First we set the page width for jupyter notebook.

In [1]:
```
from IPython.core.display import display, HTML
display(HTML("<style>.container { width:100% !important; }</style>"))
```

Now we import matplotlib, numpy, interactive and widgets. Additionally, we set the backend of matplotlib to the inline setting.

In [2]:
```
import matplotlib
import matplotlib.pyplot as plt
import numpy as np
from ipywidgets import interactive, widgets
%matplotlib inline
```

Now, we establish a function func(x) which we will name fun to be called in our other functions. This can be changed to study other objective functions.

In [3]:
```
def func(xi):
    y= -(2*xi[0]+3*xi[1]-xi[0]**3-2*xi[1]**2);
    #y=0.5*xi[0]**2+xi[1]**2-xi[0]*xi[1]-7*xi[0]-7*xi[1];
    return y
```

Now, we establish a function to return the value of the function fun(x) along with it's derivative dy at the input point xi. The derivative for this case is defined analytically. Issues arose when using the finite difference methodology.

In [4]:
```
def fun(xi):
    from numpy import arange,append,transpose,asarray
    #y= -(2*xi[0]+3*xi[1]-xi[0]**3-2*xi[1]**2);
    y=func(xi)
    #dy=[]
    #dx=0.000000001
    #k=arange(1,len(xi)+1,1)
    ##print(k)
    #for i in k:
    #    xtemp=xi[:]
    #    xtemp[i-1]=xtemp[i-1]+dx
    #    ytemp=func(xtemp)
    #    ydif=ytemp-y
    #    a = (ydif)/dx
    #    dy=append(dy,a)
    dy=asarray([-2+3*xi[0]**2,-3+4*xi[1]])
    #dy= asarray([xi[0]-xi[1]-7,2*xi[1]-xi[0]-7])
    return y,dy
```

Next, we will establish a function func_a which will define the search function along its path.

In [5]:
```
def func_a(fun,dk,xk,ai):
    from numpy import asarray
    xi = xk + ai*asarray(dk)
    fi= fun(xi)
    return fi
```

Now, we will establish a function to search for the upper and lower bounds of our search. If this domain contains the function minimum then the while loop will break.

In [6]:
```
def bound_search(fun,xk,dk,LB,UB,delta):
    from numpy import arange
    from numpy import asarray
    as1= arange(LB,UB,delta)
    q=1
    aq=as1[q]
    xq=xk+asarray(dk)*aq
    yq1=func_a(fun,dk,xk,aq)
    while 1:
        q=q+1;
        aq=as1[q]
        yq2 = func_a(fun,dk,xk,aq)
        if yq1<yq2:
            new_LB=as1[q-2]
            new_UB=as1[q]
            break

        if q==len(as1)-1:
            new_LB=as1[q-2]
            new_UB=as1[q]
            break
        yq1=yq2
    return (new_LB,new_UB)
```

Now, a function lscov to replace functionality of lscov in matlab is established

In [7]:
```
def lscov(A, B, w=None):

    if w is None:
        Aw = A.copy()
        Bw = B.T.copy()
    else:
        W = np.sqrt(np.diag(np.array(w).flatten()))
        Aw = np.dot(W, A)
        Bw = np.dot(B.T, W)

    x, residuals, rank, s = np.linalg.lstsq(Aw, Bw.T, rcond=1e-10)
    return np.array(x).flatten()
```

Now, we will establish a function to impliment the line search method. The bounds are searched iteratively until they are within a tolerance epsilon value

In [8]:
```
#from matplotlib import *
def line_search_equal_interval(fun, xk, dk):

    from numpy import arange
    from numpy import append
    #from matplotlib.pyplot import subplots,plot,show,xlabel,ylabel,legend,tit
le,tick_params,grid,ylim,xlim,xticks,yticks
    #%matplotlib inline
    UB=10
    LB=0

    delta = .05
    r = .2
    epsilon_tolerance2 = .00001
    aopt_found = 0
    iter1 = 1


    while not (aopt_found == 1):
        (new_LB, new_UB) = bound_search(fun, xk, dk, LB, UB, delta)
        if abs(new_UB - new_LB) < epsilon_tolerance2:
            aopt_found = 1;
        delta = r * delta
        LB = new_LB
        UB = new_UB

        iter1 = iter1 + 1
        a_opt = (new_UB + new_LB) / 2


    return a_opt
```

Now, we will establish a function to impliment the golden line search and return its optimum search direction

In [9]:
```
def golden_search(fun,xk,dk):
    import math
    from numpy import arange,append,asarray,sum

    LB=0
    UB=10
    delta=.01
    etol =.0001
    r=(1+math.sqrt(5))/2
    ir=1/r
    at=arange(LB,UB,delta)
    xas=[]
    yts=[]
    sn = len(at)
    sn = arange(0, sn, 1)
    iterations=0

    for i in sn:
        fi = func_a(fun,dk,xk,at[i])
        yts = asarray(append(yts,fi))
    ymin=min(yts)
    ymax=max(yts)

    a0=delta
    f0=func_a(fun,dk,xk,a0)
    a1=delta+delta*r
    f1=func_a(fun,dk,xk,a1)
    id1=2


    while 1:
        a2 = delta*sum(r**asarray(arange(0,id1,1)))
        f2 = func_a(fun,dk,xk,a2)
        if (f0>f1) & (f1<f2):
            break
        else:
            id1=id1+1
            a0=a1
            a1=a2
            f0=f1
            f1=f2

    ##phase 2


    aL=a0
    aA=a1
    aU=a2
    fL=f0
    fA=f1
    fU=f2


    Intv0=aU-aL
    aB=aL+ir*Intv0
    fB=func_a(fun,dk,xk,aB)
```

```
    while 1:


        if (fA<fB):
            aL=aL
            aU=aB
            aB=aA
            fL=fL
            fU=fB
            fB=fA
            Intv1=aU-aL
            aA=aL+(1-ir)*Intv1
            fA=func_a(fun,dk,xk,aA)
        else:
            aL=aA
            aU=aU
            aA=aB
            fL=fA
            fU=fU
            fA=fB
            Intv1=aU-aL
            aB=aL+ir*Intv1
            fB=func_a(fun,dk,xk,aB)
        if (abs(Intv1-Intv0) < etol):
            break
        else:
            Intv0=Intv1
        iterations=iterations+1
    a_opt = (aU+aL)/2
    f_opt = (fU+fL)/2
    return a_opt
```

Now, we establish a steepest descent function and a ex10_25_fx_graphic() function:

In [10]:
```
def ex10_25_fx_graphic():
    from numpy import arange,meshgrid

    xs=arange(0,1,.01)
    [x1,x2]=meshgrid(xs,xs)
    f=func([x1,x2])
    fig,ax=plt.subplots()
    CS=ax.contour(x1,x2,f,1000,cmap='Spectral')
    plt.colorbar(CS)
    plt.xlabel('x1',fontsize=24)
    plt.ylabel('x2',fontsize=24)
    fig.set_size_inches(18.5, 10.5)
```

We now establish the function to return the optimal search direction for the armijio method.

In [11]:
```
def line_search_armijo(fun,xk,dk):
    from numpy import linspace,asarray,append,arange
    LB=0
    UB=1

    as1=linspace(0,1,30)
    sn =arange(0,len(as1),1)
    yts=[]

    for i in sn:
        fi = func_a(fun,dk,xk,as1[i])
        yts = asarray(append(yts,fi))

    nita=2
    rho=.8
    ai=.01
    forwardcase=1
    k=0
    da=0.00001
    fo=func_a(fun,dk,xk,0.0)
    f0_da=func_a(fun,dk,xk,da)
    dfo=(f0_da-fo)/da
    fa=func_a(fun,dk,xk,ai)
    qa=fo+rho*dfo*ai


    if (fa>qa):
        nita=1/nita
        ai=nita*ai
        forwardcase=0
    k=1

    while 1:

        fa=func_a(fun,dk,xk,ai)
        #print("fa:"+str(fa))
        qa=fo+ rho*dfo*ai

        if (fa>qa) & forwardcase:
            break
        if (fa<qa) & ~forwardcase:
            break

        ai=nita*ai
        k=k+1
    a_opt=ai
    return a_opt
```

We now establish a function to perform the polynomial line search.

In [12]:
```
def polynomial_line_search(fun,xk,dk):
    from numpy import linspace,arange,append,transpose
    aL=0
    aU=2
    aI=(aL+aU)/2
    etol=0.00001
    at=linspace(0,3,30)
    yts=[]
    k=0

    sn = arange(0, len(at), 1)


    for i in sn:
        fi = func_a(fun,dk,xk,at[i])
        yts = append(yts,fi)
        #print("yts:"+str(yts))
    fU=func_a(fun,dk,xk,aU)
    fL=func_a(fun,dk,xk,aL)
    fI=func_a(fun,dk,xk,aI)
    a_bar_old=aL
    #print("aold:"+str(a_bar_old))

    while 1:
        k=k+1
        #print("Loop "+str(k))
        X=[[1, aL, aL**2],[1,aU,aU**2],[1,aI,aI**2]]
        Y=append(append(fL,fU),fI)
        #print("Y: "+str(Y))
        a=lscov(X,Y)
        #print("a:"+str(a))
        a0=a[0]
        a1=a[1]
        a2=a[2]
        #print("a0: "+str(a0)+'\na1: '+str(a1)+'\na2: '+str(a2))
        a_bar_new=-a1/(2*a2)
        #print("a_bar_new:"+str(a_bar_new))
        f_abar=func_a(fun,dk,xk,a_bar_new)
        #print("f_abar:"+str(f_abar))


        if aI<a_bar_new:

            if fI<f_abar:

                aU=a_bar_new
                fU=f_abar
            else:
                aL=aI
                fL=fI
                aI=a_bar_new
                fI=f_abar

        else:
```

```
            if fI<f_abar:
                aL=a_bar_new
                fL=f_abar
            else:
                aU=aI
                fU=fI
                aI=a_bar_new
                fI=f_abar
        a_opt = a_bar_new
        check=abs(a_bar_new-a_bar_old)
        #print("check:"+str(check))
        if (check<etol):
            break


        a_bar_old = a_bar_new
        #print("a_bar_old:"+str(a_bar_old))


    return a_opt
```

We now establish a function to define the steepest descent method and perform it using equal interval, golden section, armijo, and polynomial line search methodologies

In [43]:
```
def steepest_descent(method):
    import time
    start=time.perf_counter()
    from numpy import linalg,transpose,arange,meshgrid,append,asarray
    xka=[]
    xkx=[]
    xky=[]
    x0=[.2,0]
    k=0
    epsilon_tol1=.0001
    xk=x0[:]

    xka=append(xka,xk[:])
    iter=0
    while 1:


        iter=iter+1;
        #print("xk_"+str(k)+': '+str(xk))
        y,dy=fun(xk)
        ck=dy
        #print('dy:'+str(dy))




        if (linalg.norm(ck)<epsilon_tol1):
            stop=time.perf_counter()
            print('break occured after '+str(iter)+" iterations",end=", ")
            print("Final xk:"+str(xk))
            if method==1:
                print('line search section method used')
            elif method==2:
                print('golden section search method used')
            elif method==3:
                print('armijo section search method used')
            elif method==4:
                print('polynomial line search method used')
            break

        dk=-ck


        if method==1:
            a_opt=line_search_equal_interval(func, xk, dk)
        elif method==2:
            a_opt=golden_search(func,xk,dk)
        elif method==3:
            a_opt=line_search_armijo(func,xk,dk)
        elif method==4:
            a_opt=polynomial_line_search(func,xk,dk)
        else:
            print("not a proper input")

        #print("a_opt: "+str(a_opt))
        xk=xk+dk*a_opt
```

```
        xka=append(xka,xk[:])


        k=k+1

    #print("xka:"+str(xka))

    for i in arange(0,len(xka),2):
        xkx=append(xkx,xka[i])
    #print("xkx:"+str(xkx))
    for i in arange(1,len(xka),2):
        xky=append(xky,xka[i])
    #print("xky:"+str(xky))

    elapsed=stop-start
    print(str(elapsed)+" seconds calculation time")
    ex10_25_fx_graphic()
    #CS= plt.scatter(xka[0],xka[1])
    CS= plt.scatter(xkx,xky,cmap='binary',c=xkx)
    CS= plt.plot(xkx,xky,'b--',linewidth=2)
    if method==1:
        plt.title("equal interval line search with steepest descent method",fo
ntsize=24)
    elif method==2:
        plt.title("golden section line search method with steepest descent met
hod",fontsize=24)
    elif method==3:
        plt.title("armijo line section search method with steepest descent met
hod",fontsize=24)
    elif method==4:
        plt.title("polynomial interpolation search method with steepest descen
t method",fontsize=24)
    plt.show()
```

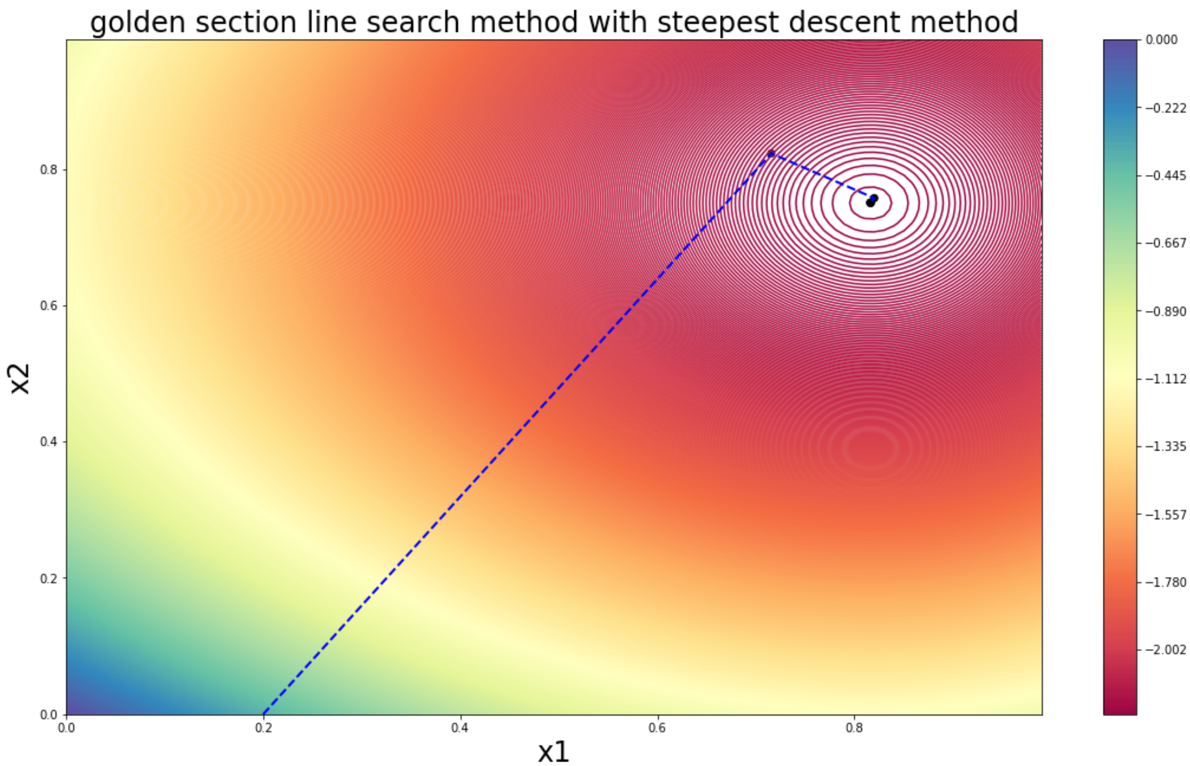
Finally, we create a widget which allows the usage of the steepest descent method and the methodologies: equal interval line search, golden section line search, armijo line search, and polynomial line search. This widget will take inputs 1,2,3,4 for each of the corresponding aforementioned methodologies.

In [44]:
```
#interactive(steepest_descent,method=[1,2,3,4])
```

In [50]:
```
#steepest_descent(1)
```

In [46]:
```
steepest_descent(2)
```

```
break occured after 6 iterations, Final xk:[0.81649195 0.75000352]
golden section search method used
0.06526189999999588 seconds calculation time
```




In [51]:
```
#steepest_descent(3)
```

In [52]:
```
#steepest_descent(4)
```

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]: