```
from __future__ import division
from Database_Schomer import species,R_u
from Simclass import *
import numpy as np
from squareup import *
from scipy.optimize import root,    minimize,   bisect
from math import log as mathlog

'''
This is the prototype solver for the eilmer chemical equilibrium
module. This file contains the solver for both the stoichiometric
and LNR methods, each of which is called by the 'solve' method.

'Simclass.py' defines the species properties.

Hugh McDougall, 1/6 2018
'''

ZG_SWITCH=True      #A switch to apply the mu/RT+=1 switch of the Z&G formulation

class solver:
    def __init__(self,sim):
        '''
        A solver is created with a 'simulation' as a template
        to read its physical properties from. These can be changed
        at runtime. A simulation has set U, V and species abundances.
        '''

        self.debug=False

        self.U0 =sim.u          #Prescribed internal energy
        self.V  =sim.V          #Prescribed volume

        self.I=sim.N_spec       #The number of chemical species
        self.J=sim.N_atom       #The number of atomic species

        self.b      =np.zeros([sim.N_atom])     #Elemental abundance atom
        self.C      =np.zeros([self.J,self.I])  #Elemental conservation matrix

        #Seed conditions for solver
        self.T0 =298
        self.N0 =np.zeros([self.I])+(sum(self.b)/self.I)
        self.n0 =np.zeros(self.I-self.J)+(sum(self.b)/self.I)
        self.PI0=np.zeros([self.J])

        #Make Conservation matrices
        for i,entry in zip( range(self.I),  sim.speclist):
            for j,name in zip(range(self.J),sim.atomlist):
                for atomname in entry[0].atom.keys():
                    if atomname==name:
                        self.b[j]       +=entry[0].atom[atomname]*entry[1]
                        self.C[j,i]      =entry[0].atom[atomname]

        #Sort into primary and secondary species using squareup()
        """
        This is the separation of species step.
        From this point forward, all ordering of chemical species
        is of the form N=[n_primary,n_secondary]

        This step is only reversed in the output of solve()
        """
```

```
        self.A, self.D, self.cutcols=squareup(self.C)
        self.Ainv=np.linalg.inv(self.A)
        self.sorter,self.unsorter=assembly_vectors(self.I,self.cutcols)

        #split species list
        self.specs    =(np.array(sim.speclist)[:,0])[self.sorter]

        #Establish primary species boundaries
        self.nmax=self.getnmax()

    #Sys State Evaluation Functions
    #---------------------------------
    def getnmax(self):
        '''
        Makes a rough estimate of the 'square' region that
        the physical solution space lies within.

        Takes no arguments.
        Returns an array of primary species
        '''
        nmax=np.zeros([self.I-self.J])
        for l in range(self.I-self.J):
            for j in range(self.J):

                nmax[l]=min([(self.b[j]/self.D[j,l]) for j in range(self.J) if self.D[j,l]!=0])

        return(nmax)

    def resetseed(self):
        '''
        Resets the solver's initial estimate for the solvers
        by way of the averaged atom method.
        Takes no arguments.
        Acts directly on the solver variables.
        '''
        self.N0=np.zeros([self.I])+sum(self.b)/self.I
        self.T0=298
        self.PI0=np.zeros([self.J])
        self.n0=self.N0[self.sorter][:self.I-self.J] #Read the primary species from current seed


    #Sys State Evaluation Functions
    #---------------------------------
    def sys_internalenergy(self,N,T):
        '''
        Calculates the internal enegry for a set temperature
        and species count. For use in temp_solve

        Takes (sorted) complete species list and temperature.
        Returns internal energy.
        '''
        assert len(N)==len(self.specs), "Wrong count length in sys_internalenergy"

        outU=0
        for entry,n in zip(self.specs,N):
            outU+=entry.u(T)*n
        return(outU)

    def sys_enthalpy(self,N,T):
        '''
        Calculates the enthalpy for a set temperature and species count
```

```
    For use in getting gibbs

    Takes (sorted) complete species list and temperature.
    Returns total enthalpy.
    '''
    assert len(N)==len(self.specs), "Wrong count length in sys_enthalpy"

    outH=0
    for entry,n in zip(self.specs,N):
        outH+=entry.h(T)*n
    return(outH)

def sys_entropy(self,N,T):
    '''
    Calculates the entropy for a set temperature and species count
    For use in getting gibbs

    Takes (sorted) complete species list and temperature.
    Returns total entropy.
    '''
    assert(min(N))>0,"min(N)<0 in entropy eval"
    assert(T>0), "Negative temp in entropy eval"
    outS=0

    #Perform Summation
    for entry,n in zip(self.specs,N):
        if n>0:
            si  =   entry.s(T)
            si +=   -R_u*mathlog(n*R_u*T/(self.V*101.3*10**3))
            outS+=n*si
    return(outS)

def sys_gibbs(self,N,T):
    '''
    Calculates the entropy for a set temperature and species count
    For use in getting gibbs

    Takes (sorted) complete species list and temperature.
    Returns total gibb's free energy.
    '''

    outG=self.U0+(T*sum(N)*R_u)-T*self.sys_entropy(N=N,T=T)
    return(outG)

#Gibbs Solver Functions
#---------------------------------
def temp_solve(self,N):
    '''
    Uses a 1D rootfinder to get the temp for a known
    species count and set internal energy.
    'root' imported from scipy

    Takes a sorted complete species list
    Returns the temperature.
    '''

    def f(T):
        return(self.sys_internalenergy(N=N,T=T)-self.U0)
    out=root(f  , x0=self.T0)
```

```
    if float(out.x)<0:
        print("WARNING! No valid temp solution found for N=",N)
    return(float(out.x))

def n_to_N(self,n):
    '''
    Takes a primary species vector 'n' and sorts into secondary
    '''
    ndash=np.dot(   self.Ainv,  (self.b-np.dot(self.D,n)    ))
    N=np.hstack([n,ndash])
    return(N)

def check_physical(self,n=None,N=None):
    '''
    Checks to see if a system state is physical or not

    Takes either primary 'n' or complete species 'N' array
    Returns as a boolean.
    '''

    if type(N)==type(None):
        N=self.n_to_N(n)

    if min(N)<=0:
        return(False)

    if self.sys_internalenergy(N=N,T=0)>self.U0:
        return(False)

    return(True)
```

```
def genseed(self):
    '''
    Randomly pings points in the in the n<nmax domain
    until a physical solution is found

    Takes no arguments
    Returns an array of primary species
    '''
    maxits=1E5
    noits=0
    ntest=np.zeros([self.I-self.J])

    while self.check_physical(n=ntest)==False:
            if noits<maxits:
                ntest=np.random.rand( len(ntest) ) * self.nmax
                noits+=1
            else:
                print("WARNING!: Failed to find physical seed")
                return(None)

    return(ntest)

def windowgibbs(self,n=None,N=None):
    '''
    The potential that is fed into the nelder solver
    Accepts a primary species vector, ensures a physical solution

    Takes either primary 'n' or complete species 'N' array
    Returns the gibb's function with a window applied.
    '''
```

```
        if type(N)==type(None):
            N=self.n_to_N(n)

        if self.check_physical(N=N)==False:
            return(1E100)
        else:
            T=self.temp_solve(N)
            G=self.sys_gibbs(N=N,T=T)
            return(G)

    def LSD_step(self,Y,logN,alpha=1):
        '''
        Performs the newton rapshon step to locate approach the laplace potential
        minimum.

        Takes array inputs in the form:
            -logN=log|n|
            -Y=[log|T|,pi]
            -(optional) alpha=alpha

        Outputs as Y and logN arrays.
        '''

        #Read matrices to extract physical values
        T   =np.exp(Y[0])
        N   =np.exp(logN)+1E-16

        if self.debug:
            print("logN:    ",logN)
            print("N:   ",np.exp(logN))
        if self.debug:
            print(N,T)

        J=np.zeros([self.J+1,self.J+1])
        F=np.zeros([self.J+1])
        A=self.C[:,self.sorter]

        B=np.dot(A,N) #Current  nodes

        #J0 and F0
        J[0,0]  =self.U0/R_u/T
        F[0]    =self.U0/R_u/T
        F[1:]   -=self.b

        MU_RT   =np.zeros([self.I])
        K       =np.zeros([self.I,self.J+1])
        K[:,1:] +=A.T

        #Summate over i
        for i in range(self.I):

            ui  =self.specs[i].u(T)
            hi  =self.specs[i].h(T)

            ui_RT   =ui/R_u/T
            hi_RT   =hi/R_u/T
            mui_RT  =hi_RT-self.specs[i].s(T)/R_u+(logN[i]+mathlog(R_u*T/self.V/101.3/10**3))

            if ZG_SWITCH==True:
                mui_RT+=1
```

```
            cv_R    =self.specs[i].cp(T)/R_u-1

            J[0,0]  +=N[i]*(cv_R-ui_RT*(2-hi_RT))
            F[0]    +=N[i]*ui_RT*(mui_RT-1)

            MU_RT[i]=mui_RT
            K[i,0]  =1-hi_RT

            for row in range(1,self.J+1):
                    for col in range(1,self.J+1):
                        J[row,col]+=A[col-1,i]*A[row-1,i]*N[i]

            for row in range(1,self.J+1):
                J[row,0]+=A[row-1,i]*N[i]*(1-hi_RT)
                F[row]  +=A[row-1,i]*N[i]*(1-mui_RT)

            for col in range(1,self.J+1):
                J[0,col]+=-N[i]*ui_RT*A[col-1,i]

        #Calculate Update
        JINV=np.linalg.inv(J)
        dY  =np.dot(JINV,F)
        dlogN=-MU_RT-np.dot(K,dY)

        #Get outputs
        Yout    =np.zeros([self.J+1])
        Yout[0] =Y[0]+dY[0]*alpha
        Yout[1:]+=dY[1:]

        logNout=logN+dlogN*alpha

        return(Yout,logNout)

    #Equilibrium Optimizers
    #---------------------------------
    def nelder_optimizer(self,n0,tol=1E-10):
        '''
        Solves the equilibrium concentrations by minimizing the
        windowgibbs function. Called by solve().

        Takes arguments
            -Initial primary species vector 'n0'
            -Tolerance 'tol', passed to existing Eilmer solver.

        Outputs are in the form of a primary species array
        '''

        out=minimize(self.windowgibbs,x0=n0,method="nelder-mead",tol=tol)
        return(out.x,out.fun)

    def laplace_optimizer(self,Y0,logN0,tol=1E-10,maxits=1E5,alpha=0.9):
        '''
        Solves the minimum of the laplace potential by itterating
        the newton rapshon equation. Called by solve().

        Takes array inputs in the form:
            -logN0=log|n|
            -Y0=[log|T|,pi]
        And optional inputs
            -(optional) alpha=0.9
```

```
        -(optional) maxits=1E5
        -(optional) tol=1E-10

    Outputs as Y and logN arrays.
    '''

    stepsize=tol*2

    #Set up initial conditions
    Y   =Y0
    logN=logN0

    itno=0
    while stepsize>=tol:
        assert itno<maxits, "Itteration Number exceeded in laplace solver"
        itno+=1
        Ystep,logNstep=self.LSD_step(Y=Y,logN=logN,alpha=alpha)
        stepsize=min(min(abs(Ystep-Y)),min(abs(logNstep-logN)))

        Y   =Ystep
        logN=logNstep

    return(Y,logN)

#Solver Wrapper
#---------------------------------
def solve(self,method='nelder-mead',tol=1E-10,maxits=1E5,alpha=0.9):
    '''
    The actual solver that uses a given method
    to calculate the equilibrium conditions.
Called with solver method as a string:
    method='nelder-mead'
or
    method='LSD'

Takes optional inputs
    -(optional) alpha=0.9
    -(optional) maxits=1E5
    -(optional) tol=1E-10

Returns a tuple of the form:
    (N,T,P)
Where:
    N   is an array of solution species abundances sorted in the order the species were added to the solver's parent sim
    T is the solution temperature
    P is the corresponding pressure.

'''

assert method in ['nelder-mead','steepest-descent','LSD'], method+" is not a valid solver method"

#Check if using constrained method
constrained = method in ['nelder-mead','steepest-descent']

#Get initial conds
if not constrained:
    if min(self.N0)<=0:
        self.N0=self.N0*0+sum(self.b)/self.I

self.n0=self.N0[self.sorter][:self.I-self.J] #Read the primary species from current seed
```

```
if constrained:
    if self.check_physical(self.n0)==False: #If non physical, make a new one
        print("Bad initial seed, making new")
        self.n0=self.genseed()
        self.N0=self.n_to_N(self.n0)
        print("New Seed:",self.N0)

#Actual solver runtime
#---------------------
if method=='nelder-mead':
    #Get results from nelder mead method
    outx,outf=self.nelder_optimizer(self.n0,tol=tol)

    #Sort into meaningful information
    outputN=self.n_to_N(outx)
    outputT=self.temp_solve(outputN)
    outputN=outputN[self.unsorter]

elif method=='steepest-descent':
    print("Steepest Descent not yet implemented")
    return(None)

elif method=='LSD':
    #Get initial conditions
    Y0      =np.zeros([self.J+1])
    Y0[0]   =mathlog(self.T0)
    Y0[1:]   +=self.PI0
    logN0   =np.log(self.N0[self.sorter])

    try:
        Y,logN  =self.laplace_optimizer(Y0,logN0,tol=tol,maxits=maxits,alpha=alpha)
    except:
        alpha=alpha
        while alpha>=0.01:
            try:
                print("Itterations failed to converge. Decreasing Step Scaling")
                alpha*=0.5
                Y,logN  =self.laplace_optimizer(Y0,logN0,tol=tol,maxits=maxits,alpha=alpha)
                break
            except:
                continue
        assert alpha>=0.01, "Simulation failed to converge"
except:
    print("Itterations failed to converge. Resetting Seed")
    self.resetseed()
    outputN,outputT,outputP=solve(self,method='LSD',tol=tol,maxits=maxits)

#Sort into meaningful information
self.PI0=np.zeros([self.J])+Y[1:]
outputN=np.exp(logN)
outputT=np.exp(Y[0])
outputN=outputN[self.unsorter]


outputP=sum(outputN*R_u*outputT/self.V)

self.N0=outputN
self.T0=outputT
```

```
return(outputN,outputT,outputP)
```