Solution 8

EE 578B - Winter 2021

Due Date: Sunday, Mar 14th, 2021 @ 11:59 PM

Newton's Method

Consider the following unconstrained quadratic program.

$$\min_x \quad f(x) = \frac{1}{2}x^T Q x + c^T x$$

for $x \in \mathbb{R}^2$

$$Q = \begin{bmatrix} 101 & -99 \\ -99 & 101 \end{bmatrix}, \qquad c = \begin{bmatrix} 2 \\ 2 \end{bmatrix}$$

Perform regular first order gradient descent using the update equation

$$x^+ = x - \gamma \frac{\partial f}{\partial x}^T$$

starting from the initial condition $x = \begin{bmatrix} 10 & 0 \end{bmatrix}^T$ with a fixed step size γ. (Pick the step size.)
Note: if you want you can use a more sophisticated step-size method.

In [232]:

```
import numpy as np
import numpy.linalg as mat
import matplotlib.pyplot as plt

Q = np.array([[101.,-99.],[-99.,101.]]);
c = np.array([2.,2.]);

def df(x): return x@Q + c
def d2f(x): return Q

n = 2;
kmax = 1000;
ks = np.array(list(range(kmax)));
gams = [0.01,0.008,0.005];
x0= np.array([10.,0.]);
x = np.zeros((len(gams),kmax,n));
for i in range(len(gams)):
    x[i,0] = x0;
    for k in range(kmax-1):
        x[i,k+1] = x[i,k] - gams[i]*df(x[i,k]);

```

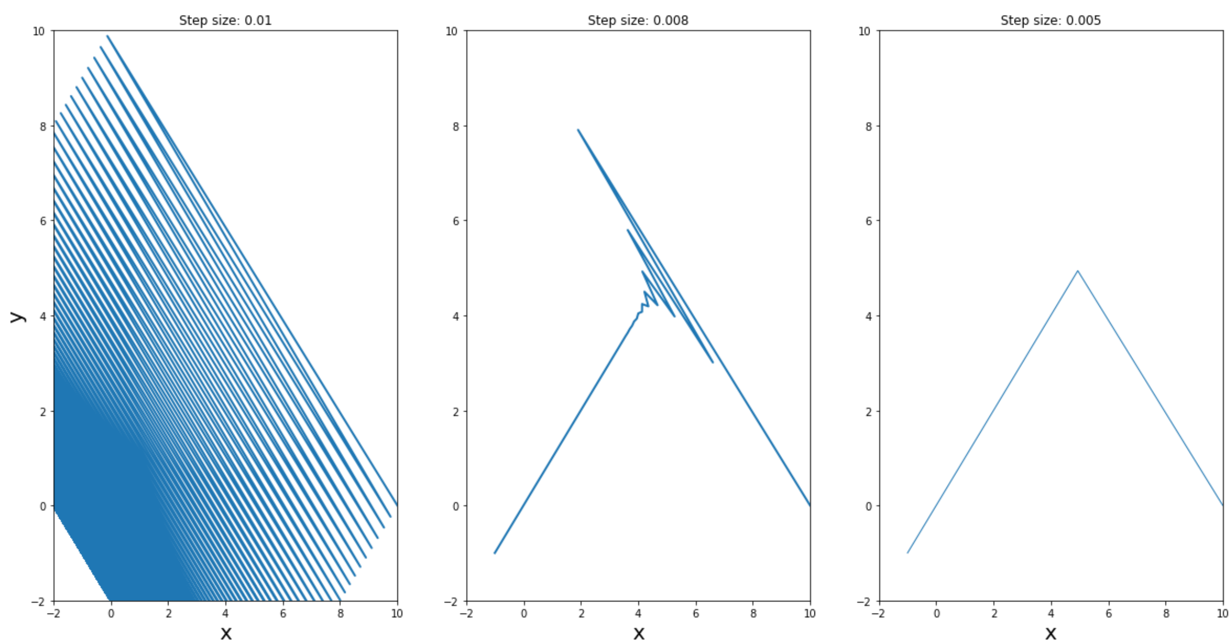
Plot the trajectory of x and describe the behavior intuitively.

In [233]:

```


fig,ax = plt.subplots(1,3,figsize=(20,10));

ax[0].set_ylim([-2,10]); ax[0].set_xlim([-2,10]);
ax[0].set_xlabel('x',fontsize=20)
ax[0].set_ylabel('y',fontsize=20)
ax[0].set_title('Step size: '+str(gams[0]))

ax[1].set_xlim([-2,10]); ax[1].set_ylim([-2,10])
ax[1].set_xlabel('x',fontsize=20)
ax[1].set_title('Step size: '+str(gams[1]))


ax[2].set_xlim([-2,10]); ax[2].set_ylim([-2,10])
ax[2].set_xlabel('x',fontsize=20)
ax[2].set_title('Step size: '+str(gams[2]))


ax[0].plot(x[0,:,0],x[0,:,1],linewidth=2.)
ax[1].plot(x[1,:,0],x[1,:,1],linewidth=2.)
ax[2].plot(x[2,:,0],x[2,:,1],linewidth=1.)

```

Out[233]:

```
[<matplotlib.lines.Line2D at 0x113682908>]
```



Discussion: Note the sensitivity of the convergence to the step size as well as the oscillations in the solution

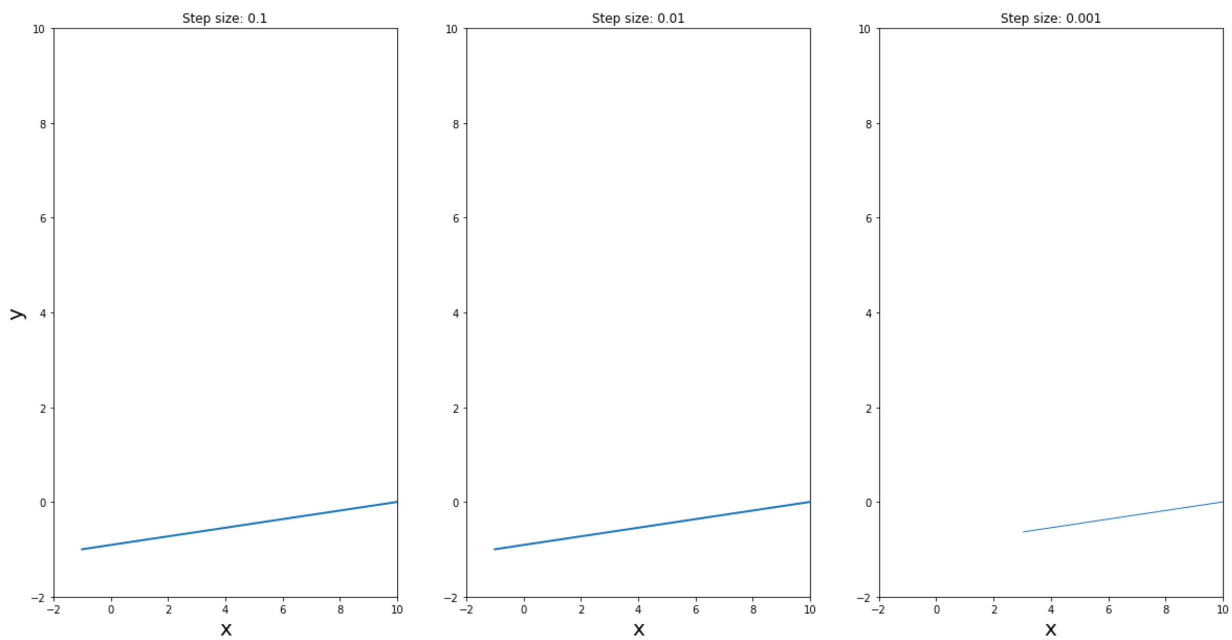
Perform Newton's Method starting from the same initial condition with the same step size.

$$x^{+} = x - \gamma \left(\frac{\partial^2 f}{\partial x^2} \right)^{-1} \frac{\partial f}{\partial x}^{T}$$

In [234]:
```
gams = [0.01,0.008,0.005];
gams = [0.1,0.01,0.001];
x0= np.array([10.,0.]);
x = np.zeros((len(gams),kmax,n));
for i in range(len(gams)):
    x[i,0] = x0;
    for k in range(kmax-1):
        x[i,k+1] = x[i,k] - gams[i]*mat.inv(Q)@df(x[i,k]);

```

In [235]:
```

fig,ax = plt.subplots(1,3,figsize=(20,10));

ax[0].set_ylim([-2,10]); ax[0].set_xlim([-2,10]);
ax[0].set_xlabel('x',fontsize=20)
ax[0].set_ylabel('y',fontsize=20)
ax[0].set_title('Step size: '+str(gams[0]))

ax[1].set_xlim([-2,10]); ax[1].set_ylim([-2,10])
ax[1].set_xlabel('x',fontsize=20)
ax[1].set_title('Step size: '+str(gams[1]))


ax[2].set_xlim([-2,10]); ax[2].set_ylim([-2,10])
ax[2].set_xlabel('x',fontsize=20)
ax[2].set_title('Step size: '+str(gams[2]))


ax[0].plot(x[0,:,0],x[0,:,1],linewidth=2.)
ax[1].plot(x[1,:,0],x[1,:,1],linewidth=2.)
ax[2].plot(x[2,:,0],x[2,:,1],linewidth=1.)

```

Out[235]: [<matplotlib.lines.Line2D at 0x112524e48>]



Plot the trajectory of x and compare the qualitative performance to the first-order gradient descent method.

Discussion: Note the success of the convergence for a wide range of step size and the direct path of descent.

Now consider the following constrained convex program

$$\begin{aligned} \min_{x} \quad & 10x_1^4 + 2x_2^4 + 2x_3^4 + 2x_4^4 \\ \text{s.t.} \quad & Ax = b \end{aligned}$$

for $x \in \mathbb{R}^4$ and

$$A = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix}, \qquad b = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

(PTS:0-2) Use Newton's method to perform gradient descent on this constrained optimization problem to solve for the optimal $x \in \mathbb{R}^n$ and the optimal dual variable $v \in \mathbb{R}^2$

In [236]:

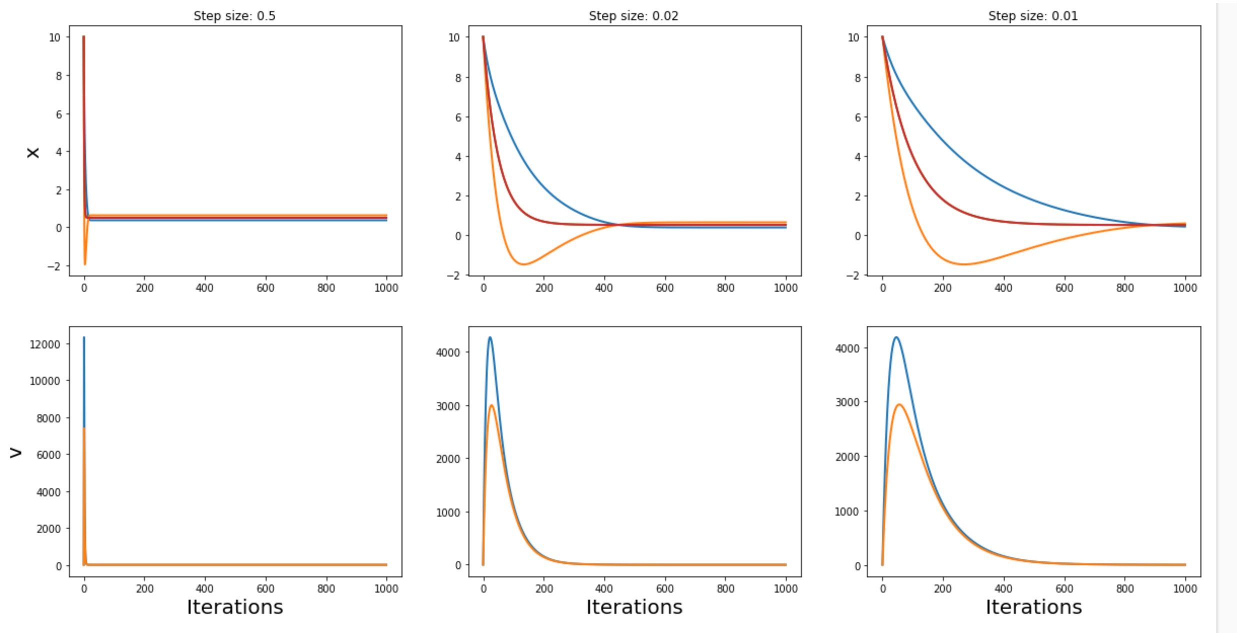
```python
import numpy as np
import numpy.linalg as mat
import matplotlib.pyplot as plt

A = np.array([[1.,1.,0.,0.],
              [0.,0.,1.,1.]]);
b = np.array([1.,1.]);

def df(x):  return np.array([40.*x[0]**3,8.*x[1]**3,8.*x[2]**3,8.*x[3]*
def d2f(x): return np.diag(np.array([120.*x[0]**2,24.*x[1]**2,24.*x[2]*

def dL(x,v):  return np.hstack([df(x)+v@A,A@x-b])
def d2L(x,v): return np.block([[d2f(x),A.T],[A, np.zeros([2,2])]]);

n = 4;
kmax = 1000;
ks = np.array(list(range(kmax)));
gams = [0.5,0.02,0.01];
x0= np.array([10.,10.,10.,10.]);
xv= np.zeros((len(gams),kmax,n+2));
for i in range(len(gams)):
    xv[i,0,:n] = x0;
    for k in range(kmax-1):
        x = xv[i,k,:n]; v = xv[i,k,n:];
        xv[i,k+1] = xv[i,k] - gams[i]*mat.inv(d2L(x,v))@dL(x,v);
```

In [237]:

```python
fig,ax = plt.subplots(2,3,figsize=(20,10));


ax[0][0].set_title('Step size: '+str(gams[0]))
# ax[0][0].set_ylim([-2,10]);
# ax[0][0].set_xlim([-2,10]);
ax[0][0].set_ylabel('x',fontsize=20)
# ax[1][0].set_xlim([-2,10]);
# ax[1][0].set_ylim([-2,10])
ax[1][0].set_xlabel('Iterations',fontsize=20)
ax[1][0].set_ylabel('v',fontsize=20)

ax[0][1].set_title('Step size: '+str(gams[1]))
# ax[0][1].set_ylim([-2,10]);
# ax[0][1].set_xlim([-2,10]);
# ax[1][1].set_xlim([-2,10]);
# ax[1][1].set_ylim([-2,10])
ax[1][1].set_xlabel('Iterations',fontsize=20)

ax[0][2].set_title('Step size: '+str(gams[2]))
# ax[0][2].set_ylim([-2,10]);
# ax[0][2].set_xlim([-2,10]);
# ax[1][2].set_xlim([-2,10]);
# ax[1][2].set_ylim([-2,10])
ax[1][2].set_xlabel('Iterations',fontsize=20)



ax[0][0].plot(xv[0,:,:n],linewidth=2.)
ax[1][0].plot(xv[0,:,n:],linewidth=2.)

ax[0][1].plot(xv[1,:,:n],linewidth=2.)
ax[1][1].plot(xv[1,:,n:],linewidth=2.)

ax[0][2].plot(xv[2,:,:n],linewidth=2.)
ax[1][2].plot(xv[2,:,n:],linewidth=2.)
```

Out[237]:

```
[<matplotlib.lines.Line2D at 0x11125ebe0>,
 <matplotlib.lines.Line2D at 0x103983da0>]
```



(PTS:0-2) Compare $\frac{\partial f}{\partial x}$ and $v^T A$ at optimum. How do they relate?

In [238]:

```
print('Step size: ',gams[0])
print('df/dx: ', np.round(df(xv[0,-1,:n]),3))
print('v^TA: ', np.round(xv[0,-1,n:]@A,3))
print('')
print('Step size: ',gams[1])
print('df/dx: ', np.round(df(xv[1,-1,:n]),3))
print('v^TA: ', np.round(xv[1,-1,n:]@A,3))
print('')
print('Step size: ',gams[2])
print('df/dx: ', np.round(df(xv[2,-1,:n]),3))
print('v^TA: ', np.round(xv[2,-1,n:]@A,3))

print('Note that the last step size hasnt converged yet.')
```

```
Step size:  0.5
df/dx:  [2.01 2.01 1.   1.  ]
v^TA:  [-2.01 -2.01 -1.   -1.  ]

Step size:  0.02
df/dx:  [2.01 2.01 1.   1.  ]
v^TA:  [-2.01 -2.01 -1.   -1.  ]

Step size:  0.01
df/dx:  [2.991 1.557 1.002 1.002]
v^TA:  [-1.2   -1.2   -0.649 -0.649]
Note that the last step size hasnt converged yet.
```

Interior Point Method

Consider the constrained optimization problem

$$\begin{aligned} \min_x \quad & f(x) = 10x_1^4 + x_2^4 \\ \text{s.t.} \quad & h(x) \leq 0 \end{aligned}$$

for $x \in \mathbb{R}^2$

$$h(x) = \begin{bmatrix} (x - \mathbf{1})^T Q_1 (x - \mathbf{1}) - 1 \\ (x - \mathbf{1})^T Q_2 (x - \mathbf{1}) - 1 \end{bmatrix}$$

where $\mathbf{1}^T = [1 \ 1]$

$$Q_1 = \begin{bmatrix} 3 & 1 \\ 1 & 3 \end{bmatrix}, \quad Q_2 = \begin{bmatrix} 3 & -1 \\ -1 & 3 \end{bmatrix}$$

(PTS:0-2) Replace the objective function $f(x)$ with $tf(x)$ for some $t > 0$. Replace the inequality constraints with equality constraints of the form $h_i(x) = s_i$ and add barrier function terms of the form $\mu \ln(s_i)$ to the objective for some $\mu > 1$.

$$\begin{aligned} \min_x \quad & f(x) = t(10x_1^4 + x_2^4) - \mu \sum_i \ln(s_i) \\ \text{s.t.} \quad & h(x) + s = 0, \end{aligned}$$

(PTS:0-2) Write the Lagrangian for this new optimization problem (with barrier functions) with dual variables $v \in \mathbb{R}^2$ for the equality constraints.

$$L(x, s, v) = t(10x_1^4 + x_2^4) - \mu \sum_i \ln(s_i) + v^T(h(x) + s)$$

(PTS:0-4) Write code to perform Newton's method for gradient descent to solve for the optimal $x \in \mathbb{R}^2$, $s \in \mathbb{R}^2$, and $v \in \mathbb{R}^2$ for a given value of t.

In [159]:

```
import numpy as np
import numpy.linalg as mat
import matplotlib.pyplot as plt

mu = 1.1; # 2.0 # 4.0
n = 2;

Q1 = np.array([[3.,1.],[1.,3.]]);
Q2 = np.array([[3.,-1.],[-1.,3.]]);

def df(x,t=1):  return t*np.array([40.*x[0]**3,4.*x[1]**3])
def d2f(x,t=1): return t*np.diag(np.array([120.*x[0]**2,12.*x[1]**2]))


def h(x):   return np.block([(x-np.ones(n))@Q1@(x-np.ones(n))-1.,
                             (x-np.ones(n))@Q2@(x-np.ones(n))-1.]);
def dh(x):  return np.block([[(x-np.ones(n))@Q1],
                             [(x-np.ones(n))@Q2]]);
def d2h(x): return np.array([Q1,Q2]);


def dL(z,t=1):
    x = z[:2]; s = z[2:4]; v = z[4:];
    return np.hstack([df(x)+v@dh(x),v-mu*(1./s),h(x)+s])

def d2L(z,t=1):
    x = z[:2]; s = z[2:4]; v = z[4:];
    H = d2f(x,t)+v[0]*d2h(x)[0]+v[1]*d2h(x)[1];
    return np.block([[H              ,np.zeros((2,2))  ,   dh(x).T],
                     [np.zeros((2,2)),np.diag(mu/(s*s)), np.eye(2)],
                     [dh(x)          , np.eye(2)       ,np.zeros((2,2))


def newtons(grad,Hess,z0,t,tol=0.0001,maxiter=100,gam=0.01):
    # - grad:  (function)
    # - H :  Hessian (function)
    # - z0:  state (init. condition)
    # - t: parameter
    # - tol: convergence tolerance value
    # - maxiter: max # of iteration:
    # - gam: step size
    # - returns z: trajectory
    z = [z0]; k = 0; converged = False;
    while not(converged):
        z.append(z[k]-gam*mat.inv(Hess(z[k],t))@grad(z[k],t))
        k = k + 1;
        if (mat.norm(z[-1]-z[-2]) <= tol) or (k >= maxiter): converged
    #print('total iterations: ',k)
    return np.array(z)

```

(PTS:0-4) For each value of t, run Newton's method till $|x - x^+| < \delta$ for some tolerance $\delta > 0$. After x converges, update t as $t^+ = \mu t$ and repeat solving for x. (Note, it's important that $\mu > 1$ so that t will grow, ie. the objective gets more weight as you approach the

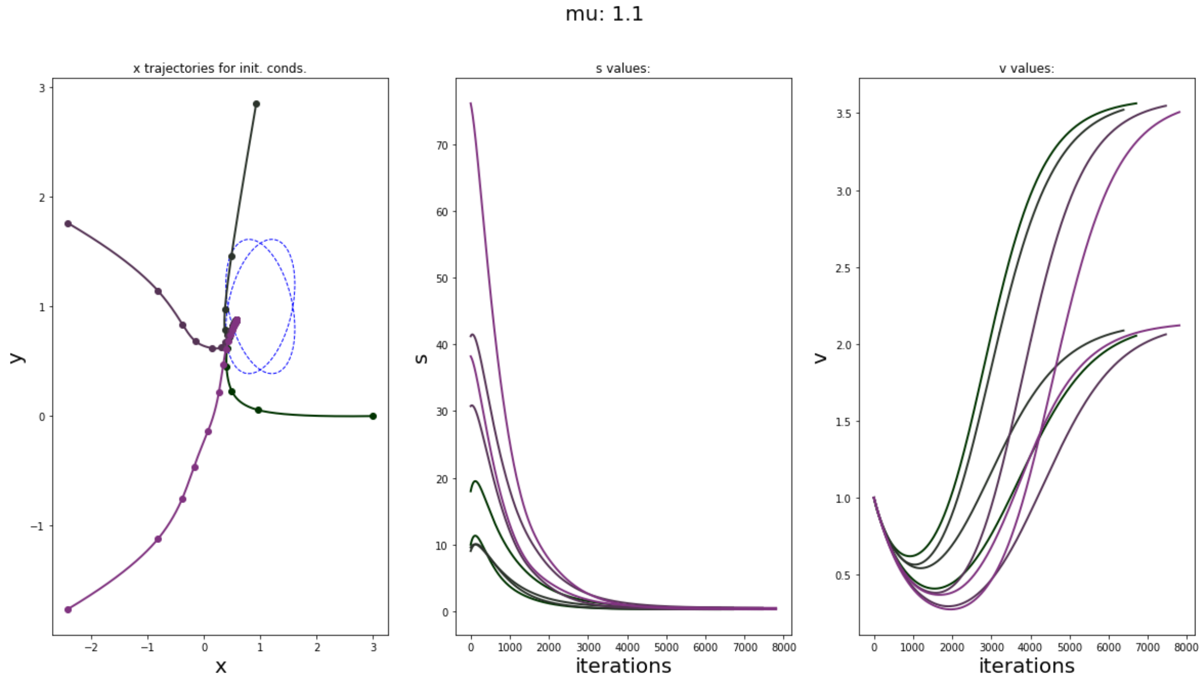
boundary.) Iterate on this process till the optimal x is found. How does this method perform for different values of μ?

In [160]:

```


tmax = 20;

### Number of initial conditions...
num_inits = 6;

## creating initial x's
x0s = 3.*np.block([[np.cos(np.linspace(0,2*np.pi,num_inits))],
                   [np.sin(np.linspace(0,2*np.pi,num_inits))]]).T;

## full trajectories z = [x,s,v]
z = list(np.empty(x0s.shape[0]))
z_outer = np.zeros([x0s.shape[0],tmax,6]);
ts = np.zeros([x0s.shape[0],tmax]);


for i in range(x0s.shape[0]):
    x0 = x0s[i];
    z_outer[i,0] = np.hstack([x0,h(x0),np.ones(2)]);
    ts[i,0] = 0.1

    z[i] = np.zeros([0,6]);
    for t in range(tmax):
        z[i] = np.block([[z[i]],[newtons(dL,d2L,z_outer[i,t],ts[i,t],ma
        if t < tmax-1:
            z_outer[i,t+1] = z[i][-1];
            ts[i,t+1] = mu*ts[i,t]
```

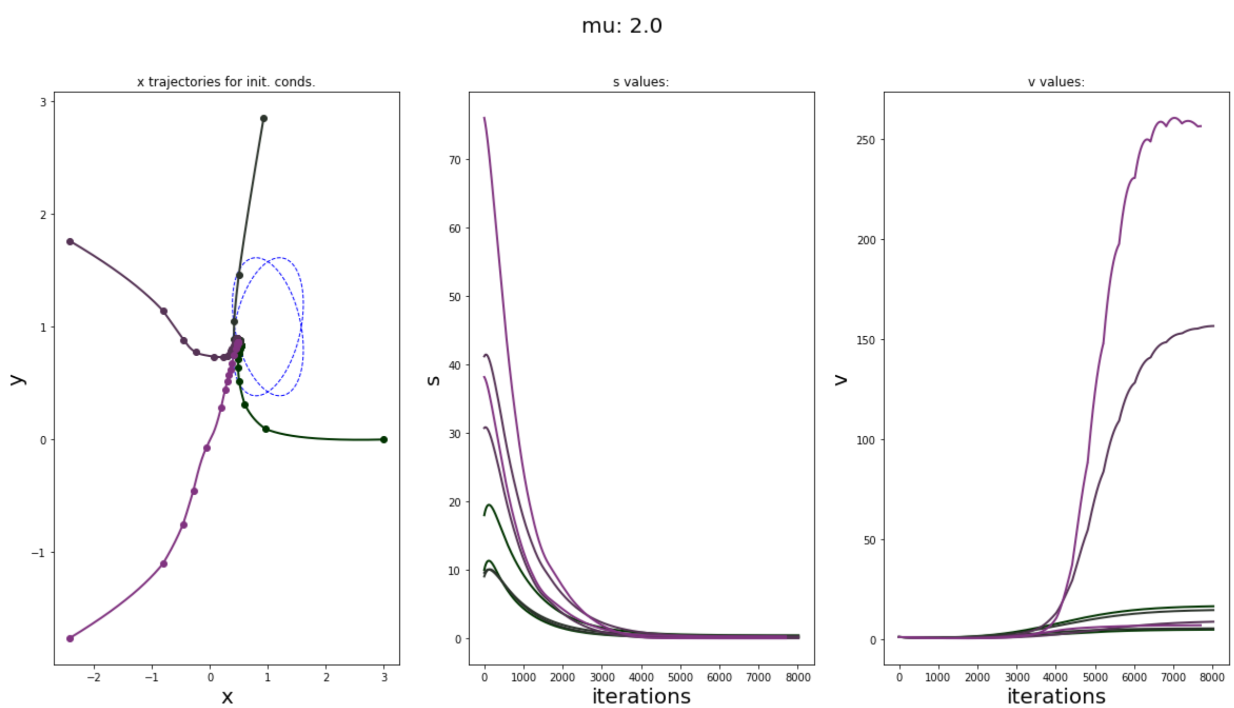
In [155]:

```
import scipy.linalg as smat
from matplotlib.patches import Polygon

# drawing an ellipse shape
def ellipse(mu,P): #draws an ellipse with shape P (covariance matrix) a
    thetas = np.linspace(0,2*np.pi,100);
    Phalf = smat.sqrtm(P);
    return mu+np.dot(Phalf,np.array([np.cos(thetas),np.sin(thetas)])).T


fig,ax = plt.subplots(1,3,figsize=(20,10));
fig.suptitle('mu: '+str(mu),fontsize=20)



#ax[0].set_ylim([-2,10]); ax[0].set_xlim([-2,10]);
ax[0].set_xlabel('x',fontsize=20)
ax[0].set_ylabel('y',fontsize=20)
ax[0].set_title('x trajectories for init. conds.')

ax[1].set_ylabel('s',fontsize=20)
ax[1].set_xlabel('iterations',fontsize=20)
ax[1].set_title('s values:')

ax[2].set_ylabel('v',fontsize=20)
ax[2].set_xlabel('iterations',fontsize=20)
ax[2].set_title('v values:')


ax[0].add_patch(Polygon(ellipse(np.ones(2),mat.inv(Q1)),closed=True,fil
ax[0].add_patch(Polygon(ellipse(np.ones(2),mat.inv(Q2)),closed=True,fil

for i in range(central_path.shape[0]):

    color = [float(i)/num_inits,0.2,float(i)/num_inits]
    ax[0].plot(z_outer[i,:,0],z_outer[i,:,1],'o',color=color);
    ax[0].plot(z[i][:,0],z[i][:,1],'-',linewidth=2.,color=color);

    ax[1].plot(z[i][:,2],linewidth=2.,color=color);
    ax[1].plot(z[i][:,3],linewidth=2.,color=color);

    ax[2].plot(z[i][:,4],linewidth=2.,color=color);
    ax[2].plot(z[i][:,5],linewidth=2.,color=color);

```



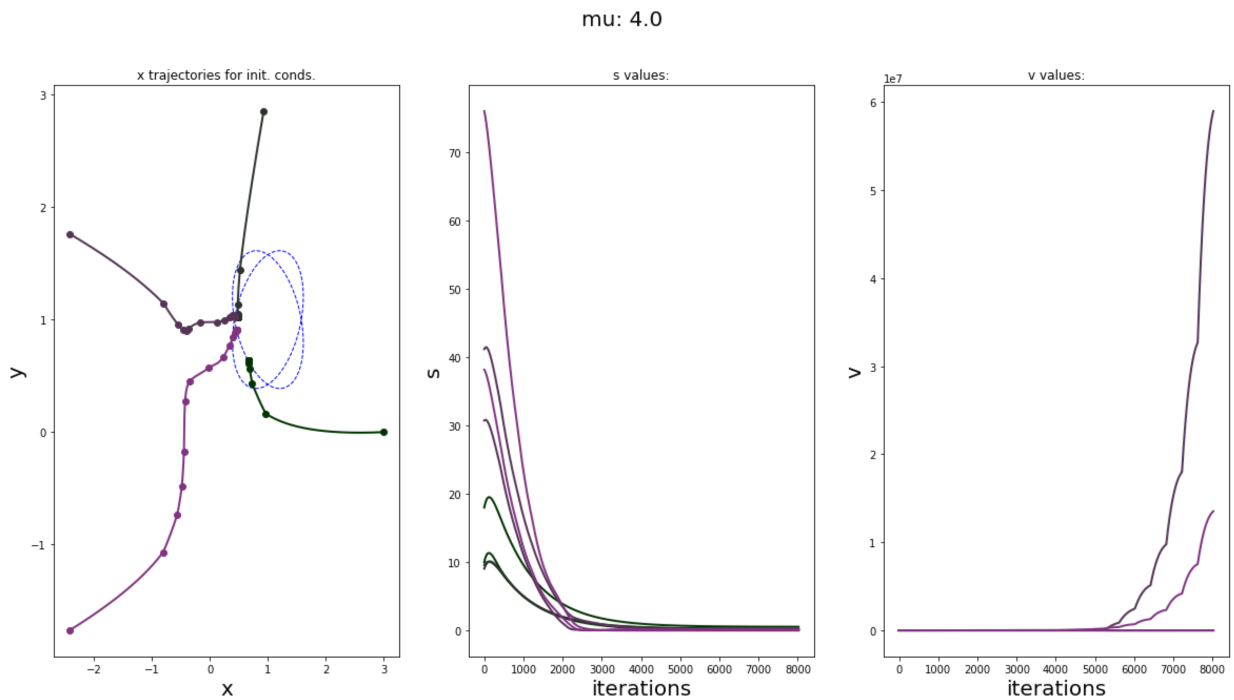
In [158]:

```
fig,ax = plt.subplots(1,3,figsize=(20,10)); fig.suptitle('mu: '+str(mu)
ax[0].set_xlabel('x',fontsize=20); ax[0].set_ylabel('y',fontsize=20); a
ax[1].set_ylabel('s',fontsize=20); ax[1].set_xlabel('iterations',fontsi
ax[2].set_ylabel('v',fontsize=20); ax[2].set_xlabel('iterations',fontsi
ax[0].add_patch(Polygon(ellipse(np.ones(2),mat.inv(Q1)),closed=True,fil
ax[0].add_patch(Polygon(ellipse(np.ones(2),mat.inv(Q2)),closed=True,fil
for i in range(central_path.shape[0]):
    color = [float(i)/num_inits,0.2,float(i)/num_inits]
    ax[0].plot(z_outer[i,:,0],z_outer[i,:,1],'o',color=color); ax[0].pl
    ax[1].plot(z[i][:,2],linewidth=2.,color=color); ax[1].plot(z[i][:,3
    ax[2].plot(z[i][:,4],linewidth=2.,color=color);  ax[2].plot(z[i][:,
```



In [161]:

```

fig,ax = plt.subplots(1,3,figsize=(20,10)); fig.suptitle('mu: '+str(mu)
ax[0].set_xlabel('x',fontsize=20); ax[0].set_ylabel('y',fontsize=20); a
ax[1].set_ylabel('s',fontsize=20); ax[1].set_xlabel('iterations',fontsi
ax[2].set_ylabel('v',fontsize=20); ax[2].set_xlabel('iterations',fontsi
ax[0].add_patch(Polygon(ellipse(np.ones(2),mat.inv(Q1)),closed=True,fil
ax[0].add_patch(Polygon(ellipse(np.ones(2),mat.inv(Q2)),closed=True,fil
for i in range(central_path.shape[0]):
    color = [float(i)/num_inits,0.2,float(i)/num_inits]
    ax[0].plot(z_outer[i,:,0],z_outer[i,:,1],'o',color=color); ax[0].pl
    ax[1].plot(z[i][:,2],linewidth=2.,color=color); ax[1].plot(z[i][:,3
    ax[2].plot(z[i][:,4],linewidth=2.,color=color);  ax[2].plot(z[i][:,

```



Simplex Method - Row Geometry

Consider the following linear program for $z \in \mathbb{R}^3$

$$\begin{aligned} \max_x \quad & c^T z \\ \text{s.t.} \quad & Cz \leq d, \ x \geq 0 \end{aligned}$$

where

$$c = \begin{bmatrix} 1 & 1 & 1 \end{bmatrix}, \qquad C = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix}, \quad d = \begin{bmatrix} 2 \\ 2 \\ 2 \\ 3 \\ 3 \\ 3 \end{bmatrix}$$

(PTS:0-2) Use a slack variable $s \in \mathbb{R}^6$ to rewrite the LP in standard form for the simplex method

$$\begin{aligned} \max_x \quad & r^T x \\ \text{s.t.} \quad & Ax = b,\ x \geq 0 \end{aligned}$$

What is x? A? b? What feasible x corresponds to $z = 0$?

problem with slack variables

$$\begin{aligned} \max_x \quad & c^T z \\ \text{s.t.} \quad & Cz + s = d,\ x \geq 0,\ s \geq 0 \end{aligned}$$

$$\begin{aligned} \max_x \quad & \begin{bmatrix} c^T & \mathbf{0}^T \end{bmatrix} \begin{bmatrix} z \\ s \end{bmatrix} \\ \text{s.t.} \quad & \begin{bmatrix} C & I \end{bmatrix} \begin{bmatrix} z \\ s \end{bmatrix} = d,\ x \geq 0,\ s \geq 0 \end{aligned}$$

(PTS:0-2) Write a tableau for the linear program in the form

$$\begin{bmatrix} 1 & -r^T & 0 \\ \mathbf{0} & A & b \end{bmatrix}$$

In [239]:

```
## Tableau
r = np.array([1.,1.,1.,0.,0.,0.,0.,0.,0.])

C = np.array([[1.,0.,0.],
              [0.,1.,0.],
              [0.,0.,1.],
              [1.,1.,0.],
              [1.,0.,1.],
              [0.,1.,1.]])

d = np.array([2.,2.,2.,3.,3.,3.])
b = np.array([d]).T; A = np.block([C,np.eye(6)])


T = np.block([[1.,-r,0.],
              [np.zeros((6,1)),A,b]])
print('Tableau: ')
print(T)
```

```
Tableau: 
[[ 1. -1. -1. -1. -0. -0. -0. -0. -0. -0.  0.]
 [ 0.  1.  0.  0.  1.  0.  0.  0.  0.  0.  2.]
 [ 0.  0.  1.  0.  0.  1.  0.  0.  0.  0.  2.]
 [ 0.  0.  0.  1.  0.  0.  1.  0.  0.  0.  2.]
 [ 0.  1.  1.  0.  0.  0.  0.  1.  0.  0.  3.]
 [ 0.  1.  0.  1.  0.  0.  0.  0.  1.  0.  3.]
 [ 0.  0.  1.  1.  0.  0.  0.  0.  0.  1.  3.]]
```

(PTS:0-4) Starting at the initial solution $z = 0$ (vertex 0), perform pivot steps to find the optimal solution to the linear program. What is the optimal x? What is the corresponding optimal z? Which rows of the constraint $Cz \leq d$ are satisfied with equality?

NOTE: it's fine if you followed a different order of operations. there are many ways to get to the optimal solution

In [240]:

```
T = np.block([[1.,-r,0.],
              [np.zeros((6,1)),A,b]])

print('Initial solution: feasible so no need for initial tableau')
print('Initial basis columns: s1,s2,s3,s4,s5,s6')

print('')
print('Look for negative values in objective row...add in z3...')
print('b column must stay positive so choose to add row 3 (zero indexed
print('row reducing to identity column, swapping in for s3 column...')
T[0] = T[0] + T[3];
T[5] = T[5] - T[3];
T[6] = T[6] - T[3];
print('Updated Tableau:')
print(T)
print('Updated basis columns: z3,s1,s2,s4,s5,s6')



print('')
print('Look for negative values in objective row...  add in z2.')
print('b column must stay positive so choose to add row 6 (zero indexed
print('row reducing to identity column, swapping in for s6 column...')
T[0] = T[0] + T[6];
T[2] = T[2] - T[6];
T[4] = T[4] - T[6];
print('Updated Tableau:')
print(T)
print('Updated basis columns: z2,z3,s1,s2,s4,s5')

print('')
print('Look for negative values in objective row...  add in z1.')
print('b column must stay positive so choose to add row 5 (zero indexed
print('row reducing to identity column, swapping in for s5 column...')
T[0] = T[0] + T[5];
T[1] = T[1] - T[5];
T[4] = T[4] - T[5];
print('Updated Tableau:')
print(T)
print('Updated basis columns: z1,z2,z3,s1,s2,s4')

print('')
print('Look for negative values in objective row...  add back in s3.')
print('b column must stay positive so choose to add row 4 (zero indexed
print('dividing row by 2...')
T[4] = 0.5*T[4];
print('row reducing to identity column, swapping in for s4 column...')
T[0] = T[0] + T[4];
T[1] = T[1] - T[4];
T[2] = T[2] - T[4];
T[3] = T[3] - T[4];
T[5] = T[5] + T[4];
T[6] = T[6] + T[4];
print('Updated Tableau:')
print(T)
print('Updated basis columns: z1,z2,z3,s1,s2,s3')
```

```
print('')
print('')
print('No negatives in objective row.  Done.')
```

```
Initial solution: feasible so no need for initial tableau
Initial basis columns: s1,s2,s3,s4,s5,s6

Look for negative values in objective row...add in z3...
b column must stay positive so choose to add row 3 (zero indexed)...
row reducing to identity column, swapping in for s3 column...
Updated Tableau:
[[ 1. -1. -1.  0.  0.  0.  1.  0.  0.  0.  2.]
 [ 0.  1.  0.  0.  1.  0.  0.  0.  0.  0.  2.]
 [ 0.  0.  1.  0.  0.  1.  0.  0.  0.  0.  2.]
 [ 0.  0.  0.  1.  0.  0.  1.  0.  0.  0.  2.]
 [ 0.  1.  1.  0.  0.  0.  0.  1.  0.  0.  3.]
 [ 0.  1.  0.  0.  0.  0. -1.  0.  1.  0.  1.]
 [ 0.  0.  1.  0.  0.  0. -1.  0.  0.  1.  1.]]
Updated basis columns: z3,s1,s2,s4,s5,s6

Look for negative values in objective row...  add in z2.
b column must stay positive so choose to add row 6 (zero indexed)...
row reducing to identity column, swapping in for s6 column...
Updated Tableau:
[[ 1. -1.  0.  0.  0.  0.  0.  0.  0.  1.  3.]
 [ 0.  1.  0.  0.  1.  0.  0.  0.  0.  0.  2.]
 [ 0.  0.  0.  0.  0.  1.  1.  0.  0. -1.  1.]
 [ 0.  0.  0.  1.  0.  0.  1.  0.  0.  0.  2.]
 [ 0.  1.  0.  0.  0.  0.  1.  1.  0. -1.  2.]
 [ 0.  1.  0.  0.  0.  0. -1.  0.  1.  0.  1.]
 [ 0.  0.  1.  0.  0.  0. -1.  0.  0.  1.  1.]]
Updated basis columns: z2,z3,s1,s2,s4,s5

Look for negative values in objective row...  add in z1.
b column must stay positive so choose to add row 5 (zero indexed)...
row reducing to identity column, swapping in for s5 column...
Updated Tableau:
[[ 1.  0.  0.  0.  0.  0. -1.  0.  1.  1.  4.]
 [ 0.  0.  0.  0.  1.  0.  1.  0. -1.  0.  1.]
 [ 0.  0.  0.  0.  0.  1.  1.  0.  0. -1.  1.]
 [ 0.  0.  0.  1.  0.  0.  1.  0.  0.  0.  2.]
 [ 0.  0.  0.  0.  0.  0.  2.  1. -1. -1.  1.]
 [ 0.  1.  0.  0.  0.  0. -1.  0.  1.  0.  1.]
 [ 0.  0.  1.  0.  0.  0. -1.  0.  0.  1.  1.]]
Updated basis columns: z1,z2,z3,s1,s2,s4

Look for negative values in objective row...  add back in s3.
b column must stay positive so choose to add row 4 (zero indexed)...
dividing row by 2...
row reducing to identity column, swapping in for s4 column...
Updated Tableau:
[[ 1.   0.   0.   0.   0.   0.   0.   0.5  0.5  0.5  4.5]
 [ 0.   0.   0.   0.   1.   0.   0.  -0.5 -0.5  0.5  0.5]
 [ 0.   0.   0.   0.   0.   1.   0.  -0.5  0.5 -0.5  0.5]
 [ 0.   0.   0.   1.   0.   0.   0.  -0.5  0.5  0.5  1.5]
 [ 0.   0.   0.   0.   0.   0.   1.   0.5 -0.5 -0.5  0.5]
 [ 0.   1.   0.   0.   0.   0.   0.   0.5  0.5 -0.5  1.5]
 [ 0.   0.   1.   0.   0.   0.   0.   0.5 -0.5  0.5  1.5]]
```

```
Updated basis columns: z1,z2,z3,s1,s2,s3


No negatives in objective row.  Done.
```

In [241]:

```
print('Final solution:')
print('z1: ',T[5,-1])
print('z2: ',T[6,-1])
print('z3: ',T[3,-1])
print('s1: ',T[1,-1])
print('s2: ',T[2,-1])
print('s3: ',T[4,-1])
print('')

print('s4,s5,s6 all 0   =>   Last 3 rows of Cz<=d satisfied with equali
```

```
Final solution:
z1:  1.5
z2:  1.5
z3:  1.5
s1:  0.5
s2:  0.5
s3:  0.5

s4,s5,s6 all 0   =>   Last 3 rows of Cz<=d satisfied with equality
```

(PTS:0-2) What route did you follow through the polytope? You can list the route referencing the vertex numbers.

Route taken through polytope...

Vertices: 0 to 3 to 7 to 12 to 13

Simplex Method - Column Geometry

Consider the following linear program for $x \in \mathbb{R}^7$

$$\max_x \quad r^T x$$
$$\text{s.t.} \quad Ax = b,\ x \geq 0$$

where

$$A = \begin{bmatrix} A_1 & \cdots & A_7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 2 & 1 & 1 & -1 & -1 \\ 0 & 1 & 1 & 2 & -1 & 1 & 2 \end{bmatrix}$$

(PTS:0-2) Draw the columns of A as vectors in R^2.

figure given in inset of drawings...

(PTS:0-2) Suppose $b = \begin{bmatrix} -1 \\ 2 \end{bmatrix}$. Find all possible pairs of basis vectors (A_i and $A_{i'}$) such that $\begin{bmatrix} A_i & A_{i'} \end{bmatrix} \begin{bmatrix} x_i \\ x_{i'} \end{bmatrix} = b$. for $x \geq 0$. (Hint: there are 4 pairs. Drawing b with the columns of A may help.)

NOTE: the hint is wrong because I added in extra columns of A after I wrote it.

$x_7 = 1$ is a solution and thus A_7 and any other column works (6 pairs)

Without A_7, A_6 is required since the vector b must lie inbetween the two columns of A so that x_i and $x_{i'}$ can remain positive... Thus A_6 and any of the columns from A_2 to A_5 work (5 pairs).

Total: 11 pairs. (again the hint was wrong. my apologies.)

(PTS:0-2) Suppose $b = \begin{bmatrix} 3 \\ 1 \end{bmatrix}$. Find all possible pairs of basis vectors (A_i and $A_{i'}$) such that $\begin{bmatrix} A_i & A_{i'} \end{bmatrix} \begin{bmatrix} x_i \\ x_{i'} \end{bmatrix} = b$. for $x \geq 0$. (Hint: there are 7 pairs. Drawing b with the columns of A may help.)

Columns A_1 or A_5 must be included cause they are the only columns to the right of b. If A_5 is chosen A_6 cannot be chose since it is linearly dependent on A_5 and b needs to be in the span of the two columsn. So A_1 and any other column besides A_5 works. (4 pairs) and A_5 and any other column besides A_1 and A_6 works (3 pairs).

Total: 7 pairs.

(PTS:0-4) Now consider the reward vector $r^T = [-3\ -1\ -1\ 1\ -3\ 3\ 3]$ for $b^T = [2\ 2]^T$. Write the tableau for the linear program to maximize $r^T x$. Perform the pivot steps shown in the following illustrations.

What is the optimal x and $r^T x$?

In [242]:
```
## Tableau
r = np.array([-3.,-1.,-1.,1.,-3.,3.,3.])

A = np.array([[1.,0.,2.,1., 1.,-1.,-1.],
              [0.,1.,1.,2.,-1., 1., 2.]])
b = np.array([[2.,2.]]).T


T = np.block([[1.,-r,0.],
              [np.zeros((2,1)),A,b]])
print('Tableau: ')
print(T)
```

```
Tableau:
[[ 1.  3.  1.  1. -1.  3. -3. -3.  0.]
 [ 0.  1.  0.  2.  1.  1. -1. -1.  2.]
 [ 0.  0.  1.  1.  2. -1.  1.  2.  2.]]
```

In [243]:

```
T = np.block([[1.,-r,0.],
              [np.zeros((2,1)),A,b]])
print('Initial basis columns: x1,x2')
print('Cashing out...')
T[0] = T[0] - 3*T[1];
T[0] = T[0] -   T[2];
print('Updated Tableau:')
print(T)
print('Initial basis columns: x1,x2')
print('Initial reward: ',T[0,-1])

print('')
print('Swapping in x4 (-6 in objective row leads to reward increase)...
print('choose row 2... (swapping out x2)...')
T[2] = 0.5*T[2];
T[1] = T[1] - T[2];
T[0] = T[0] + 6*T[2];
print('Updated Tableau:')
print(T)
print('Updated basis columns: x1,x4')
print('Updated reward: ',T[0,-1])


print('')
print('Swapping in x3 (-3 in objective row leads to reward increase)...
print('choose row 1... (swapping out x1)...')
T[1] = (1./1.5)*T[1];
T[2] = T[2] - 0.5*T[1];
T[0] = T[0] + 3*T[1];
print('Updated Tableau:')
print(np.round(T,2))
print('Updated basis columns: x3,x4')
print('Updated reward: ',T[0,-1])

print('')
print('Swapping in x6 (-1 in objective row leads to reward increase)...
print('choose row 2... (swapping out x4)...')
T[1] = T[1] + T[2];
T[0] = T[0] + T[2];
print('Updated Tableau:')
print(np.round(T,2))
print('Updated basis columns: x3,x6')
print('Updated reward: ',np.round(T[0,-1],2))
print('')
print('No negative values in objective row. Done.')
print('Final basis columns: x3,x6')
print('Optimal reward (r^Tx): ',np.round(T[0,-1],2))
print('x3: ',np.round(T[1,-1],2))
print('x6: ',np.round(T[2,-1],2))
```

```
Initial basis columns: x1,x2
Cashing out...
Updated Tableau:
[[ 1.  0.  0. -6. -6.  1. -1. -2. -8.]
 [ 0.  1.  0.  2.  1.  1. -1. -1.  2.]
```

```
 [ 0.  0.  1.  1.  2. -1.  1.  2.  2.]]
Initial basis columns: x1,x2
Initial reward:  -8.0

Swapping in x4 (-6 in objective row leads to reward increase)...
choose row 2... (swapping out x2)...
Updated Tableau:
[[ 1.   0.   3.  -3.   0.  -2.   2.   4.  -2. ]
 [ 0.   1.  -0.5  1.5  0.   1.5 -1.5 -2.   1. ]
 [ 0.   0.   0.5  0.5  1.  -0.5  0.5  1.   1. ]]
Updated basis columns: x1,x4
Updated reward:  -2.0

Swapping in x3 (-3 in objective row leads to reward increase)...
choose row 1... (swapping out x1)...
Updated Tableau:
[[ 1.    2.    2.    0.    0.    1.   -1.    0.    0.  ]
 [ 0.    0.67 -0.33  1.    0.    1.   -1.   -1.33  0.67]
 [ 0.   -0.33  0.67  0.    1.   -1.    1.    1.67  0.67]]
Updated basis columns: x3,x4
Updated reward:  0.0

Swapping in x6 (-1 in objective row leads to reward increase)...
choose row 2... (swapping out x4)...
Updated Tableau:
[[ 1.    1.67  2.67  0.    1.    0.    0.    1.67  0.67]
 [ 0.    0.33  0.33  1.    1.    0.    0.    0.33  1.33]
 [ 0.   -0.33  0.67  0.    1.   -1.    1.    1.67  0.67]]
Updated basis columns: x3,x6
Updated reward:  0.67

No negative values in objective row. Done.
Final basis columns: x3,x6
Optimal reward (r^Tx):  0.67
x3:  1.33
x6:  0.67
```

(PTS:0-4) Now consider the reward vector $r^T = [-3\ 0\ 1\ 2\ 1\ -1\ 2]$ for $b^T = [2\ 0]^T$ Write the tableau for the linear program to maximize $r^T x$. Perform the pivot steps shown in the following illustrations. (see pdf)

What is the optimal x and $r^T x$?

In [244]:

```
## Tableau
r = np.array([-3.,0.,1.,2.,1.,-1.,2.])

A = np.array([[1.,0.,2.,1., 1.,-1.,-1.],
              [0.,1.,1.,2.,-1., 1., 2.]])
b = np.array([[2.,0.]]).T


T = np.block([[1.,-r,0.],
              [np.zeros((2,1)),A,b]])
print('Tableau: ')
print(T)
```

```
Tableau:
[[ 1.  3. -0. -1. -2. -1.  1. -2.  0.]
 [ 0.  1.  0.  2.  1.  1. -1. -1.  2.]
 [ 0.  0.  1.  1.  2. -1.  1.  2.  0.]]
```

In [245]:

```
T = np.block([[1.,-r,0.],
              [np.zeros((2,1)),A,b]])
print('START')
print('Initial basis columns: x1,x2')
print('Cashing out...')
T[0] = T[0] - 3*T[1];
print('Updated Tableau:')
print(T)
print('Initial basis columns: x1,x2')
print('Initial reward: ',T[0,-1])

print('')
print('STEP 1')
print('Swapping in x3 (-7 in objective row leads to reward increase)...
print('choose row 2... (swapping out x2)...')
T[1] = T[1] - 2*T[2];
T[0] = T[0] + 7*T[2];
print('Updated Tableau:')
print(T)
print('Updated basis columns: x1,x3')
print('Updated reward: ',T[0,-1])

print('')
print('STEP 2')
print('Swapping in x5 (-11 in objective row leads to reward increase)..
print('choose row 1... (swapping out x1)...')
T[1] = (1./3)*T[1];
T[2] = T[2] + T[1];
T[0] = T[0] + 11*T[1];
print('Updated Tableau:')
print(np.round(T,2))
print('Updated basis columns: x3,x5')
print('Updated reward: ',np.round(T[0,-1],2))

print('')
print('STEP 3')
print('Swapping in x7 (-3.33 in objective row leads to reward increase)
print('choose row 2... (swapping out x3)...')
T[2] = 3.*T[2];
T[1] = T[1] + (5/3)*T[2];
T[0] = T[0] + (10/3)*T[2];
print('Updated Tableau:')
print(np.round(T,2))
print('Updated basis columns: x5,x7')
print('Updated reward: ',T[0,-1])

print('')
print('')
print('No negative values in objective row. Done.')
print('Final basis columns: x5,x7')
print('Optimal reward (r^Tx): ',np.round(T[0,-1],2))
print('x5: ',np.round(T[1,-1],2))
print('x7: ',np.round(T[2,-1],2))
```

```
START
```

```
Initial basis columns: x1,x2
Cashing out...
Updated Tableau:
[[ 1.  0. -0. -7. -5. -4.  4.  1. -6.]
 [ 0.  1.  0.  2.  1.  1. -1. -1.  2.]
 [ 0.  0.  1.  1.  2. -1.  1.  2.  0.]]
Initial basis columns: x1,x2
Initial reward:  -6.0

STEP 1
Swapping in x3 (-7 in objective row leads to reward increase)...
choose row 2... (swapping out x2)...
Updated Tableau:
[[  1.   0.   7.   0.   9. -11.  11.  15.  -6.]
 [  0.   1.  -2.   0.  -3.   3.  -3.  -5.   2.]
 [  0.   0.   1.   1.   2.  -1.   1.   2.   0.]]
Updated basis columns: x1,x3
Updated reward:  -6.0

STEP 2
Swapping in x5 (-11 in objective row leads to reward increase)...
choose row 1... (swapping out x1)...
Updated Tableau:
[[ 1.    3.67 -0.33  0.   -2.    0.    0.   -3.33  1.33]
 [ 0.    0.33 -0.67  0.   -1.    1.   -1.   -1.67  0.67]
 [ 0.    0.33  0.33  1.    1.    0.    0.    0.33  0.67]]
Updated basis columns: x3,x5
Updated reward:  1.33

STEP 3
Swapping in x7 (-3.33 in objective row leads to reward increase)...
choose row 2... (swapping out x3)...
Updated Tableau:
[[ 1.  7.  3. 10.  8.  0.  0.  0.  8.]
 [ 0.  2.  1.  5.  4.  1. -1.  0.  4.]
 [ 0.  1.  1.  3.  3.  0.  0.  1.  2.]]
Updated basis columns: x5,x7
Updated reward:  8.0


No negative values in objective row. Done.
Final basis columns: x5,x7
Optimal reward (r^Tx):  8.0
x5:  4.0
x7:  2.0
```

(PTS:0-2) Which individual x_i's could correspond to the positive and negative part of a single unconstrained variable?

Columns A_5 and A_6 could represent the positive and negative part of a single unconstrained variable since they point in exactly opposite directions (and have rewards that are negatives of each other.)

In [ ]:

In [ ]:

```
1
```