
[ 3.00e+00]
[ 2.00e+00]

J*= 4.99999999882211
```