```
'''
    LINE RADIATIVE TRANSFER

    Post-process RAMSES simulations to generate mock observations of
     molecular lines.
    Can generate images and spectra.
    Can only do one line at a time.

    To use:
    python mock_box_parallell.py <resolution> <molecule> <up> <down>

'''

import sys
import time
import gc
import numpy as np
import pyfits
import multiprocessing as mp
import matplotlib
matplotlib.use('agg')

from lineData import LineData
from calc_occupations import *
from constants_lineRT import *
from rt_functions import *
from get_density import *
from species_info import *
from observe import *

def do_rt_stuff(res, species, comp_fracs, abundance, up=1, down=0,
 sim='elena', ncpu=33):

    start_all=time.time()

    #----SETUP----
    start_setup=time.time()

    # Get simulation data (in SI units)
    nx, ny, nz, dx, rho, vx, vy, vz, grad_v = load_sim_data(sim, res,
     statistics=False)

    # fix temperature, TODO get this from simulation for each cell
    T = 10.0 #K fixed for now
    mu = 2.37
    c_s = np.sqrt(kb_si*T/(mu*MH))

    # get line data and coefficients
    ld = LineData(species)

    # calculate C coeffs. Since this in dependent on T, it should in
     principle be done for each cell
    C = ld.calc_total_C(T, comp_fracs)

    # get number densities in 1/m3
```

```
if species=='CO':
    UV_rad_field = get_rad_field(rho, dx)
    n_H2 = get_H2_density(rho, UV_rad_field)
    n_molec = get_CO_density(rho, n_H2, UV_rad_field)
else:
    n_H2 = rho/(2.*MH)
    n_molec = n_H2 * abundance #m-3

# diagnostics
M_tot = np.sum(rho) * dx**3/ MSUN
M_H2 = np.sum(n_H2) * dx**3 * 2*MH / MSUN
M_molec = np.sum(n_molec) * dx**3 * 2*MH /MSUN
print 'M_tot = {}, M_H2 = {} Msun, M_molec = {} Msun'.format(M_tot,
 M_H2, M_molec)

# Set initial radiation field to the CMB (same for all cells)
# TODO this is in contradition with how n_CO is calculated
rad_field_CMB = np.zeros((ld.num_lvls, ld.num_lvls))
T_bg = 2.725 # cmb
for i in range(ld.num_lvls):
    for j in range(ld.num_lvls):
        if ld.freq[i][j] != 0.0:
            rad_field_CMB[i][j] = B_nu_ev(ld.freq[i][j], T_bg) # eV/m2

end_setup = time.time()
print 'Finished setup in:', end_setup-start_setup, 's'

#----CALC LVLPOPS----
start_lvl=time.time()

num_iter = 10 #max number of iteration for solving level populations

# Determine the level populations and escape fraction for each cell in
 parallel
beta, lvl_pops = multi_proc_lvlpops(ld, grad_v, c_s, dx, n_H2,
 comp_fracs,
                                    n_molec, T, rad_field_CMB,
                                    num_iter, up, down, ncpu)
end_lvl = time.time()
print 'Finished calculation of the level populations in:', end_lvl-
 start_lvl, 's'

#----GENERATE OBSERVATIONS----
start_obs=time.time()

# get statistics on beta
outname = 'beta_pdf_{}{}-{}_abu{:1.1e}_T{}_res{}.png'.format(species,
 up,down, abundance, T, res)
calc_PDF(beta, outname, units='/')

# observe the box from each axis
for axis in ['vx','vy','vz']:
    if axis=='vx':
        v_los = vx
    elif axis=='vy':
```

```
            v_los = vy
        else: #axis=='vz'
            v_los = vz
        multi_proc_observe(axis, v_los, ld, up, down, lvl_pops[:,:,:,up],
         n_molec, T,
                           beta, dx)

    end_obs = time.time()
    print 'Finished observing in:', end_obs-start_obs, 's'

    end_all = time.time()
    print 'Finished run. Total calculation time:', end_all-start_all

''' Calculate the level populations of the cells in parallel '''
def multi_proc_lvlpops(ld, grad_v, c_s, dx, n_H2, comp_fracs, n_molec, T,
 rad_field_CMB,
                   num_iter, up, down, ncpu):

    (nx, ny, nz) = n_molec.shape
    lvl_pops = np.zeros((nx,ny,nz,ld.num_lvls))
    beta = np.zeros((nx,ny,nz))

    num_workers = ncpu-1
    chunk = int(np.ceil(float(nx)/num_workers))

    # spawn a process for each slice of the datacube (one could also flatten
     the array)
    output = mp.Queue()
    processes = [mp.Process(target=lvlpops_core_func,\
                            args=(i, ld, grad_v[i:i+chunk,:,:], c_s, dx,\
                                  n_H2[i:i+chunk,:,:], comp_fracs,
                                   n_molec[i:i+chunk,:,:], T,
                                  rad_field_CMB,\
                                  num_iter, up, down, output)) for i in
                                   range(0, nx, chunk)]
    for p in processes:
        p.start()

    # retrieve output
    for p in processes:
        r = output.get()
        i = r[0]
        beta[i:i+chunk,:,:] = r[1]
        lvl_pops[i:i+chunk,:,:,:] = r[2]

    for p in processes:
        p.join()
    gc.collect()

    return beta, lvl_pops

''' Function that does the main calculation of the level populations for a
 part of the cube '''
def lvlpops_core_func(i_start, ld, grad_v, c_s, dx, n_H2, comp_fracs,
 n_molec, T,
```

```
                    rad_field_CMB, num_iter, up, down, output):

    (nx, ny, nz) = n_molec.shape
    lvl_pops = np.zeros((nx,ny,nz,ld.num_lvls))
    beta_cell = np.zeros((ld.num_lvls,ld.num_lvls))
    beta_trans = np.zeros((nx,ny,nz))
    for i in range(nx):
        for j in range(ny):
            for k in range(nz):
                dv = max(grad_v[i][j][k], c_s/dx)
                lvl_pops[i][j][k], beta_cell = solve_lvlpops_LVG(ld, n_H2[i]
                 [j][k], comp_fracs,
                                                        n_molec[i][j][k],
                                                    T, dv, rad_field_CMB,
                                                        num_iter,
                                                    flag_reduce=True)
                beta_trans[i][j][k] = beta_cell[up][down]

    output.put((i_start, beta_trans, lvl_pops))
    print 'Finished', i_start, 'to', i_start + nx - 1

''' Deal with the observing stuff
    x_up = level population of the upper level of the transition for each
     cell
    beta = escape probability for transition up-down'''
def multi_proc_observe(axis, v_los, ld, up, down, x_up, n_molec, T, beta,
 dx):

    (nx, ny, nz) = n_molec.shape

    n_bins = 50
    global_units = 'K' #'W'

    # split the relevant velocity range in bins
    v_bins = np.linspace(np.min(v_los),np.max(v_los), n_bins+1)
    v_bin_size = v_bins[1] - v_bins[0]
    # convert to doppler shift frequency range
    freq_bins = ld.freq[up][down] * (1. + v_bins/c_si)
    freq_bin_size = v_bin_size * ld.freq[up][down]/c_si
    print freq_bins

    spectrum = np.zeros(n_bins)
    if axis=='vx':
        (im_x, im_y) = (ny, nz)
    elif axis=='vy':
        (im_x, im_y) = (nx, nz)
    else: #axis=='vz'
        (im_x, im_y) = (nx, ny)
    total_intensity = 0.

    # make a process for each wavelength bin
    output = mp.Queue()

    processes = [mp.Process(target=core_func_observe, args=(f, freq_bins[f],
     freq_bins[f+1],\
```

```
                            ld, up, down, x_up, beta, n_molec, T, nx, ny,
                             nz, dx, v_los ,\
                            axis, global_units, output)) for f in
                             range(n_bins)]
for p in processes:
    p.start()

# calculate and plot the integrated image directly (this way is
 independent of line profile)
emis_tot = integrated_emissivity(ld.freq[up][down], x_up, n_molec,
 ld.A[up][down])
direct_integrated_im, units_im = calc_image(emis_tot, beta, dx,
 ld.freq[up][down],
                                             global_units, axis,
                                              distance=1.e4,
                                              integrated=True)
outname = 'image_integrated_{}_{}{}-{}_abu{:1.1e}_T{}
 _res{}.png'.format(axis, species,
                                                   up, down, abundance,
                                                T, res)
plot_image(direct_integrated_im, units_im, 'direct'+outname, log=False)

# retrieve and process output
image_cube = np.zeros((n_bins, im_x, im_y))
for p in processes:
    r = output.get()
    f = r[0]
    image_cube[f] = r[1]
    units_im = r[2]
    spectrum[f] = r[3]
    units_sp = r[4]
    # add to the total intensity of the entire image
    if global_units=='K':
        bin_size = v_bin_size * 1.e-3 # km/s
    else:
        bin_size = freq_bin_size # Hz
    total_intensity = total_intensity + spectrum[f]*bin_size

for p in processes:
    p.join()

# output FITS file with the full image cube
outname = 'cube_{}_{}{}-{}_abu{:1.1e}_T{}_res{}.fits'.format(axis,
 species,
                                                          up, down,
                                                abundance, T, res)
write_fits(image_cube, outname)

# plot the spectrum
if global_units=='K':
    bins = v_bins*1.e-3
    units_bins = 'km/s'
    # check totals
    print 'total value from spectrum', total_intensity, 'K pc2 km/s'
```

```
        print 'total value from integrated image',
         np.sum(direct_integrated_im) * (dx/PC)**2,\
              'K km/s pc2'
    else:
        bins = freq_bins
        units_bins = 'Hz'
        # check totals
        print 'total value from spectrum', total_intensity, 'W'
        print 'total value from integrated image',
         np.sum(direct_integrated_im) * dx**2, 'W'

    outname = 'spectrum_{}_{}{}-{}_integr_abu{:1.1e}_T{}
     _res{}.png'.format(axis, species, up, down,

                                                  abundance, T, res)
    plot_spectrum(spectrum, units_sp, bins, units_bins, outname)

''' function that calculates the image and spectrum value in a wavelength
 bin '''
def core_func_observe(f, nu_a, nu_b, ld, up, down, lvl_pop_up, beta_up_down,
 n_molec, T,
                      nx, ny, nz, dx, v_los, axis, global_units, output):

    #print 'observing band', f, nu_a, nu_b
    band_center = (nu_b+nu_a)/2.
    freq_bin_size = nu_b - nu_a

    # calculate the emissivity of each cell of line nu_ij in frequency range
     [nu_a, nu_b]
    # (including doppler shift)
    emis_part = emissivity_part(nu_a, nu_b, ld.freq[up][down], lvl_pop_up,
     n_molec, ld.A[up][down],\
                                T, ld.mu, integrate_thermal_profile,
                                 v_los=v_los)

    # calculate the image and spectral bin at this frequency
    image, units_im = calc_image(emis_part, beta_up_down, dx, ld.freq[up]
     [down], global_units, axis,
                                 distance=1.e4, integrated=False)
    spect_val, units_sp = calc_spectrum_value(emis_part, beta_up_down, dx,
     ld.freq[up][down],
                                           global_units, axis,
                                            distance=1.e4)

    # plot the specific image
    #outname = 'image_{}_{}{}-{}_at{}Hz_abu{:1.1e}_T{}
     _res{}.fits'.format(axis, species, up, down,
    #                                                  band_center,
     abundance, T, res)
    #plot_image(image, units_im, outname, log=False)
    output.put((f, image, units_im, spect_val, units_sp))

''' Write a fits file of data_cube to outname '''
def write_fits(data_cube, outname):
    pyfits.writeto(outname, data_cube, clobber=True)
```

```
#-------------------------------------------------------------------------
 ------------------------

if __name__=='__main__':

    res = int(sys.argv[1])
    species = sys.argv[2]
    up = int(sys.argv[3])
    down = int(sys.argv[4])
    ncpu = int(sys.argv[5])

    sim='elena'#'large_box' #'box' #'elena'
    # provide code units and boxlen in get_density's code_units(sim)
     function

    comp_fracs, abundance = load_species_info(species)

    do_rt_stuff(res, species, comp_fracs, abundance, up, down, sim=sim,
     ncpu=ncpu)
```