Chapter Problems

Problem 4

Given data arrays of $\Delta h_o(kJ/mol)$ and $\Delta s_o(J/mol*K)$ versus δ for Ceria, fit a third degree poynomical function suitable for describing Δh_o and Δs_o as a function of δ using the _curve*fit* function from the *scipy.optimize* library.

NOTE: For this we will use Python and import tabulated data from a *.csv* file.

In [6]:
```
#Import necessary libraries and data arrays from files.
import numpy as np
import pandas as pd
data1 = pd.read_csv('Change_in_EnthalpyVSDelta.csv')
data2 = pd.read_csv('Change_in_EntropyVSDelta.csv')
delta_h = np.array(data1) #Numpy array [[delta1,delta_h1],[delta2,delta_h2],...]
delta_s = np.array(data2) #Numpy array [[delta1,delta_s1],[delta2,delta_s2],...]
```

> Plot extracted data by slicing the data array in different columns. The first column represents δ and the second column represents either Δh_o or Δs_o

In [7]:
```
#Import necessary libraries and plot raw data for better visualization
import matplotlib.pyplot as plt
plt.plot(delta_h[:,0], delta_h[:,1], 'bo', markersize=3, label='Data '+'$\Delta$'+'$h_o$')
plt.plot(delta_s[:,0], delta_s[:,1], 'g--', markersize=3, label='Data '+'$\Delta$'+'$s_o$')
plt.xlabel('$\delta$')
plt.ylabel('$\Delta$'+'$h_o$'+'$(kJ/mol)$'+' and '+'$\Delta$'+'$s_o$'+'$(J/molK)$')
plt.legend(loc='center right')
plt.show()
```

```
<Figure size 640x480 with 1 Axes>
```

> Define a polynomial function *poly* dependent on delta (**d**) and the different coefficients of the polynomial (**c1,c2,c3,c4**)

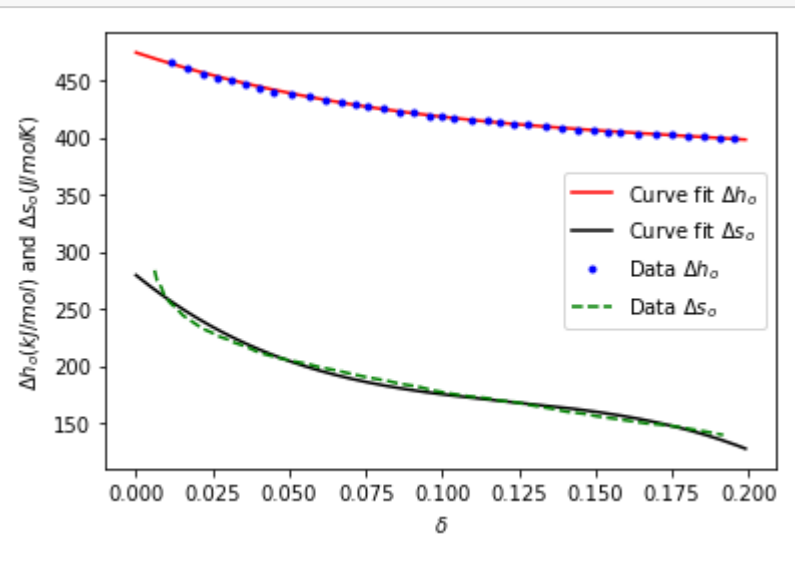
In [8]:
```
def poly(d,c1,c2,c3,c4):
     return c1*d**3+c2*d**2+c3*d+c4
```

In [9]:
```
#Import necessary libraries and calculate the polynomial coefficients
from scipy.optimize import curve_fit
coeff1, pcov = curve_fit(poly, delta_h[:,0], delta_h[:,1])
coeff2, pcov = curve_fit(poly, delta_s[:,0], delta_s[:,1])
print('Coefficients for delta_h : ',coeff1)
print('Coefficients for delta_s : ',coeff2)
```

```
Coefficients for delta_h :  [-7385.80014712   4017.51492446  -889.65156973   474.07082102]
Coefficients for delta_s :  [-42659.14948158  15579.16742436  -2173.31571416    279.33859835]
```

> Plot curve fit polynomial function *poly* with the obtained coefficients along with the previously plotted raw data to show comparison

In [10]:
```
d_fit = np.arange(0,0.2,0.001) #Define new array for delta
plt.plot(d_fit, poly(d_fit, *coeff1), 'r', label='Curve fit '+'$\Delta$'+'$h_o$')
plt.plot(d_fit, poly(d_fit, *coeff2), 'k', label='Curve fit '+'$\Delta$'+'$s_o$')
plt.plot(delta_h[:,0], delta_h[:,1], 'bo', markersize=3, label='Data '+'$\Delta$'+'$h_o$')
plt.plot(delta_s[:,0], delta_s[:,1], 'g--', markersize=3, label='Data '+'$\Delta$'+'$s_o$')
plt.xlabel('$\delta$')
plt.ylabel('$\Delta$'+'$h_o$'+'$(kJ/mol)$'+' and '+'$\Delta$'+'$s_o$'+'$(J/molK)$')
plt.legend(loc='center right')
plt.show()
```



> **Solution:** Δh_o and Δs_o can now be computed as a function of δ and the curve fit coefficients found, *coeff1* and *coeff2* respectively

Problem 5

Given a temperature of operation $T = 1273K$ for Ceria and an initial amount of moles of water $n_{H_2O,i} = \delta_f = 0.1$. Solve for δ using an iterative approach.

Background: The partial molar Gibbs free energy for Ceria Δg_o, which is a function of both nonstoichiometry and temperature, is expressed as two equations below:

$$\Delta g_o(\delta,T) = -RT\ln p_{O2}(\delta,T)^{1/2} \qquad (1.50)$$
$$\Delta g_o(\delta,T) = \Delta h_o(\delta) - T\Delta s_o(\delta) \qquad (1.51)$$

Also, the oxygen partial pressure p_{O2} can be obtained from the reaction equilibrium analysis of the dissociation of H_2O. From (1.33) we recognize that the reaction coordinate ϵ is equal to $\delta_f - \delta$,

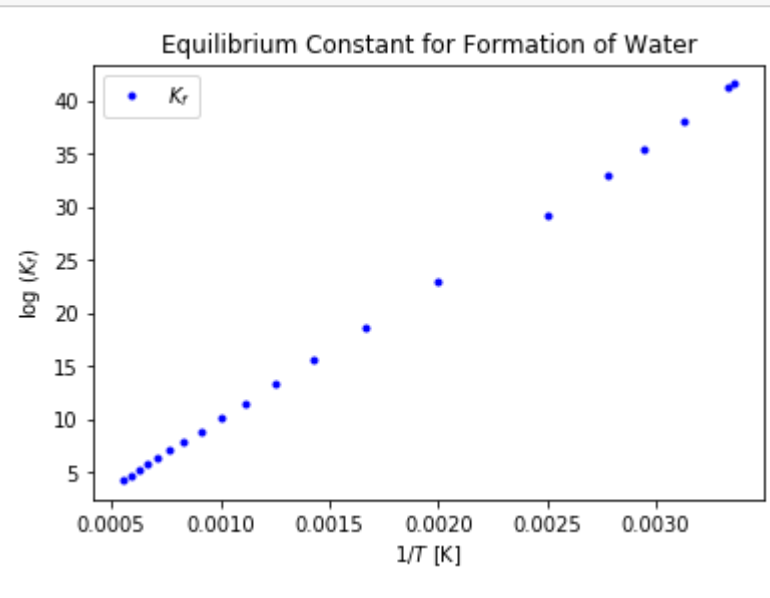
$$K_{H_2O} = \frac{(\frac{\epsilon}{n_{total}}) * p_{O2}{}^{1/2}}{(\frac{n_{H_2O,i}-\epsilon}{n_{total}})} = \frac{(\delta_f - \delta) * p_{O2}{}^{1/2}}{n_{H_2O,i} - (\delta_f - \delta)}$$
$$p_{O2}{}^{1/2} = \frac{n_{H_2O,i} - (\delta_f - \delta)}{(\delta_f - \delta) * K_{f,H_2O}}$$

NOTE: For this we will use Python, and import data for the equilibrium constant of formation of water K_{f,H_2O} from the NIST-JANAF website. A curve fitting procedure needs to be applied, similar to Problem 4.

In [11]:
```
#Import Equilibrium Constant Data for formation of Water, 1bar, (l,g)
data3 = pd.read_csv('H2O_Equilibrium_Constant.csv')
Kf_H2O = np.array(data3)
```

> Plot extracted data by slicing the data array in different columns. The first column represents $T(K)$ while the second column represents $log(K_{f,H_2O})$. The data has a linear relationship when plotted with $\frac{1}{T}$ in the x-axis.

In [12]:
```
#Plot raw data fro better visualization
plt.plot(1/(Kf_H2O[:,0]), Kf_H2O[:,1], 'bo', markersize=3, linewidth=0, label='$K_f$')
plt.xlabel('$1/T$'+' [K]')
plt.ylabel('log '+'$(K_f)$')
plt.title('Equilibrium Constant for Formation of Water')
plt.legend()
plt.show()
```



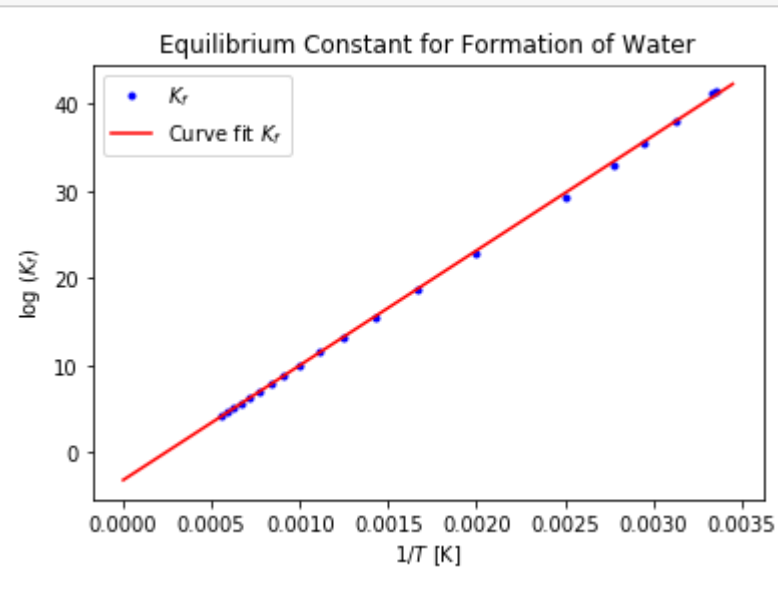
> Define a function named *linear* dependent on the **(x)**, the slope **(m)** and the y-intercept **(b)**. Calculate the *linear* function coefficients. Then, plot the curve fit function *linear* with the obtained coefficients along with the previously plotted raw data to show comparison.

In [13]:
```
def linear(x, m, b):
     return m*x+b

#Calculate the linear function coefficients in the same way as Problem 4
coeff3, pcov = curve_fit(linear, 1/(Kf_H2O[:,0]), Kf_H2O[:,1])
print('Coefficients for Kf : ', coeff3)
```

```
Coefficients for Kf :  [ 1.31648379e+04 -3.13560258e+00]
```

In [14]:
```
xfit = np.arange(0,0.0035,0.00005) #Define new array for x
plt.plot(1/(Kf_H2O[:,0]), Kf_H2O[:,1], 'bo', markersize=3, linewidth=0, label='$K_f$')
plt.plot(xfit, linear(xfit, *coeff3), color='red', label='Curve fit '+'$K_f$')
plt.xlabel('$1/T$'+' [K]')
plt.ylabel('log '+'$(K_f)$')
plt.title('Equilibrium Constant for Formation of Water')
plt.legend()
plt.show()
```



> Define functions for the partial molar Gibbs free energy as a function of both nonstoichiometry and temperature, make sure to replace the expression for oxygen partial pressure into one of the functions. Delta(nonstoichiometry) is represented as d.

In [15]:
```
#Define variables and fixed parameters
R = 0.008314 #(kJ/mol K)
T = 1273 #(K)
df = 0.1 #(moles)
n_H2O = df #(moles)
```

In [16]:
```
def Gibbs1(d):
    return -R*T*np.log((n_H2O-(df-d))/((df-d)*10**(linear((1/T), *coeff3))))
def Gibbs2(d):
    return poly(d, *coeff1)-poly(d, *coeff2)*T/1000
```

> Trial and error approach. Iteratively input values for delta such that $0.0 < \delta < \delta_f$. Modify values in both functions until the result is approximately the same. If we substract both functions the value should get close to zero when the function values are the same.

In [17]:
```
#Try d=0.08
attempt = Gibbs1(0.08)-Gibbs2(0.08)
print(attempt)
```

```
-30.502246290666164
```

In [18]:
```
#Try d=0.09
attempt = Gibbs1(0.09)-Gibbs2(0.09)
print(attempt)
```

```
-41.151086232985364
```

In [19]:
```
#Try d=0.06
attempt = Gibbs1(0.06)-Gibbs2(0.06)
print(attempt)
```

```
-12.975066084019687
```

In [20]:
```
#Try d=0.05
attempt = Gibbs1(0.05)-Gibbs2(0.05)
print(attempt)
```

```
-3.0405284425706327
```

> **Solution:** $\delta = 0.05$ approximately

Problem 6

Given a temperature of operation $T = 1273K$ for Ceria and an initial amount of moles of water $n_{H_2O,i} = \delta_f = 0.1$. Solve for δ and the H_2 yield. Create a solver/minimizer using the *minimize* function from the *scipy.optimize* library.

NOTE: For this we will use Python, and remember that $H_2 = \delta_f - \delta$, which is the difference between the final and initial nonstoichiometries.

> Define the function *objective* , which is the absolute value of the difference between both partial molar Gibbs free energy expressions, in that way we guarantee the result gets as close possible to zero. The function *objective* is only dependent on delta **(d)**, an array with multiple elements. The first element **d[0]** is the one we will be solving for, while the rest belong to the different coefficients we previously found.

In [21]:
```
#Import necessary libraries for minimization procedure
from scipy.optimize import minimize
def objective(d):
    return np.abs(((d[1]*d[0]**3+d[2]*d[0]**2+d[3]*d[0]+d[4])-(d[5]*d[0]**3+d[6]*d[0]**2+d[7]*d[0]+d[8
])*T/1000)\
                  +(R*T*np.log((n_H2O-(df-d[0]))/((df-d[0])*10**(linear((1/T), *coeff3))))))
```

> When solving for **d[0]** we do not want to vary any of the other elements from delta **(d)**. Therefore, we are setting every other element to its respective coefficient value via constraints:

In [22]:
```
con1 = {'type': 'eq', 'fun': lambda d: d[1]-coeff1[0]}
con2 = {'type': 'eq', 'fun': lambda d: d[2]-coeff1[1]}
con3 = {'type': 'eq', 'fun': lambda d: d[3]-coeff1[2]}
con4 = {'type': 'eq', 'fun': lambda d: d[4]-coeff1[3]}
con5 = {'type': 'eq', 'fun': lambda d: d[5]-coeff2[0]}
con6 = {'type': 'eq', 'fun': lambda d: d[6]-coeff2[1]}
con7 = {'type': 'eq', 'fun': lambda d: d[7]-coeff2[2]}
con8 = {'type': 'eq', 'fun': lambda d: d[8]-coeff2[3]}
constraints_d = [con1, con2, con3, con4, con5, con6, con7, con8]
```

> The minimization method we are using, **(SLSQP)**, requires a set of initial guesses and boundaries for every value. Define those parameters, then minimize the *objective* function.

In [23]:
```
initial_guess = np.array([0.07, 1, 1, 1, 1, 1, 1, 1, 1])
b = (-50000, 50000)
bounds_d = ((0.0,df-0.0001), b, b, b, b, b, b, b, b)
solution = minimize(objective, initial_guess, method='SLSQP', constraints=constraints_d, options={'dis
p': False}, bounds=bounds_d)
print('Value of Delta is: ', solution.x[0])
```

```
Value of Delta is:  0.0472257161777355
```

```
C:\Users\antma\Anaconda3\lib\site-packages\ipykernel_launcher.py:5: RuntimeWarning: divide by zero en
countered in log
  """
```

In [24]:
```
print("H_2 yield is: ",df-solution.x[0])
```

```
H_2 yield is:  0.052774283822264505
```

> **Solution:** $\delta = 0.0472$ and $H_2 = 0.0528$

Problem 7

Given 6 different temperatures of reduction $T_H = 1573, 1673, 1773, 1873, 1973, 2073K$, temperature of oxidation $800K < T_L < 1200K$, and oxygen partial pressure $p_{O2} = 0.00001$ for Ceria. Compute H_2 yield to reproduce data from Chueh et al. **Figure 17.a**

NOTE: For this we will use Python, and remember that $n_{H_2O,i} = \delta_f$ (initial moles of water is equal to the nonstoichiometry after reduction).

> Perform minimization procedure to find the nonstoichiometry after reduction **(d_red)** knowing the relationship in between p_{O2}, T, and δ given by (1.52):
>
> $$\ln p_{O2}(\delta,T)^{1/2} = -\frac{1}{T}\frac{\Delta h_o(\delta)}{R} + \frac{\Delta s_o(\delta)}{R} \qquad (1.52)$$
>
> Define function _objective*red* depending only on delta **(d)** similar to Problem 6. However, the minimization procedure will repeat using a *for loop* for every temperature of reduction **(T_red)**. Then, the values of delta **(d)** will be stored in the array **(d_red)** for later use.

In [25]:
```
#Define variables and fixed parameters
T_red = np.array([2073, 1973, 1873, 1773, 1673, 1573]) #(K)
O_pp = 1e-5
d_red = np.array([])
```

In [26]:
```
for i in range(0, T_red.size):
    def objective_red(d):
        return np.abs(np.log(O_pp**0.5)+((d[1]*d[0]**3+d[2]*d[0]**2+d[3]*d[0]+d[4])/(T_red[i]*R))\
              -((d[5]*d[0]**3+d[6]*d[0]**2+d[7]*d[0]+d[8])/(1000*R)))

    #Define constraints to their respective coefficients values
    con1 = {'type': 'eq', 'fun': lambda d: d[1]-coeff1[0]}
    con2 = {'type': 'eq', 'fun': lambda d: d[2]-coeff1[1]}
    con3 = {'type': 'eq', 'fun': lambda d: d[3]-coeff1[2]}
    con4 = {'type': 'eq', 'fun': lambda d: d[4]-coeff1[3]}
    con5 = {'type': 'eq', 'fun': lambda d: d[5]-coeff2[0]}
    con6 = {'type': 'eq', 'fun': lambda d: d[6]-coeff2[1]}
    con7 = {'type': 'eq', 'fun': lambda d: d[7]-coeff2[2]}
    con8 = {'type': 'eq', 'fun': lambda d: d[8]-coeff2[3]}
    constraints_red = [con1, con2, con3, con4, con5, con6, con7, con8]

    #Define initial guess and bounds
    ini_guess_red = np.array([0.03, 1, 1, 1, 1, 1, 1, 1, 1])
    b = (-50000, 50000)
    bounds_red = ((0.0001,0.2), b, b, b, b, b, b, b, b)

    #Minimize function objective_red
    sol_red = minimize(objective_red, ini_guess_red, method='SLSQP', constraints=constraints_red, optio
ns={'disp': False}, bounds=bounds_red)
    d_red = np.append(d_red, sol_red.x[0])
```

> Perform minimization procedure to find nonstoichiometry after oxidation **(d_ox)** similar to Problem 6. Define function _objective*ox* depending only on delta **(d)**. The minimization procedure will repeat using a *for loop* for different temperatures of oxidation in the range $800K < T_{ox} < 1200K$. Then the values of delta **(d)** will be stored in the **(H_2)** array, remembering that $H_2 = \delta_{red} - \delta_{ox}$.

In [27]:
```
#Define variables and fixed parameters
n_H2O = d_red*1
T_ox = np.arange(800,1200,10)
```

In [28]:
```
for ii in range(0, d_red.size-1):
    H_2 = np.array([])
    for i in range(0, T_ox.size):
        def objective_ox(d):
            return np.abs(((d[1]*d[0]**3+d[2]*d[0]**2+d[3]*d[0]+d[4])-(d[5]*d[0]**3+d[6]*d[0]**2+d[7]*d
[0]+d[8])*T_ox[i]/1000)\
                          +(R*T_ox[i]*np.log((n_H2O[ii]-(d_red[ii]-d[0]))/((d_red[ii]-d[0])*10**linear((1/T
_ox[i]), *coeff3)))))

        #Constraints will remain the same from reduction
        #Define new initial guesses and bounds
        ini_guess_ox = np.array([0.01, 1, 1, 1, 1, 1, 1, 1, 1])
        bounds_ox = ((0.0,d_red[ii]-0.0001), b, b, b, b, b, b, b, b)

        #Minimize function objective_ox
        sol_ox = minimize(objective_ox, ini_guess_ox, method='SLSQP', constraints=constraints_red, opti
ons={'disp': False}, bounds=bounds_ox)
        H_2 = np.append(H_2, d_red[ii]-sol_ox.x[0])

    #For every T_red plot H_2 productivity as a function of T_ox
    plt.plot(T_ox, H_2, 'r')
    plt.plot(T_ox, np.repeat(d_red[ii], 40), 'b--', linewidth=1)
    plt.xlabel('Oxidation Tempertature, '+' $T_L [K]$')
    plt.ylim(0, 0.2)
    plt.ylabel('$H_2$'+' Productivity')

#Plot image with labels and text to reproduce original image as accurately as possible
plt.text(1170, 0.182, '$\delta_i$')
plt.text(810, 0.182, '$T_H=2073 K$')
plt.text(810, 0.16, '$1973 K$')
plt.text(810, 0.105, '$1873 K$')
plt.text(810, 0.065, '$1773 K$')
plt.text(810, 0.042, '$1673 K$')
plt.text(810, 0.024, '$1573 K$')
plt.plot(T_ox, np.repeat(d_red[5], 40), 'b--', linewidth=1)
plt.show()
```

```
C:\Users\antma\Anaconda3\lib\site-packages\ipykernel_launcher.py:6: RuntimeWarning: divide by zero en
countered in log
```