```
''' Core methods to calculate the level populations '''

import numpy as np
from constants_lineRT import *
from rt_functions import source_function_thick

''' calculate the partition function
    PARAMS:
      T = temperature (K)
      num_lvls = number of energy levels
      g = statistical weight of each level
      E = energy of each level (eV)'''
def partion_function(T, num_lvls, g, E):
    Z=0.0
    for i in range(0,num_lvls):
        Z = Z + g[i]*np.exp(-E[i]/(kb_ev*T))
    return Z

''' calculate LTE occupation numbers with partition function method
    PARAMS:
      T = temperature (K)
      num_lvls = number of energy levels
      g = statistical weight of each level
      E = energy of each level (eV)
    RETURN:
      ni = level populations '''
def calc_lvlpops_partion(T, num_lvls, g, E):
    ni = []
    Z = partion_function(T, num_lvls, g, E)
    for i in range(0,num_lvls):
        ni.append(g[i]*np.exp(-E[i]/(kb_ev*T)) / Z)
    return ni

''' calculate LTE occupation numbers by solving the system of coupled
 collision balance equations
    PARAMS:
      num_lvls = number of energy levels
      C = netto collision coeff matrix (m3/s)
    RETURN:
      sol = level populations '''
def calc_lvlpops_coll(num_lvls, C):
    rad_field = np.zeros((num_lvls,num_lvls))
    A = np.zeros((num_lvls,num_lvls))
    B = np.zeros((num_lvls,num_lvls))
    density = 1.0 #arbitrary
    sol = calc_lvlpops_nonLTE(num_lvls, A, B, C, density, rad_field)
    return sol

''' calculate non-LTE occupation numbers by solving the system of coupled
 balance equations (A and C)
    without radiation field.
    PARAMS:
      num_lvls = number of energy levels
      A = Einstein A coeff matrix (1/s)
      C = netto collision coeff matrix (m3/s)
```

```
      n_coll = total number density of collision partners (1/m3)
    RETURN:
      sol = level populations '''
def calc_lvlpops_AC(num_lvls, A, C, n_coll):
    rad_field = np.zeros((num_lvls,num_lvls))
    B = np.zeros((num_lvls,num_lvls))
    sol = calc_lvlpops_nonLTE(num_lvls, A, B, C, n_coll, rad_field)
    return sol

''' Full non-LTE, with radiation field:
    Calculate non-LTE occupation numbers by solving the system of coupled
     balance equations
    PARAMS:
      num_lvls = number of energy levels
      A = Einstein A coeff matrix (1/s)
      B = Einstein B coeff matrix (m2/(eV*s))
      C = netto collision coeff matrix (m3/s)
      n_coll = total number density of collision partners (1/m3)
      rad_field = incomming radiation field for each transition I[i][j] (eV/
       s/m2/Hz = eV/m2)
    RETURN:
      sol = level populations '''
def calc_lvlpops_nonLTE(num_lvls, A, B, C, n_coll, rad_field):
    # solve  M*n = 0
    # n = [n1,n2,...ni,...,nn]

    # fill matrix M
    M = np.zeros((num_lvls,num_lvls))
    for a in range(0, num_lvls):
        for b in range(0,num_lvls):
            M_ab = 0
            # upper triangle
            if b>a:
                M_ab = A[b][a] + B[b][a]*rad_field[b][a] + C[b][a]*n_coll
                 #1/s
            # diagonal
            elif a==b:
                for j in range(0, a):
                    M_ab = M_ab - A[a][j] - (B[a][j]*rad_field[a][j]) - C[a]
                     [j]*n_coll
                for j in range(a+1, num_lvls):
                    M_ab = M_ab - B[a][j]*rad_field[j][a] - C[a][j]*n_coll
            # lower triangle
            else:
                M_ab = B[b][a]*rad_field[a][b] + C[b][a]*n_coll
            M[a][b] = M_ab

    # solve M*n=0 with svd: M = U*S*V.T
    U, S, Vt = np.linalg.svd(M)
    # In S the smallest singular value is given last. Check if it
     sufficiently small
    if S[num_lvls-1] > 1.0e-4*S[num_lvls-2]:
        print 'WARNING: unreliable solution:', S
    sol = Vt[num_lvls-1]
```

```
    # sol can be multiplied by a constant factor: normalise to sum(sol_i) =
     1
    # (assumes all values are either positive or neg)
    norm=np.sum(sol)
    sol=sol/norm
    return sol

''' LVG + EscProb method
    Calculate non-LTE occupation numbers by solving the system of coupled
     balance equations with
    the LVG approximation
    PARAMS:
      num_lvls = number of energy levels
      A = Einstein A coeff matrix (1/s)
      B = Einstein B coeff matrix (m2/(eV*s))
      C = netto collision coeff matrix (m3/s)
      n_coll = total number density of collision partners (1/m3)
      rad_field_bg = background radiation field for each transition I[i][j]
       (eV/s/m2/Hz = eV/m2)
      beta =  escape probability for each transition (/)
    RETURN:
      sol = level populations '''
def calc_lvlpops_LVG(num_lvls, A, B, C, n_coll, rad_field_bg, beta):
    # solve  M*n = 0
    # n = [n1,n2,...ni,...,nn]

    # fill matrix M
    M = np.zeros((num_lvls,num_lvls))
    for a in range(0, num_lvls):
        for b in range(0,num_lvls):
            M_ab = 0
            # upper triangle
            if b>a:
                M_ab = A[b][a]*beta[b][a] + B[b][a]*rad_field_bg[b]
                 [a]*beta[b][a] + C[b][a]*n_coll
            # diagonal
            elif a==b:
                for j in range(0, a):
                    M_ab = M_ab - A[a][j]*beta[a][j] - B[a]
                     [j]*rad_field_bg[a][j]*beta[a][j]\
                           - C[a][j]*n_coll
                for j in range(a+1, num_lvls):
                    M_ab = M_ab - B[a][j]*rad_field_bg[j][a]*beta[j][a] -
                     C[a][j]*n_coll
            # lower triangle
            else:
                M_ab = B[b][a]*rad_field_bg[a][b]*beta[a][b] + C[b]
                 [a]*n_coll
            M[a][b] = M_ab

    # solve M*n=0 with svd: M = U*S*V.T
    U, S, Vt = np.linalg.svd(M)
    # In S the smallest singular value is given last. Check if it
     sufficiently small
    if S[num_lvls-1] > 1.0e-4*S[num_lvls-2]:
```

```
        print 'WARNING: unreliable solution', S
    sol = Vt[num_lvls-1]

    # sol can be multiplied by a constant factor: normalise to sum(sol_i) =
     1
    # (assumes all values are either positive or neg)
    norm=np.sum(sol)
    sol=sol/norm
    return sol

#-------------------------------------------------------------------------
 ------------------------

''' Self-consistently solve for the level populations with the general
 method
    in the optical thick limit!
    PARAMS:
      line_data = lineData object with info about the transitions
      n_coll = total number density of collision partners (1/m3)
      comp_fracs = fractions of n_coll for each collision partner
      n_molec = number density of the molecule (1/m3)
      T = temperature (K)
      rad_field_bg =  background radiation field (???)
      num_iter = number of iterations for determining the lvl pops'''
def solve_lvlpops_nonLTE(line_data, n_coll, comp_fracs, n_molec, T,
 rad_field_bg, num_iter):

    # calc netto collision coeffs
    C = line_data.calc_total_C(T,comp_fracs)

    # set initial radiation field
    source = np.zeros((line_data.num_lvls,line_data.num_lvls))
    #J = emis * 4.*np.pi #* dx # eV/m2
    J = rad_field_bg

    for it in range(num_iter):
        lvl_pops = calc_lvlpops_nonLTE(line_data.num_lvls, line_data.A,
         line_data.B, C, n_coll, J)
        # calculate the emissivity
        for i in range(line_data.num_lvls):
            for j in range(i):
                if line_data.freq[i][j] != 0.:
                    source[i][j] = source_function_thick(line_data.freq[i]
                     [j],
                                                         lvl_pops[i], lvl_pops[j],
                                                         n_molec, line_data.A[i][j],
                                                         line_data.B[i][j],
                                                          line_data.B[j][i])
        J = source
        #J = J + rad_field_bg

    return lvl_pops
```

```
#-------LVG
 solver-------------------------------------------------------------------
 -------------

''' Self-consistently solve for the level populations with LVG approximation
    PARAMS:
      line_data = lineData object with info about the transitions
      n_coll = total number density of collision partners (1/m3)
      comp_fracs = fractions of n_coll for each collision partner
      n_molec = number density of the molecule (1/m3)
      T = temperature (K)
      grad_v = velocity gradient (s-1)
      rad_field_bg =  background radiation field (???)
      num_iter = maximum number of iterations for determining the lvl
       pops'''
def solve_lvlpops_LVG(line_data, n_coll, comp_fracs, n_molec, T, grad_v,
 rad_field_bg, num_iter, flag_reduce=False):

    if flag_reduce:
        # reduce the number of lines to only the relevent ones
        lvl_pops = calc_lvlpops_partion(T, line_data.num_lvls, line_data.g,
         line_data.E)
        relevant_lvls = len(lvl_pops)
        while (relevant_lvls>5) and (lvl_pops[relevant_lvls-1]<1e-8):
            relevant_lvls = relevant_lvls - 1

        ld = line_data.reduce_linedata(relevant_lvls)
    else:
        ld = line_data

    # calc netto collision coeffs
    C = ld.calc_total_C(T, comp_fracs)

    # initialize x_i to LTE
    lvl_pops = calc_lvlpops_partion(T, ld.num_lvls, ld.g, ld.E)

    # calculate tau and beta for LVG
    tau, beta = LVG_coeffs_all_lines(n_molec, lvl_pops, ld.B, grad_v)
    T_ex_prev = 0.0
    T_ex_curr = calc_T_ex(1, 0, lvl_pops, ld.g, ld.E)
    it=0
    convergence = abs(T_ex_curr-T_ex_prev)/(T_ex_curr+T_ex_prev)/2.

    # update level pops with LVG and iterate
    while (convergence>0.01) and (it<num_iter):
        it=it+1
        lvl_pops = calc_lvlpops_LVG(ld.num_lvls, ld.A, ld.B, C, n_coll,
         rad_field_bg, beta)
        tau, beta = LVG_coeffs_all_lines(n_molec, lvl_pops, ld.B, grad_v)
        T_ex_prev = T_ex_curr
        T_ex_curr = calc_T_ex(1, 0, lvl_pops, ld.g, ld.E)
        convergence = abs(T_ex_curr-T_ex_prev)/(T_ex_curr+T_ex_prev)/2.
        #print 'iteration {} -> convergence {:1.3}%'.format(it, convergence)

    if flag_reduce:
```

```
        # pad up irrelavant part
        restored_lvl_pops = np.zeros(line_data.num_lvls)
        restored_lvl_pops[:relevant_lvls] = lvl_pops
        restored_beta = np.zeros((line_data.num_lvls,line_data.num_lvls))
        restored_beta[:relevant_lvls,:relevant_lvls] = beta
        return restored_lvl_pops, restored_beta
    else:
        return lvl_pops, beta

''' Calculate tau and beta in LVG approx for all lines
    PARAMS:
      n_molec = number density of the molecule (1/m3)
      lvl_pops = level populations
      B = Einstein B coeff matrix (m2/(eV*s))
      grad_v = velocity gradient (1/s) '''
def LVG_coeffs_all_lines(n_molec, lvl_pops, B, grad_v):
    ni, nj = B.shape
    tau = np.ones((ni,nj))
    beta = np.ones((ni,nj))
    for i in range(ni):
        for j in range(i):
            if B[i][j]==0:
                tau[i][j] = 1.0 # arbitrary
                beta[i][j] = 1.0 #0.0
            else:
                tau[i][j] = tau_LVG(n_molec, grad_v, lvl_pops[i],
                 lvl_pops[j], B[i][j], B[j][i])
                beta[i][j] = beta_LVG(tau[i][j])
                #print 'i {} j {} tau={} beta={}'.format(i,j,tau[i]
                 [j],beta[i][j])
    return tau, beta

''' Optical depth in LVG approximation for line ij '''
def tau_LVG(n_molec, grad_v, x_i, x_j, B_ij, B_ji):
    # units: m/s * eV*s * 1/m3 * m2/(eV*s) / (1/s) = none
    return c_si*h_ev/(4.*np.pi) * n_molec * (x_j*B_ji - x_i*B_ij)  /
     (1.064*grad_v)

''' Escape probability in LVG approximation for line ij
    PARAMS:
      tau = optical depth'''
def beta_LVG(tau):
    if tau < 0.01:
        return 1. - tau/2.
    elif tau > 100.:
        return 1./tau
    else:
        return (1.0 - np.exp(-tau)) / tau

''' Determine the density for which tau LVG is 1 using B coeffs '''
def critical_density_B(grad_v, x_i, x_j, B_ij, B_ji, abundance):
    n_molec_crit = (4.*np.pi*1.064*grad_v)/((x_j*B_ji - x_i*B_ij)*c*h)
    print 'n_crit', n_molec_crit/ 1e6 / abundance, 'H2/cc'
    return n_molec_crit / abundance * (2*MH) # kg/m3
```

```
''' Determine the density for which tau LVG is 1 using A coeff '''
def critical_density_A(grad_v, x_i, x_j, A_ij, freq, g_i, g_j, abundance):
    n_molec_crit = (8.*np.pi*(freq**3)*1.064*grad_v)/((x_j*(g_i/g_j) -
     x_i)*A_ij*(c**3))
    print 'n_crit', n_molec_crit/ 1e6 / abundance, 'H2/cc'
    print 'args rho_crit:', grad_v, x_i, x_j, A_ij, freq, g_i, g_j,
     abundance
    return n_molec_crit / abundance * (2*MH) # kg/m3

''' TODO Estimate the critical density in function of grad_v in the
 simulation '''

#-------------------------------------------------------------------------
 ------------------------
''' Calculate the excitation temperature. This is the LTE temperature
 corresponding to the given x_i
    PARAMS:
      u = number of the upper energy level (0=ground state)
      d = number of the lower energy level
      x = array of occupation numbers
      g = weights of all levels
      E = energy of all levels (eV) '''
def calc_T_ex(u, d, x, g, E):
    if x[u]==0.:
        return 0.0
    elif x[d]==0.:
        return 0.0 #should be inf but this is annoying to plot
    else:
        return -(E[u]-E[d])/(kb_ev * np.log(x[u]*g[d]/(x[d]*g[u])))
```