opac

October 5, 2025

Connected to base (Python 3.11.6)

[ ]:
```
import astropy.units as u
import astropy.constants as c
import numpy as np
from astropy.io import fits
import matplotlib.pyplot as plt
#import grey_model.old.saha_eos as eos
from scipy.interpolate import RectBivariateSpline
from scipy.special import voigt_profile
from molecular_lines import MolecularLines
plt.ion()

"""
This is based on:
https://iopscience.iop.org/article/10.3847/1538-4357/ac9b40/pdf

NB there are other resources for non-stellar atmosphere opacities,
e.g.
https://chiantipy.readthedocs.io/en/latest/
https://chianti-atomic.github.io/api/ChiantiPy.core.html#id91
http://spiff.rit.edu/classes/phys370/lectures/statstar/statstar_python3.py

"""
import eos as eos
_ = a = eos.P_T_tables(None, None, savefile='saha_eos.fits')

#From OCR online, from https://articles.adsabs.harvard.edu/pdf/1988A%26A...193..
 ↪189J
#A to F in columns, n=2 to 6 in rows
Hmff_table = np.array(
[[2483.3460 ,          285.8270   ,      -2054.2910    ,   2827.7760     , ␣
 ↪-1341.5370     ,       208.9520],
[-3449.8890 ,         -1158.3820  ,      8746.5230     ,   -11485.6320   , ␣
 ↪5303.6090      ,      -812.9390],
[2200.0400  ,          2427.7190  ,       -13651.1050  ,    16755.5240   , ␣
 ↪-7510.4940    ,      1132.7380],
```

```
[-696.2710   ,       -1841.4000    ,       8624.9700    ,    -10051.5300   ,
 ↪4400.0670    ,      -655.0200],
[88.2830    ,         444.5170    ,      -1863.8640    ,     2095.2880   ,
 ↪-901.7880     ,       132.9850]])

Hff_const = np.sqrt(32*np.pi)/3/np.sqrt(3)*(c.e.esu**6/c.c/c.h/(c.k_B*c.
 ↪m_e**3*u.K)**(1/2)/(1*u.Hz)**3).to(u.cm**5).value
h_kB_cgs = (c.h/c.k_B).cgs.value
H_excitation_T = (13.595*u.eV/c.k_B).cgs.value
f_const = (np.pi*c.e.gauss**2/c.m_e/c.c).cgs.value  # Used in line calcs.
ev_kB_cgs = (1*u.eV/c.k_B).cgs.value

 #Element abundances
abund, masses, n_p, ionI, ionII, gI, gII, gIII, elt_names = eos.composition()

nelt = len(abund)
# Read on strong_lines and weak_lines files
strong_lines = fits.getdata('strong_lines.fits',1)
strong_nu = c.c.to(u.AA/u.s).value/strong_lines['wavelength']
weak_lines = fits.getdata('weak_lines.fits',1)
weak_nu = c.c.to(u.AA/u.s).value/weak_lines['wavelength']

# Create indices of each element and ion in weak_lines
line_elts = np.unique(weak_lines['element_name'])
neutral_indices = {}
ion_indices = {}
for name in line_elts:
    neutral_indices[name] = np.where((weak_lines['element_name'] == name) &
 ↪(weak_lines['ion_state']==1))[0]
    ion_indices[name] = np.where((weak_lines['element_name'] == name) &
 ↪(weak_lines['ion_state']==2))[0]

# Read in the equation of state and make the relevant 2D interpolation functions
f_eos = fits.open('saha_eos.fits')
h = f_eos[0].header
Ts = h['CRVAL1'] + np.arange(h['NAXIS1'])*h['CDELT1']
Ps_log10 = h['CRVAL2'] + np.arange(h['NAXIS2'])*h['CDELT2']
rho = f_eos['rho [g/cm**3]'].data
ns = f_eos['ns [cm^-3]'].data
ne_table = f_eos['n_e [cm^-3]'].data

# Here we can hack Neutral abundances.
#Fe = np.where(elt_names == 'Fe')[0]
#ns[:,:,3*Fe ] *= 0.001   #Neutral
#ns[:,:,3*Fe+1] *= 0.001 #Ionized
```

```
# Create 2D interpolation functions for each element/ion species in ns
# ns has shape (len(Ps_log10), len(Ts), number_of_elements)
number_of_elements = ns.shape[2]
log10ns = []
for i in range(number_of_elements):
    # Create 2D interpolation function for particle i
    # RectBivariateSpline expects (x, y, z) where z[i,j] = f(x[i], y[j])
    interp_func = RectBivariateSpline(Ps_log10, Ts, np.log10(ns[:,:,i]))
    log10ns.append(interp_func)
log10ne = RectBivariateSpline(Ps_log10, Ts, np.log10(ne_table))

def weak_line_kappa(nu0, dlnu, N_nu, log10P, T, microturb=1.5):
    """ For all atomic and ion species, compute the weak line opacities.
    nu0: Start frequency in Hz
    dlnu: delta log(nu)
    N_nu: number of frequencies
    log10P: Log10 of pressure in dyne/cm^2
    T: Temperature in K
    microturb: Microturbulence parameter (default is 2.0 km/s)
    """
    kappa = np.zeros(N_nu)
    max_nu = nu0 * np.exp(dlnu * (N_nu - 1))
    # Loop through all elements
    for name in line_elts:
        for ion_state in [1,2]:
            this_kappa = np.zeros_like(kappa)
            if ion_state == 1 and name in neutral_indices:
                indices = neutral_indices[name]
            elif ion_state == 2 and name in ion_indices:
                indices = ion_indices[name]
            else:
                continue
            # Remove lines outside our wavelength range
            indices = indices[(nu0<weak_nu[indices]) &␣
 ↪(weak_nu[indices]<max_nu)]
            if len(indices) == 0:
                continue

            # Compute the index of the weak_nu
            weak_ix = (np.log(weak_nu[indices]) - np.log(nu0))/dlnu
            weak_ix0 = np.clip(weak_ix, 0, N_nu-2).astype(int)
            weak_frac = weak_ix - weak_ix0

            # Number density
            elt_ix = np.where(elt_names == name)[0][0]
            if ion_state == 1:
                n = 10**(log10ns[3*elt_ix](log10P, T)[0][0])
```

```
                Zpart = gI[elt_ix]
            elif ion_state == 2:
                n = 10**(log10ns[3*elt_ix + 1](log10P, T)[0][0])
                Zpart = gII[elt_ix]

            # Opacity
            kappa_tot = n * f_const * 10**weak_lines['log_gf'][indices] * np.
 ↪exp(-weak_lines['excitation'][indices]*ev_kB_cgs/T)*(1-np.
 ↪exp(-h_kB_cgs*weak_nu[indices]/T)) / weak_nu[indices] / Zpart
            if min(kappa_tot) < 0:
                import pdb; pdb.set_trace()
            # A fast vectorised way to add all kappa values
            np.add.at(this_kappa, weak_ix0, (1 - weak_frac) * kappa_tot)
            np.add.at(this_kappa, weak_ix0+1, weak_frac * kappa_tot)

            # Compute the line width, in units of the grid spacing.
            elt_ix = np.where(elt_names == name)[0]
            line_width = np.sqrt(2*(c.k_B* T*u.K) / (masses[elt_ix]*u.u)).to(u.
 ↪km/u.s).value
            line_width = np.sqrt(line_width**2 + microturb**2)
            grid_line_width = line_width/c.c.to(u.km/u.s).value/dlnu

            #Offsets of up to +/- 3 sigma
            offsets = np.arange(-int(3*grid_line_width)-1,␣
 ↪int(3*grid_line_width)+2)
            #A Gaussian kernel
            gaussian_kernel = np.exp(-(offsets / grid_line_width)**2)
            # Normalize the kernel
            gaussian_kernel /= np.sum(gaussian_kernel)*dlnu

            # Convolve the opacity with the Gaussian kernel
            kappa += np.convolve(this_kappa, gaussian_kernel, mode='same')

    return kappa

def strong_line_kappa(nu0, dlnu, N_nu, log10P, T, microturb=2.0):
    """ For all atomic and ion species, compute the strong line opacities.
    nu0: Start frequency in Hz
    dlnu: delta log(nu)
    N_nu: number of frequencies
    log10P: Log10 of pressure in dyne/cm^2
    T: Temperature in K
    microturb: Microturbulence parameter (default is 2.0 km/s)
    """
    nu = np.exp(np.linspace(np.log(nu0), np.log(nu0 * np.exp(dlnu * N_nu)),␣
 ↪N_nu))
    kappa = np.zeros(N_nu)
```

```
    max_nu = nu0 * np.exp(dlnu * (N_nu - 1))

    # Get unique elements and ions from strong_lines
    strong_line_elts = np.unique(strong_lines['element_name'])

    # Find the current n_e and N_H (needed for broadening)
    n_e = 10**(log10ne(log10P, T)[0][0])
    n_H = 10**(log10ns[0](log10P, T)[0][0])

    # Loop through all elements
    for name in strong_line_elts:
        for ion_state in [1, 2]:
            # Find indices for this element and ion state
            indices = np.where((strong_lines['element_name'] == name) &
                               (strong_lines['ion_state'] == ion_state))[0]

            # Remove lines outside our wavelength range
            indices = indices[(nu0 < strong_nu[indices]) & (strong_nu[indices]␣
↪< max_nu)]
            if len(indices) == 0:
                continue

            # Get element index for number density lookup
            elt_ix = np.where(elt_names == name)[0][0]

            # Number density
            if ion_state == 1:
                n = 10**(log10ns[3*elt_ix](log10P, T)[0][0])
                Zpart = gI[elt_ix]
            elif ion_state == 2:
                n = 10**(log10ns[3*elt_ix + 1](log10P, T)[0][0])
                Zpart = gII[elt_ix]

            # Pre-compute Doppler velocity for this element (optimization 3)
            doppler_v = np.sqrt(2*(c.k_B* T*u.K) / (masses[elt_ix]*u.u)).to(u.
↪km/u.s).value
            doppler_dlnu = np.sqrt(doppler_v**2 + microturb**2)/c.c.to(u.km/u.
↪s).value

            # Loop through all lines for this element/ion
            for idx in indices:
                line_nu = strong_nu[idx]

                # Pre-compute frequency-dependent Doppler width
                doppler_dnu = doppler_dlnu * line_nu

                if strong_lines['line_strength'][idx] < 1e-7:
```

```
                    idx_range = int(0.01/dlnu)
                else:
                    idx_range = int(0.03/dlnu)
                # Find the start and end indices within +/- 1%
                line_idx = int((np.log(line_nu) - np.log(nu0))/dlnu)
                start_idx = np.maximum(line_idx - idx_range, 0)
                end_idx = np.minimum(line_idx + idx_range + 1, N_nu-1)

                # Compute Gamma. ignore van der Waals for now␣
↪strong_lines['waals'][idx]
                Gamma = 10**(strong_lines['rad'][idx])
                if (elt_ix == 0 and ion_state == 1):
                    Gamma += n_e * 1e-4 #Completely made up! Hydrogen is␣
↪specially treated in VALD3
                elif (strong_lines['stark'][idx] != 0):
                    Gamma += n_e * 10**strong_lines['stark'][idx]
                this_kappa = n * f_const * 10**strong_lines['log_gf'][idx] * np.
↪exp(-strong_lines['excitation'][idx]*ev_kB_cgs/T)*(1-np.
↪exp(-h_kB_cgs*strong_nu[idx]/T)) / Zpart

                #Uncomment to see if the strong lines of Ca make sense.
                #if (ion_state == 2) and (name == 'Ca') and␣
↪(strong_lines['line_strength'][idx] > 1e-7):
                #    import pdb; pdb.set_trace()

                # Add the opacity to the kappa array
                kappa[start_idx:end_idx] += this_kappa *␣
↪voigt_profile(nu[start_idx:end_idx] - line_nu, doppler_dnu/np.sqrt(2), Gamma/
↪4/np.pi)
    return kappa

def Hmbf(nu, T):
    """Compute the Hydrogen minus bound-free cross sections in cgs units as a
    function of temperature in K. Computed per atom. Compute using:
    https://ui.adsabs.harvard.edu/abs/1988A%26A...193..189J/abstract

    Parameters
    ----------
    nu: Frequency or a list (numpy array) of frequencies.
    """
    Cn = [152.519, 49.534, -118.858, 92.536,-34.194,4.982]
    nu_val = nu.to_value(u.Hz) if hasattr(nu, 'unit') else nu
    alpha = np.zeros_like(nu_val)
    wave_um = c.c.si.value/nu_val * 1e6
    for n in range(1,7):
        alpha += Cn[n-1] * np.abs(1/wave_um - 1/1.6419)**((n-1)/2)
```

```
    alpha *= (wave_um<=1.6419) * 1e-18 * wave_um**3 * np.abs(1/wave_um - 1/1.
↪6419)**(3/2)
    # Ensure T is float in K
    T_val = T.to_value(u.K) if hasattr(T, 'unit') else T
    return alpha * (1-np.exp(-h_kB_cgs*nu_val/T_val))


def Hmff(nu, T):
    """Compute the Hydrogen minus bound-free cross sections in cgs units as a
    function of temperature in K. Computed per H atom per unit (cgs) electron
    density. Compute using:
    https://ui.adsabs.harvard.edu/abs/1988A%26A...193..189J/abstract

    Parameters
    ----------
    nu: Frequency or a list (numpy array) of frequencies.
    """
    nu_val = nu.to_value(u.Hz) if hasattr(nu, 'unit') else nu
    alpha = np.zeros_like(nu_val)
    wave_um = np.maximum(c.c.si.value/nu_val * 1e6,0.3645)
    T_val = T.to_value(u.K) if hasattr(T, 'unit') else T
    for n in range(2,7):
        row = n-2
        coeff = 1e-29 * (5040/T_val)**((n+1)/2)
        for i, exponent in enumerate([2,0,-1,-2,-3,-4]):
            alpha += coeff*wave_um**exponent * Hmff_table[row,i]
    #alpha is now in units of cross section per unit electron pressure
    #We want to multiply by the ratio of electron pressure to electron
    #density, which is just k_B T
    return alpha * c.k_B.cgs.value * T_val

def Hbf(nu, T):
    """Compute the Hydrogen bound-free cross sections in cgs units as a
    function of temperature in K. Computed per atom. Computed using:
    https://articles.adsabs.harvard.edu/pdf/1970SAOSR.309.....K

    Parameters
    ----------
    nu: Frequency or a list (numpy array) of frequencies.
    """
    alpha = np.zeros_like(nu)
    ABC = np.array([[.9916,2.719e13,-2.268e30],
        [1.105,-2.375e14,4.077e28],
        [1.101,-9.863e13,1.035e28],
        [1.101,-5.765e13,4.593e27],
        [1.102,-3.909e13,2.371e27],
        [1.0986,-2.704e13,1.229e27]])
```

```
    nu_val = nu.to_value(u.Hz) if hasattr(nu, 'unit') else nu
    T_val = T.to_value(u.K) if hasattr(T, 'unit') else T
    alpha = np.zeros_like(nu_val)
    ABC = np.array([[.9916,2.719e13,-2.268e30],
        [1.105,-2.375e14,4.077e28],
        [1.101,-9.863e13,1.035e28],
        [1.101,-5.765e13,4.593e27],
        [1.102,-3.909e13,2.371e27],
        [1.0986,-2.704e13,1.229e27]])
    for n in range(1,7):
        Boltzmann_fact = n**2*np.exp(-H_excitation_T*(1-1/n**2)/T_val)
        alpha += 2.815e29/n**5/nu_val**3*(ABC[n-1,0] + (ABC[n-1,1] + ABC[n-1,2]/
↪nu_val)/nu_val) * (nu_val>3.28805e15/n**2) * Boltzmann_fact
    #FIXME : add higher values of n
    #FIXME : Add in the partition function U, which is implicitly taken to be 2.
↪0 above.
    return alpha * (1-np.exp(-h_kB_cgs*nu_val/T_val))


def Hff(nu, T):
    """Compute the Hydrogen free-free cross sections in cgs units as a
    function of temperature in K. Computed per atom per unit (cgs) electron
    density

    Parameters
    ----------
    nu: Frequency or a list (numpy array) of frequencies.
    """
    #Approximate a Gaunt factor of 1.0!
    #FIXME : Remove the approximation
    nu_val = nu.to_value(u.Hz) if hasattr(nu, 'unit') else nu
    T_val = T.to_value(u.K) if hasattr(T, 'unit') else T
    return Hff_const /nu_val**3/np.sqrt(T_val)

def HeIbf(nu, T):

    import astropy.constants as c

    h_kB_cgs = (c.h/c.k_B).cgs.value

    """Compute the Helium bound-free cross sections in cgs units as a
    function of temperature in K. Computed per atom. Computed using:
    https://articles.adsabs.harvard.edu/pdf/1970SAOSR.309.....K

    Parameters
    ----------
    nu: Frequency or a list (numpy array) of frequencies.
```

```
    """
    he_level =[# Level 1: 1S
    [1,1,0.0,5.9447e15,[33.32,-2.0]],
    # Level 2: 2^3S
    [2,3,19.819,1.1526e15,[-390.026,21.035,-0.318]],
    # Level 3: 2^1S
    [3,1,20.615,0.96025e15,[26.83,-1.91]],
    # Level 4: 2^3P^0
    [4,9,20.964,0.87607e15,[61.21,-2.9]],
    # Level 5: 2^1P^0
    [5,3,21.217,0.81465e15,[81.35,-3.5]],
    # Level 6: 3^3S
    [6,3,22.718,0.4519e15,[12.69,-1.54]],
    # Level 7: 3^1S
    [7,1,22.920,0.4031e15,[23.85,-1.86]],
    # Level 8: 3^3P^0
    [8,9,23.006,0.3821e15,[49.30,-2.60]],
    # Level 9: 3^3D+3^1D
    [9,20,23.073,0.3659e15,[85.20,-3.69]],
    # Level 11: 3^1P^0
    [11, 3, 23.086, 0.3628e15, [58.81, -2.89]]]



    nu_val = nu.to_value(u.Hz) if hasattr(nu, 'unit') else nu
    T_val = T.to_value(u.K) if hasattr(T, 'unit') else T
    alpha = np.zeros_like(nu_val)
    ev_kB_cgs = (1*u.eV/c.k_B).cgs.value


    for i in range(0,len(he_level)):
        level = he_level[i][0]
        g_i = he_level[i][1]
        E_i_eV = he_level[i][2]
        nu_threshold = he_level[i][3]
        ln_a_i = he_level[i][4]

        mask = nu_val >= nu_threshold

        nu_masked = nu_val[mask]

        if len(ln_a_i) == 2:
            a = ln_a_i[0]
            b = ln_a_i[1]
            ln_a = a + b * np.log(nu_masked)
        else:
            a = ln_a_i[0]
```

```
            b = ln_a_i[1]
            c = ln_a_i[2]
            ln_a = a + (b+c * np.log(nu_masked))*np.log(nu_masked)

        # Note question for higher levels -> do I need to do n=> 4 differently

        cross_section = np.exp(ln_a)
        boltzmann_factor = g_i * np.exp(-E_i_eV * ev_kB_cgs / T_val)

        # total crossection
        alpha[mask] += cross_section * boltzmann_factor

    return alpha * (1-np.exp(-h_kB_cgs*nu_val/T_val))

def kappa_cont(nu, log10P, T):
    """Compute the continuum opacity in cgs units as a function of
    log pressure (CGS) and K.

    Parameters:
    nu: numpy array
    log10P: float
    T: float
    """
    T_val = T.to_value(u.K) if hasattr(T, 'unit') else T
    nHI = 10**(log10ns[0](log10P, T_val, grid=False))
    nHII = 10**(log10ns[1](log10P, T_val, grid=False))
    nHm = 10**(log10ns[2](log10P, T_val, grid=False))

    nHeI = 10**(log10ns[3](log10P, T_val, grid=False)) #He

    HeIff = Hff(nu, T_val)

    ne = 10**(log10ne(log10P, T_val, grid=False))
    kappa = nHI * Hbf(nu, T_val) + nHII * ne * Hff(nu, T_val) + \
        nHm * Hmbf(nu, T_val) + nHI * ne * Hmff(nu, T_val) + \
        nHeI*HeIbf(nu, T_val) + nHeI*HeIff*ne
    return kappa

# Initialize molecular line data (do this once)
csv_files = [
    '20251002055928/20251002055928__1H2-16O__144.0K.csv',
    '20251002055928/20251002055928__12C-1H4__144.0K.csv',
    '20251002055928/20251002055928__14N-1H3__144.0K.csv'
]
molecular_lines = MolecularLines(csv_files)

def kappa_cont_molecules(nu, log10P, T, molecule_abundances=None):
```

```
    """
    Enhanced continuum opacity including molecular lines
    """

    # Get existing continuum opacity
    kappa_continuum = kappa_cont(nu, log10P, T)

    # Add molecular lines if abundances provided
    if molecule_abundances is not None:
        P = 10**log10P
        kappa_molecular = molecular_lines.compute_molecular_opacity(
            nu, T, P, molecule_abundances
        )

        return kappa_continuum + kappa_molecular

    return kappa_continuum

def kappa_cont_H(nu, T, nHI, nHII, nHm, nHeI):
    """Compute the continuum opacity in cgs units as a function of
    temperature in K and number densities.
    """
    kappa = nHI * Hbf(nu, T) + nHII * ne * Hff(nu, T) + \
            nHm * Hmbf(nu, T) + nHI * ne * Hmff(nu, T) + \
            nHeI*HeIbf(nu, T) + nHeI*Hff(nu, T)*ne
    return kappa

if __name__=="__main__":
    #Lets compute a Rosseland mean opacity!
    #Create a grid of frequencies from 30 nm to 30 microns.
    dnu = 1e13
    plt.clf()
    nu = dnu*np.arange(1000) + dnu/2
    natoms = f_eos['ns [cm^-3]'].data.shape[2]//3
    kappa_bar_Planck = np.zeros_like(f_eos[0].data)
    kappa_bar_Ross = np.zeros_like(f_eos[0].data)
    for i, P_log10 in enumerate(Ps_log10):
        for j, T in enumerate(Ts):
            nHI = f_eos['ns [cm^-3]'].data[i,j,0]
            nHII = f_eos['ns [cm^-3]'].data[i,j,1]
            nHm = f_eos['ns [cm^-3]'].data[i,j,2]
            nHeI = f_eos['ns [cm^-3]'].data[i,j,3]
            ne = f_eos['n_e [cm^-3]'].data[i,j]
            # Compute the volume-weighted absorption coefficient, using Hydrogen
            kappa = kappa_cont_H(nu, T, nHI, nHII, nHm, ne)
            # Add a small floor to kappa to avoid division by zero
            kappa_safe = np.maximum(kappa, 1e-30)
```

```
            # Now compute the Rosseland and Planck means.
            Bnu = nu**3/(np.exp(h_kB_cgs*nu/T)-1)
            dBnu = nu**4 * np.exp(h_kB_cgs*nu/T)/(np.exp(h_kB_cgs*nu/T)-1)**2
            kappa_bar_Planck[i,j] = np.sum(kappa*Bnu)/np.sum(Bnu)/rho[i,j]
            kappa_bar_Ross[i,j] = 1/(np.sum(dBnu/kappa_safe)/np.sum(dBnu))/
↪rho[i,j]
            if (i==30): #This is log_10(P)=3.5 - similar to solar photosphere.
                if ((j < 18) & (j % 2 == 0)):
                    plt.loglog(3e8/nu, kappa/rho[i,j], label=f'T={T}K')
    # Safeguard: replace NaN or non-finite values with a small positive number
    kappa_bar_Ross = np.where(np.isfinite(kappa_bar_Ross), kappa_bar_Ross,␣
↪1e-30)
    kappa_bar_Planck = np.where(np.isfinite(kappa_bar_Planck),␣
↪kappa_bar_Planck, 1e-30)
    hdu1 = fits.PrimaryHDU(kappa_bar_Ross)
    hdu1.header['CRVAL1'] = Ts[0]
    hdu1.header['CDELT1'] = Ts[1]-Ts[0]
    hdu1.header['CTYPE1'] = 'Temperature [K]'
    hdu1.header['CRVAL2'] = Ps_log10[0]
    hdu1.header['CDELT2'] = Ps_log10[1]-Ps_log10[0]
    hdu1.header['CTYPE2'] = 'log10(pressure) [dyne/cm^2]'
    hdu1.header['EXTNAME'] = 'kappa_Ross [cm**2/g]'
    hdu2 = fits.ImageHDU(kappa_bar_Planck)
    hdu2.header['EXTNAME'] = 'kappa_Planck [cm**2/g]'
    hdulist = fits.HDUList([hdu1, hdu2])
    hdulist.writeto('Ross_Planck_opac.fits', overwrite=True)
    plt.legend()
    plt.xlabel('Wavelength [m]')
    plt.ylabel(r'$\\kappa_R$ [cm$^2$/g]')
```

```
/Users/griffinkatrivesisbrown/Library/Mobile
Documents/iCloud~md~obsidian/Documents/project-10/grey_model/eos.py:369:
RuntimeWarning: overflow encountered in scalar divide
  f_e = n_e/n_h
/Users/griffinkatrivesisbrown/Library/Mobile
Documents/iCloud~md~obsidian/Documents/project-10/grey_model/eos.py:395:
RuntimeWarning: divide by zero encountered in scalar divide
  P = rho/mu*(c.k_B*T*u.K/u.u).cgs.value
/Users/griffinkatrivesisbrown/Library/Mobile
Documents/iCloud~md~obsidian/Documents/project-10/grey_model/eos.py:396:
RuntimeWarning: divide by zero encountered in log
  return np.log(P_0_cgs/P)
/Users/griffinkatrivesisbrown/Library/Mobile
Documents/iCloud~md~obsidian/Documents/project-10/grey_model/eos.py:710:
RuntimeWarning: overflow encountered in scalar divide
  cP_tab[i,j] = ((Ui_plus - Ui)/dT).cgs.value +
5/2*k_b_u_cgs*Q_tab[i,j]/mu_tab[i,j]
```

```
/Users/griffinkatrivesisbrown/Library/Mobile
Documents/iCloud~md~obsidian/Documents/project-10/grey_model/eos.py:710:
RuntimeWarning: overflow encountered in scalar multiply
  cP_tab[i,j] = ((Ui_plus - Ui)/dT).cgs.value +
5/2*k_b_u_cgs*Q_tab[i,j]/mu_tab[i,j]
<ipython-input-1-25fd1d225cd8>:478: RuntimeWarning: overflow encountered in exp
  Bnu = nu**3/(np.exp(h_kB_cgs*nu/T)-1)
<ipython-input-1-25fd1d225cd8>:479: RuntimeWarning: overflow encountered in exp
  dBnu = nu**4 * np.exp(h_kB_cgs*nu/T)/(np.exp(h_kB_cgs*nu/T)-1)**2
<ipython-input-1-25fd1d225cd8>:479: RuntimeWarning: overflow encountered in
multiply
  dBnu = nu**4 * np.exp(h_kB_cgs*nu/T)/(np.exp(h_kB_cgs*nu/T)-1)**2
<ipython-input-1-25fd1d225cd8>:479: RuntimeWarning: overflow encountered in
square
  dBnu = nu**4 * np.exp(h_kB_cgs*nu/T)/(np.exp(h_kB_cgs*nu/T)-1)**2
<ipython-input-1-25fd1d225cd8>:479: RuntimeWarning: invalid value encountered in
divide
  dBnu = nu**4 * np.exp(h_kB_cgs*nu/T)/(np.exp(h_kB_cgs*nu/T)-1)**2
```