LTE Opacity

September 13, 2017

1 Tarea 4 - LTE opacities for a pure H atmosphere

We try to reproduce Figure 8.5 a, b, c, d from Gray, which show the wavelength-dependent continuous absorption coefficient κ_λ for different temperatures T and electron pressures P_e. For simplicity, we consider only hydrogen, in the form of the neutral atom and the positive and negative ions. Ion fractions and excitation of bound levels is calculated under the assumption of local thermodynamic equilibrium.

```
In [3]: import numpy as np
        import matplotlib.pyplot as plt
        %matplotlib inline
        import seaborn as sns
        sns.set(context='notebook',
                style='whitegrid',
                palette='dark',
                font_scale=1.5,
                color_codes=True,
                rc={'figure.figsize': (8,6)},
               )
```

1.1 Set up the constants we need

```
In [4]: import astropy.units as u
        from astropy.constants import k_B, h, m_p, a0, m_e
        from astropy.constants import c as light_speed
        from astropy.table import Table, Column
```

We work with all energies in electron volts. We define the constants as regular floats because it doesn't seem possible to use `astropy.units` quantities with `@np.vectorized` functions.

```
In [5]: BOLTZMANN = k_B.to(u.eV/u.K).value
        PLANCK = h.to(u.eV*u.s).value
        RYDBERG = (1.0*u.Ry).to(u.eV).value
        HMINUS_EION = (0.754*u.eV).value
        BOHR_RADIUS = a0.cgs.value

        print('BOLTZMANN =', BOLTZMANN, 'eV/K')
        print('PLANCK =', PLANCK, 'eV.s')
        print('RYDBERG =', RYDBERG, 'eV')
        print('HMINUS_EION =', HMINUS_EION, 'eV')
        print('BOHR_RADIUS =', BOHR_RADIUS, 'cm')
```

```
BOLTZMANN = 8.617330337217213e-05 eV/K
PLANCK = 4.1356676623401646e-15 eV.s
RYDBERG = 13.60569300965081 eV
HMINUS_EION = 0.754 eV
BOHR_RADIUS = 5.2917721067e-09 cm
```

1.2 Ionization balance of hydrogen

1.2.1 The general Saha equation

We use the following version of the Saha equation:

$$\frac{N_{j+1}N_e}{N_j} = \Phi_j(T),$$

which relates the densities of adjacent ionization stages j and $j+1$.

First define the T-dependent function $\Phi_j = 4.8293744908 \times 10^{15} \left(U_{j+1}/U_j\right) T^{1.5} e^{-E_j/kT}$, where E_j is the j-th ionization potential and U_j, U_{j+1} are the partition functions.

The temperature-dependent function on the RHS is

$$\Phi_j(T) = 2\left(\frac{2\pi m_e kT}{h^2}\right)^{1.5} \frac{U_{j+1}}{U_j} e^{-E_j/kT},$$

where E_j is the j-th ionization potential and U_j, U_{j+1} are the partition functions.

We evaluate the constant

```
In [6]: saha_C = 2*(2*np.pi*m_e*k_B / (h**2))**1.5
        saha_C.cgs
```

Out[6]:

$4.8293637 \times 10^{15}\ \frac{1}{\mathrm{K^{3/2}\,cm^3}}$

which is sometimes (e.g., Mihalas) given the other way up

```
In [7]: 1./saha_C.cgs
```

Out[7]:

$2.0706662 \times 10^{-16}\ \mathrm{cm^3\,K^{3/2}}$

and define a function to evaluate $\Phi_j(T; E_j, U_j, U_{j+1})$. Default values of $U_j = 2$, $U_{j+1} = 1$ are correct for the H^+/H^0 balance at low temperatures.

```
In [8]: SAHA_CONSTANT = saha_C.cgs.value


        def Saha_Phi(T, Eion=1.0*RYDBERG, Ui=1.0, Un=2.0):
            """
            Función Phi(T) = (Ni Ne / Nn) de Saha
            para energía de ionización Eion,
            y con funciones de partición Ui y Un
            """
            return SAHA_CONSTANT * (Ui/Un) * T**1.5 * np.exp(-Eion/(BOLTZMANN*T))
```

Test the function for some typical temperatures.

```
In [9]: Ts = np.array([3, 5, 9, 15])*u.kK
        Ts.cgs
```

Out[9]:

[3000, 5000, 9000, 15000] K

Note that the T argument should be a normal number (e.g, `float`) in units of Kelvin. In this example, we set up the temperature array in kilo-Kelvin, so we need to convert to cgs (or SI) and take the `value` before sending it to the function.

```
In [10]: Saha_Phi(Ts.cgs.value)
```

```
Out[10]: array([  5.52046633e-03,   1.64958730e+07,   4.95870573e+13,
                  1.19037446e+17])
```

1.2.2 The abundance of the positive hydrogen ion

We assume that the abundance of the negative ion H^- is always a negligible of total H, so that we have equal numbers of protons and free electrons: $N_+ = N_e$. Then the H positive ionization fraction, $y = N_+/N_H$, is the solution of the polynomial $y^2 + Ay - A = 0$, where $A = \Phi_{H_0}/N_H$.

We define a function `Hplus_fraction` that calculates y as a function of total hydrogen density and temperature. We use the `@np.vectorize` decorator so that we can apply the function to arrays of density and temperature. This is necessary here since `np.roots` solves only a single polynomial.

```
In [11]: @np.vectorize
         def Hplus_fraction(Hden, T):
             """
             Calcular fracción de hidrógeno ionizado

                     `Hden` es densidad de partículas totales de H en cm^{-3}
             `T` es temperatura en K
             """
             A = Saha_Phi(T) / Hden
             # Resolver polinomio: y**2 + A*y - A = 0
             y = np.roots([1.0, A, -A]).max() # tomar raiz positivo
             return y
```

For simplicity, we are here assuming that the H^0 partition function is equal to the statistical weight of the ground level: $g_1 = 2$. This is a good approximation at lowish temperatures, where the population of excited levels is negligible. We treat the more general case in the function `Hplus_fraction_U()` below.

1.2.3 The abundance of the negative hydrogen ion

The Saha equation for H is:

$$\frac{N_{H^0} N_e}{N_{H^-}} = \Phi_{H^-}(T),$$

from which it follows that

$$N_{H^-}/N_H = \left(N_{H^0}/N_H\right) N_e/\Phi_{H^-} = (1-y)yN_H/\Phi_{H^-}$$

```
In [12]: def Hminus_fraction(Hden, T):
             """
             Calcular fracción del ión negativo de hidrógeno
             """
             y = Hplus_fraction(Hden, T)
             return y * (1. - y) * Hden/Saha_Phi(T, Eion=HMINUS_EION, Ui=2.0, Un=1.0)
```

1.2.4 Table and graphs of the ion fractions

Define some typical atmospheric densities. Then, make a table of the ion fractions four these four densities and the four temperatures that we defined above.

```
In [13]: Ns = np.array([10, 3, 1, 0.5])*1e15/u.cm**3
         Ns
```

Out[13]:

$$[1 \times 10^{16}, 3 \times 10^{15}, 1 \times 10^{15}, 5 \times 10^{14}] \; \frac{1}{\mathrm{cm^3}}$$

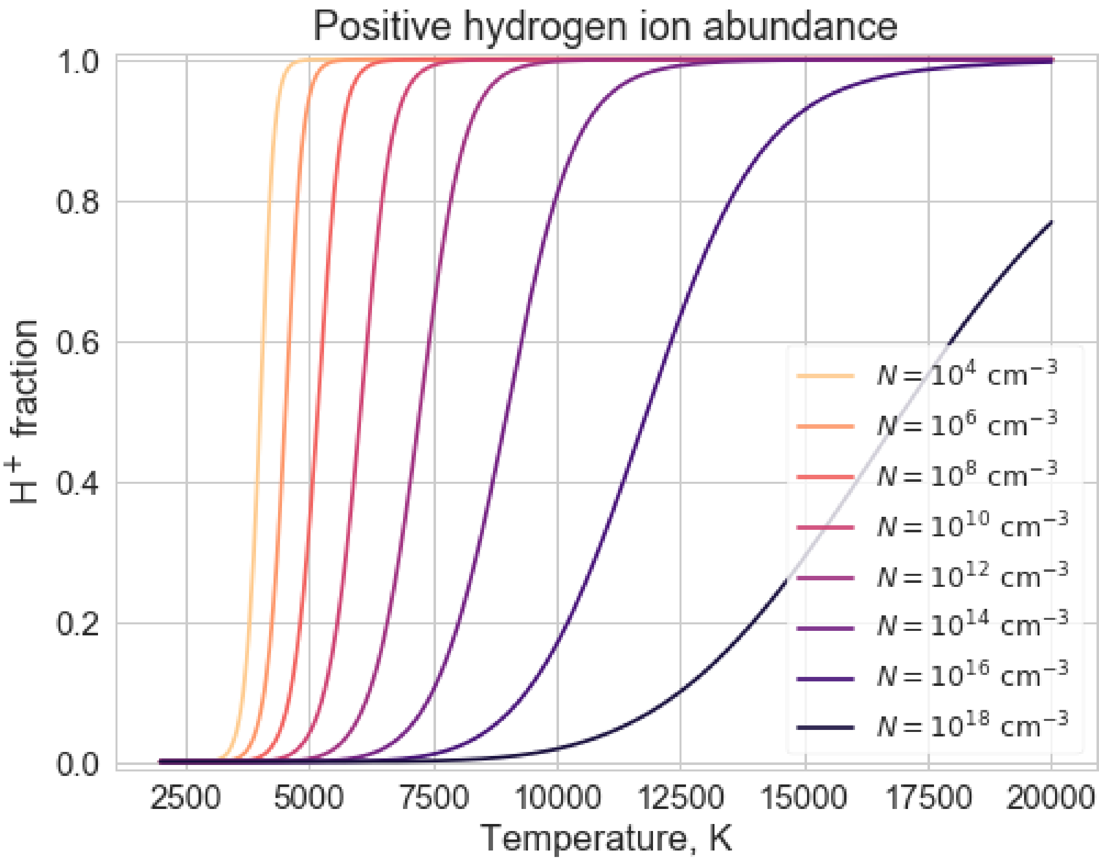
```
In [14]: Table(
           data=[
             Column(Ns.cgs, name=r'$N_H$'),
             Column(Ts.cgs, name=r'$T$'),
             Column(Hplus_fraction(Ns.cgs.value, Ts.cgs.value), name=r'$N_+/N_H$'),
             Column(Hminus_fraction(Ns.cgs.value, Ts.cgs.value), name=r'$N_-/N_H$'),
           ])
```

```
Out[14]: <Table length=4>
          $N_H$    $T$       $N_+/N_H$          $N_-/N_H$
         1 / cm3    K
         float64 float64      float64            float64
         ------- ------- ----------------- -----------------
           1e+16  3000.0 7.42998406735e-10 8.65069046435e-14
           3e+15  5000.0 7.41499601795e-05  3.7482095814e-10
           1e+15  9000.0    0.199264000514 5.11512306731e-08
           5e+14 15000.0    0.995834560847 2.09456640355e-10
```

Note that the H fraction is always very small, which justifies ignoring its effect on the electron density.

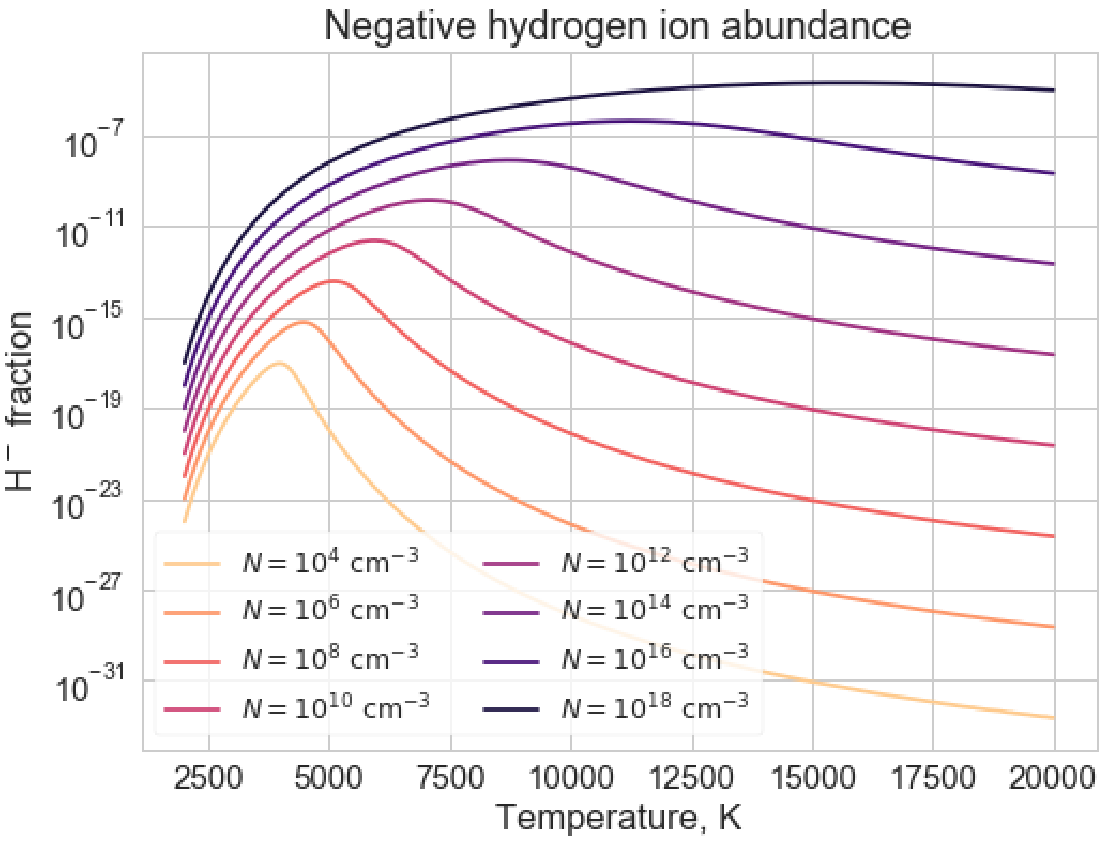
Next, we plot the ion fractions against temperature for a wide range of densities.

```
In [15]: logNgrid = range(4, 20, 2)
         Tgrid = np.linspace(2e3, 2e4, 500)
         fig, ax = plt.subplots(1, 1)
         legend_box_params = {
           'frameon': True,
           'fancybox': True,
           'fontsize': 'x-small',
         }
         colors = sns.color_palette('magma_r', n_colors=len(logNgrid))
         epsilon = 0.01
         for logN, c in zip(logNgrid, colors):
           ax.plot(Tgrid, Hplus_fraction(10**logN, Tgrid), color=c,
                   label=r'$N = 10^{{{}}}\ \mathrm{{cm}}^{{-3}}$'.format(logN))
         frame = ax.legend(loc='lower right', **legend_box_params).get_frame()
         frame.set_facecolor('white')
         ax.set_ylim(-epsilon, 1 + epsilon)
         ax.set_title('Positive hydrogen ion abundance')
         ax.set_xlabel('Temperature, K')
         ax.set_ylabel('H$^+$ fraction');
```



At the lower densities, hydrogen transitions from almost fully neutral to almost fully ionized over a narrow range of temperatures around 4000 K. But such low densities are only seen in the corona, where LTE does not apply. As the density is increased, higher temperatures are required and the curves shift to the right. For densities characteristic of stellar photospheres, the transition occurs around 7000 to 10,000 K.

```
In [16]: fig, ax = plt.subplots(1, 1)
         for logN, c in zip(logNgrid, colors):
           ax.semilogy(Tgrid, Hminus_fraction(10**logN, Tgrid), color=c,
                   label=r'$N = 10^{{{}}}\ \mathrm{{cm}}^{{-3}}$'.format(logN))
         frame = ax.legend(loc='lower left', ncol=2, **legend_box_params).get_frame()
         frame.set_facecolor('white')
         ax.set_title('Negative hydrogen ion abundance')
         ax.set_xlabel('Temperature, K')
         ax.set_ylabel('H$^-$ fraction');
```



The abundance of negative hydrogen ion is shown on a logarithmic scale. It generally increases with density, and it has a peak at the temperature where H is about 50% ionized, as can be seen by comparing this graph with the previous one.

1.3 Excitation of bound levels of H

We use the Boltzmann equation to calculate the fractional population of a given bound level, n, of neutral hydrogen.

$$\frac{N_n}{N_{H^0}} = \frac{g_n}{U(T)} e^{-E_n/kT}$$

where the degeneracy is $g_n = 2n^2$ and the energy in Rydbergs above the ground ($n = 1$) level is $E_n = 1 - n^{-2}$.

Here is the function to do that:

```
In [17]: def H0_level_population(n, T, U=2.0):
             """
             Calcular la población ETL del nivel n de hidrógeno neutro
             a una temperatura T kelvin
             """
             # Energía de excitación respeto a n=1
             E = RYDBERG * (1.0 - 1.0/n**2)
             # Peso estadístico
             g = 2.0*n**2
             return (g/U)*np.exp(-E/(BOLTZMANN*T))
```

[Extra credit: not required for tarea]

At low temperatures, the population of excited levels is negligible and we can take $U(T) \approx g_1 = 2$. But, in general we need to evaluate the partition function as

$$U(T) = \sum_{1}^{n_{\max}} g_n \, e^{-E_n/kT}$$

We can calculate this by re-using the `H0_level_population` function:

```
In [18]: @np.vectorize
         def H0_partition_function(T, nmax):
           U = np.zeros_like(T)
           for n in range(1, int(nmax)+1):
             U += H0_level_population(n, T, U=1.0)
           return U
```

We cannot take $n_{\max} \to \infty$ in this func, since the sum diverges. It is therefore important to find a physically motivated argument for determining the highest bound level, $n_{\max}$.

Taking account of the *pressure ionization* due to perturbations from neighboring particles, we make the approximation that in order that a level n should be bound, the radius of the level, r_n, must be less than the average distance between particles: $\sim (N_H)^{-1/3}$. Using $r_n = n^2 a_0$, where a_0 is the Bohr radius, this gives a maximum bound level $n_{\max} = a_0^{-1/2} N_H^{-1/6}$. See Hubeny & Mihalas, Chapter 4, p. 91 for more details.

```
In [19]: H0_partition_function(15000.0, 1000)

Out[19]: array(17918.801043584062)
```

```
In [20]: def nmax_pressure_ionization(Hden):
             """
             Calcular el nivel máximo ligado de H, sujeto a perturbaciones
             por vecinos con densidad `Hden`
             """
             return 1./np.sqrt(BOHR_RADIUS*Hden**(1./3.))
```

Now we use the above function to make a table of $n_{\max}$ for different densities. It is typicially ~ 100 for photospheric densities. At the higher densities found in stellar interiors ($N_H > 10^{21}$ cm^{-3}) even the $n = 1$ level becomes unbound and H is fully ionized at all temperatures.

```
In [21]: logNgrid_wide = range(4, 28, 2)
         Ns = (10**np.array(logNgrid_wide, dtype='float'))*u.cm**-3
         Table(data=[
           Column(Ns,
                 name=r'Hydrogen density, $N_H$', format='{:.0e}'),
           Column(nmax_pressure_ionization(Ns.value).astype(int),
                 name=r'Maximum bound level, $n_\mathrm{max}$')])
```

```
Out[21]: <Table length=12>
         Hydrogen density, $N_H$ Maximum bound level, $n_\mathrm{max}$
                 1 / cm3
                 float64                          int64
         ----------------------- -------------------------------------
                           1e+04                                  2961
                           1e+06                                  1374
                           1e+08                                   638
                           1e+10                                   296
                           1e+12                                   137
                           1e+14                                    63
                           1e+16                                    29
                           1e+18                                    13
                           1e+20                                     6
                           1e+22                                     2
                           1e+24                                     1
                           1e+26                                     0
```

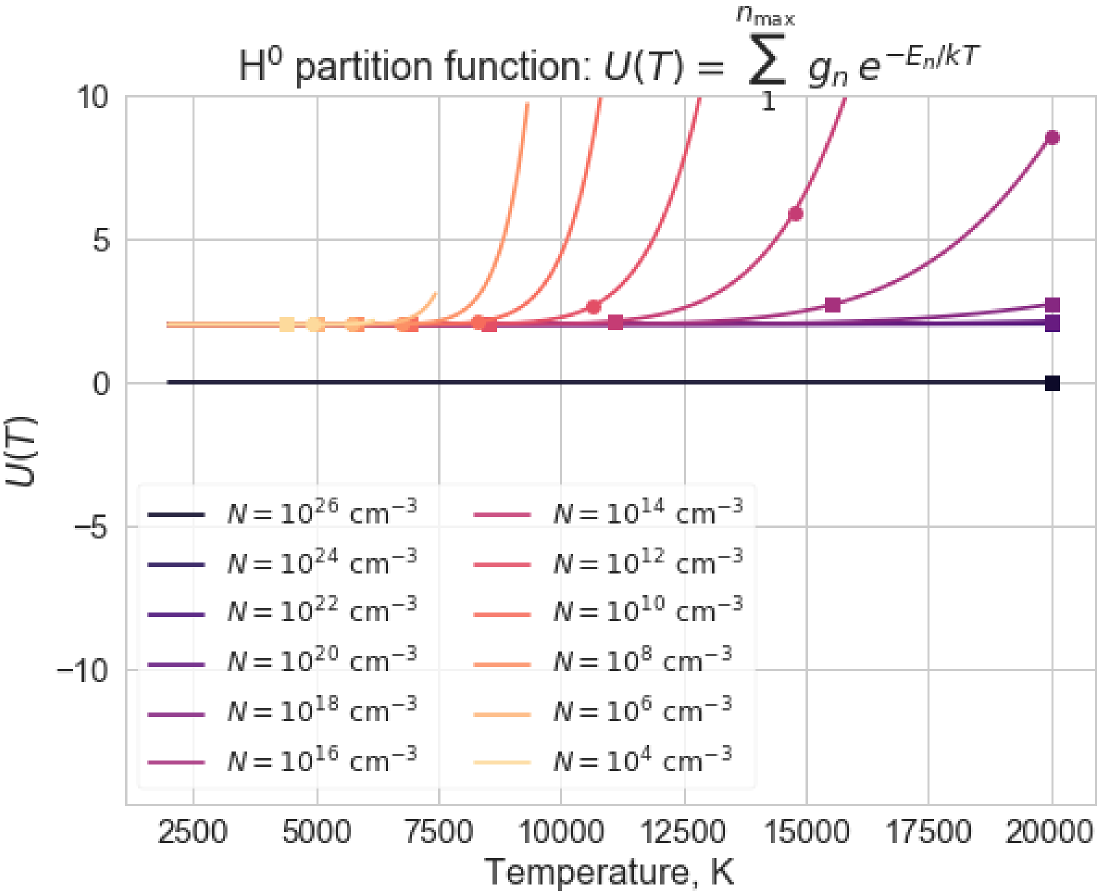
Finally, we can return to the partition function, plotting it against T using the $n_{\max}$ appropriate to different densities. For each density, the curves are only plotted for T where the neutral hydrogen fraction, $1 - y$, is larger than 10^{-6}. We also show with symbols the points where the ionization fraction is $y = 0.95$ (squares) and $y = 0.999$ (circles).

```
In [22]: np.zeros_like(0.0)
```

```
Out[22]: array(0.0)
```

```
In [23]: fig, ax = plt.subplots(1, 1)
         colors = sns.color_palette('magma_r', n_colors=len(Ns))
         for Hden, c in zip(Ns.value[::-1], colors[::-1]):
```

```
  nmax = int(nmax_pressure_ionization(Hden))
  Ugrid = H0_partition_function(Tgrid, nmax=nmax)
  mask = 1.0 - Hplus_fraction(Hden, Tgrid) > 1.e-6
  ax.plot(Tgrid[mask], Ugrid[mask], color=c,
         label=r'$N = 10^{{{}}}\ \mathrm{{cm}}^{{-3}}$'.format(int(np.log10(Hden))))
  for y, sym in [0.95, 's'], [0.999, 'o']:
    i0 = np.argmin(np.abs(Hplus_fraction(Hden, Tgrid) - y))
    ax.plot(Tgrid[i0], Ugrid[i0], sym, color=c)
ax.set_ylim(None, 10.)
frame = ax.legend(loc='lower left', ncol=2, **legend_box_params).get_frame()
frame.set_facecolor('white')
sigmatext = r'$U(T) = \sum_1^{n_\mathrm{max}}\ g_n\, e^{-E_n/k T}$'
ax.set_title('H$^0$ partition function: ' + sigmatext)
ax.set_xlabel('Temperature, K')
ax.set_ylabel(r'$U(T)$');
```



It can be seen that $U(T)$ only rises noticeably above 2 for densities above 10^8 cm^{-3}, and that it only becomes large when the hydrogen is nearly completely ionized ($y \gtrsim 0.999$). For the highest density of 10^{26} cm^{-3}, we have $n_\mathrm{max} = 0$, which means that there are no bound states at all, so $U(T) = 0$.

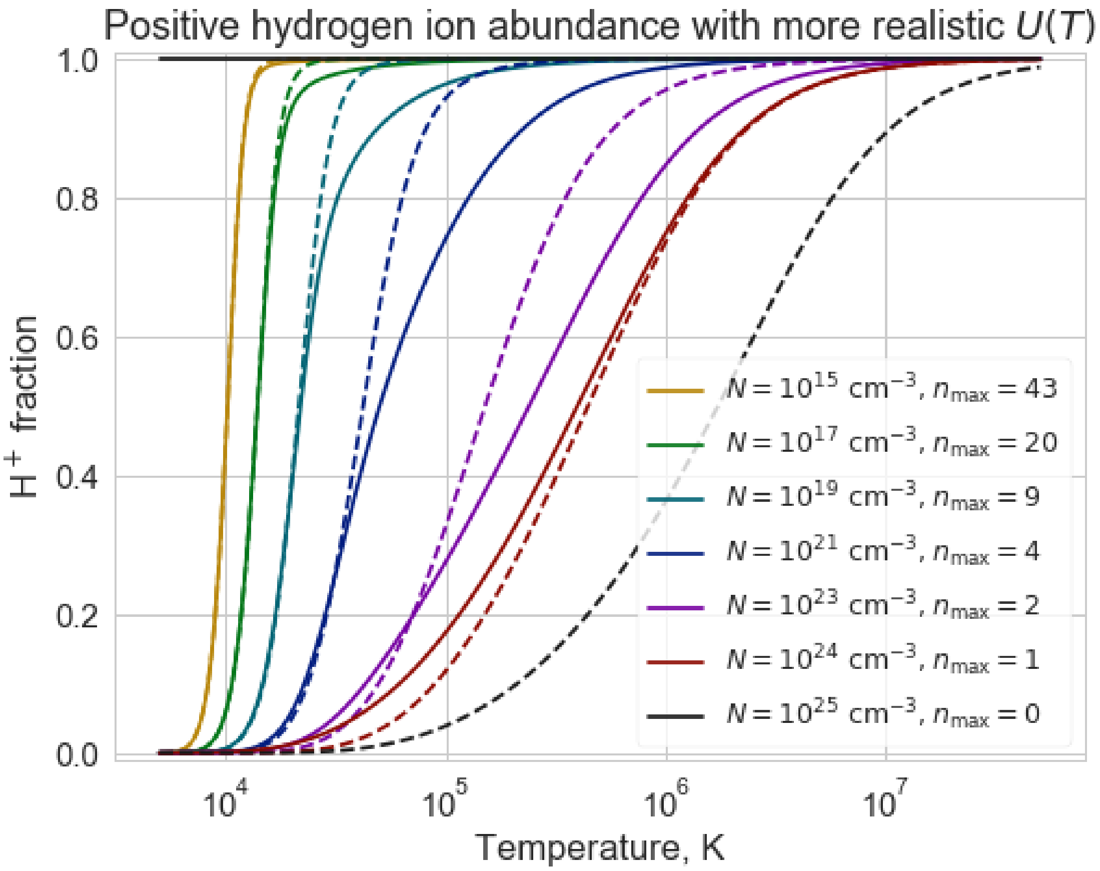
In the function Hplus_fraction above, we calculated the hydrogen ionization fraction under the approximation that $U(T) = 2$. We will now redo this function, but using the better approximation to $U(T)$ that we have just found.

For consistency, we should also incorporate the *continuum lowering* effect in the ionization balance. It can be included in a simple way by reducing the H ionization potential. However, once the ground level becomes unbound, then the approximations that we are using are no longer valid, so we should not expect this to be accurate for very large densities.

```
In [24]: @np.vectorize
         def Hplus_fraction_U(Hden, T):
             """
             Calcular fracción de hidrógeno ionizado con un U(T) más realista

                    `Hden` es densidad de partículas totales de H en cm^{-3}
             `T` es temperatura en K
             """
             nmax = nmax_pressure_ionization(Hden)
             if nmax < 1.0:
                 # pressure ionization
                 y = 1.0
             else:
                 U = HO_partition_function(T, nmax=int(nmax))
                 Ei = RYDBERG*(1.0 - 1.0/nmax**2)
                 A = Saha_Phi(T, Eion=Ei, Un=U) / Hden
                 # Resolver polinomio: y**2 + A*y - A = 0
                 y = np.roots([1.0, A, -A])[1] # tomar raiz positivo
             return y
```

Now we compare the two approximations. The constant-U version is shown as a dashed line and the new version as a solid line. We change to a logarithmic scale in temperature so we can see the effects of very large densities more clearly.

```
In [25]: Tgrid_wide = np.logspace(3.7, 7.7, 500)
         logNgrid_wide = list(range(15, 25, 2)) + [24, 25]
         #colors_wide = sns.color_palette('magma_r', n_colors=len(logNgrid_wide))
         colors_wide = 'ygcbmrk'
         fig, ax = plt.subplots(1, 1)
         for logN, c in zip(logNgrid_wide, colors_wide):
           Hden = 10**logN
           nmax = int(nmax_pressure_ionization(Hden))
           ax.plot(Tgrid_wide, Hplus_fraction_U(Hden, Tgrid_wide), color=c,
                   label=rf'$N = 10^{{{logN}}}\ \mathrm{{cm}}^{{-3}}, n_\mathrm{{max}} = {nmax
           ax.plot(Tgrid_wide, Hplus_fraction(Hden, Tgrid_wide), '--', color=c,
                   label=None)
         frame = ax.legend(loc='lower right', **legend_box_params).get_frame()
         frame.set_facecolor('white')
         ax.set_ylim(-epsilon, 1 + epsilon)
         ax.set_xscale('log')
         ax.set_title('Positive hydrogen ion abundance with more realistic $U(T)$')
         ax.set_xlabel('Temperature, K')
         ax.set_ylabel('H$^+$ fraction');
```



At densities $\leq 10^{15}$ cm^{-3}, there is almost no effect at all. At moderate densities of 10^{17} to 10^{21} cm^{-3} the prinicipal effect is to increase the neutral fraction at temperatures where H is nearly fully ionized. This is due to U increasing, which favors the neutral atom. But, for the very highest densities $> 10^{21}$ cm^{-3}, the continuum lowering starts to dominate and the partial ionization extends to lower temperatures due to reduction in the ionization potential. This also tends to reduce U again, since there *are* no excited levels to populate. Finally, for $N > 10^{24}$ cm^{-3}, even the ground level is unbound, so ionization is complete at all temperatures.

1.4 Wavelength-dependent cross sections

1.4.1 Neutral hydrogen H

Bound-free photoionization cross sections For photoionization from level n, there is a threshold energy, $E_n = n^{-2}$ Ry, with a corresponding minimum frequency, $\nu_n = E_n/h$, or maximum wavelength, $\lambda_n = hc/E_n$. The cross section is given by

$$\sigma_{\rm bf}(n,\nu) = \sigma_0 n \frac{\nu_n^3}{\nu^3} g_{\rm bf}(n,\nu)$$

where $\sigma_0 = 2.815 \times 10^{29} \nu_1^{-3} = 7.906 \times 10^{-18}$ cm^2 and $g_{\mathrm{bf}}(n, \nu)$ is the Gaunt factor that corrects for quantum mechanical effects.

```
In [26]: @np.vectorize
         def xsec_H0_boundfree(n, nu, xsec0=7.906e-18):
             """
             Sección eficaz de fotoionización de nivel n de H0 a frecuencia nu Hz

             Multiplicar por densidad de H0(n) para dar coeficiente de absorción (cm^{-1})
             """
             E = PLANCK*nu               # energía de fotón
             E0 = RYDBERG/n**2           # energía de ionización de nivel n

             if E >= E0:
                 xsec = gaunt_H0_boundfree(n, nu)*xsec0*n*(E0/E)**3
             else:
                 xsec = 0.0

             return xsec
```

For the gaunt factor we use the Menzel & Perkis approximation given in Gray's Eq (8.5):

$$g_{\mathrm{bf}}(n, \nu) = 1 - \frac{0.3456}{(\lambda R)^{1/3}} \left(\frac{\lambda R}{n^2} - \frac{1}{2} \right).$$

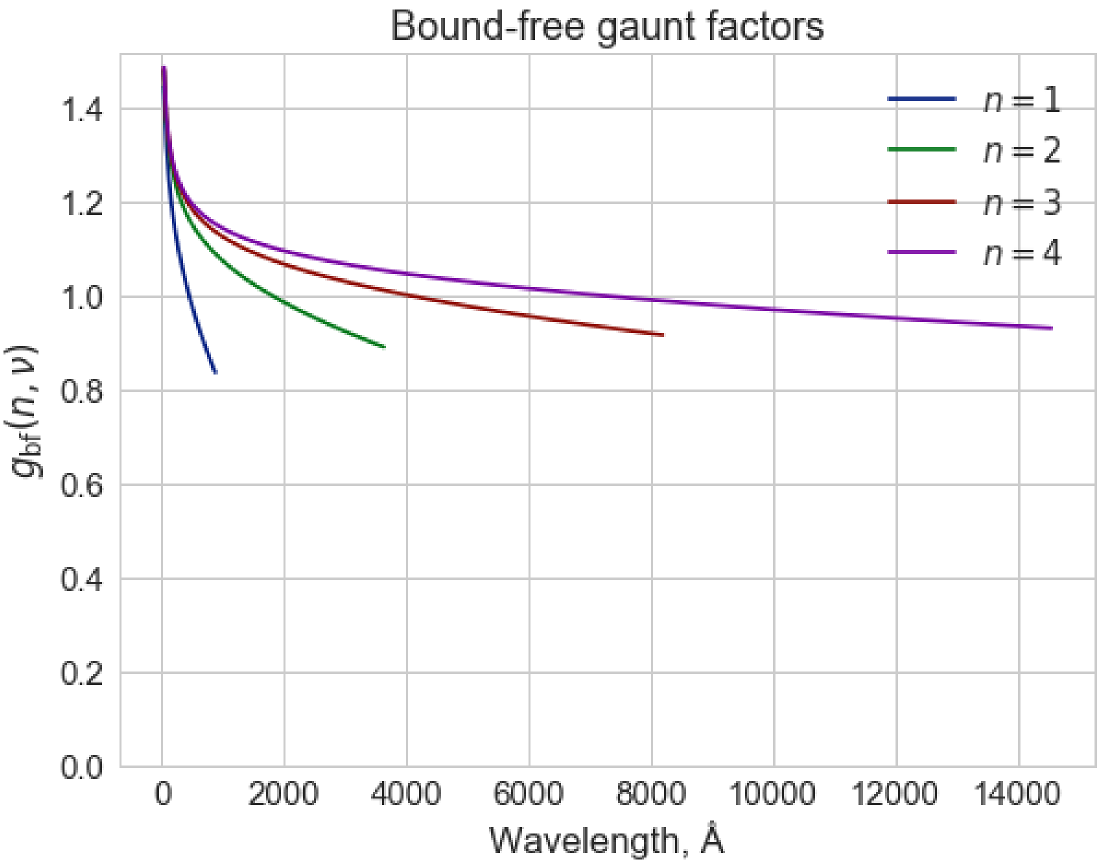
```
In [27]: def gaunt_H0_boundfree(n, nu):
           """
           Factor Gaunt para fotoionización de nivel `n` de H0 a frecuencia `nu` Hz
           """
           lambda_R = RYDBERG/(PLANCK*nu)
           return 1.0 - 0.3456*(lambda_R/n**2 - 0.5)/lambda_R**(1./3.)
```

Define an array of wavelengths for plotting and calculate the corresponding frequencies.

```
In [28]: wavs = np.linspace(40.0, 20000.0, 500)*u.AA
         freqs = (light_speed/wavs).cgs
         freqs[[0, -1]]
```

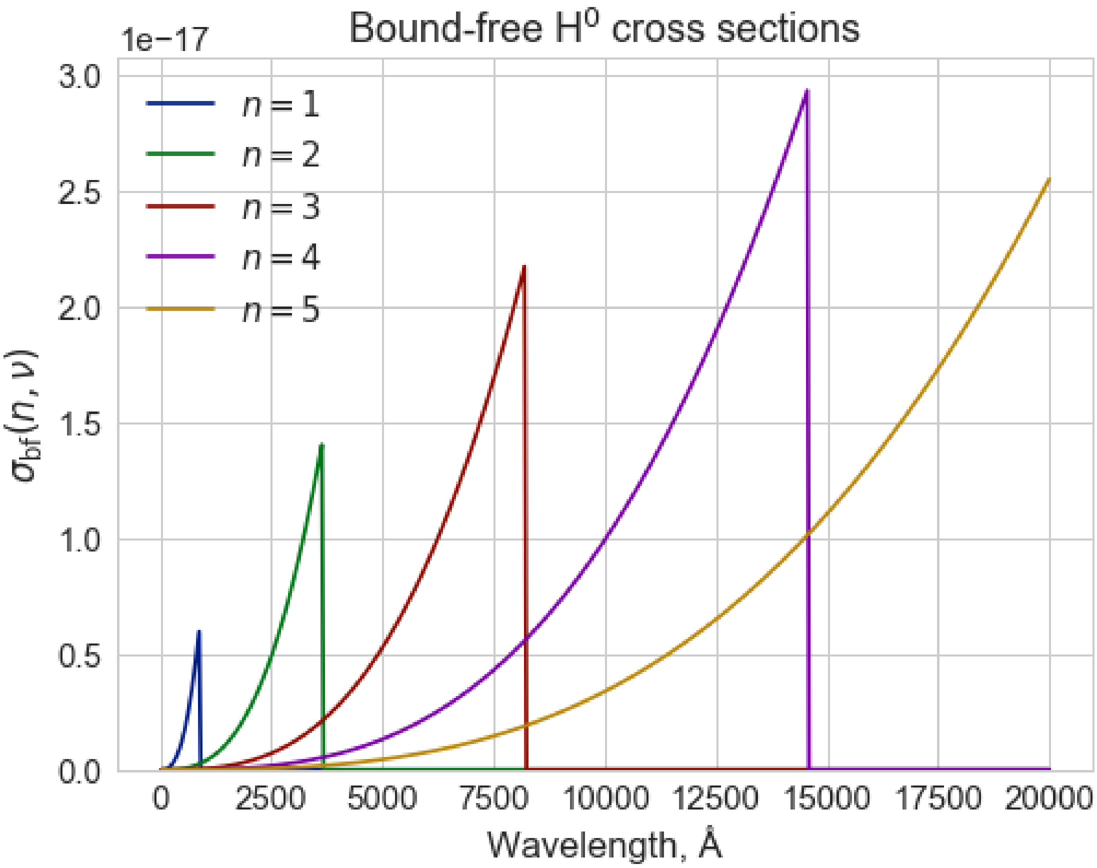
Out[28]:
$[7.4948114 \times 10^{16}, 1.4989623 \times 10^{14}] \ \frac{1}{\mathrm{s}}$

```
In [29]: fig, ax = plt.subplots(1, 1)
         for n in range(1, 5):
           m = h*freqs >= 1.0*u.Ry/n**2
           ax.plot(wavs[m], gaunt_H0_boundfree(n, freqs.value[m]),
                  label=r'$n = {}$'.format(n))
         ax.set_ylim(0.0, None)
         ax.legend()
         ax.set_xlabel(r'Wavelength, Å')
         ax.set_ylabel(r'$g_\mathrm{bf}(n, \nu)$')
         ax.set_title('Bound-free gaunt factors');
```



The gaunt factors are of order unity, tending to increase slightly in the ultraviolet. For each n, it only makes sense to plot them for $\lambda < \lambda_n$.

```
In [30]: fig, ax = plt.subplots(1, 1)
         for n in range(1, 6):
           ax.plot(wavs, xsec_H0_boundfree(n, freqs.value),
                  label=r'$n = {}$'.format(n))
         ax.set_ylim(0.0, None)
         ax.legend(loc='upper left')
         ax.set_xlabel(r'Wavelength, Å')
         ax.set_ylabel(r'$\sigma_\mathrm{bf}(n, \nu)$')
         ax.set_title('Bound-free H$^0$ cross sections');
```



The cross sections can be compared with Gray's Fig 8.2.

1.4.2 Free-free H cross-sections

The cross section per electron (see Rybicki, section 5.3) can be written as

$$\alpha_{\rm ff} = \alpha_0 \frac{g_{\rm ff}(T, \nu)}{\nu^3 T^{1/2}} \quad {\rm cm}^2 \,/\, {\rm e}^-,$$

where

$$\alpha_0 = \frac{4e^6}{3mhc}\left(\frac{2\pi}{3km}\right)^{1/2},$$

and the free-free Gaunt factor can be approximated (Gray, Eq. 8.6) as

$$g_{\rm ff}(T, \nu) = 1 - \frac{0.3456}{(\lambda R)^{1/3}}\left(\frac{kT}{h\nu} + \frac{1}{2}\right).$$

We calculate the numerical value of the constant, α_0:

```
In [31]: from astropy.constants import e, m_e
         alpha0 = np.sqrt(2*np.pi/(3*k_B*m_e))*(4*e.esu**6)/(3*m_e*h*light_speed)
         alpha0.cgs
```

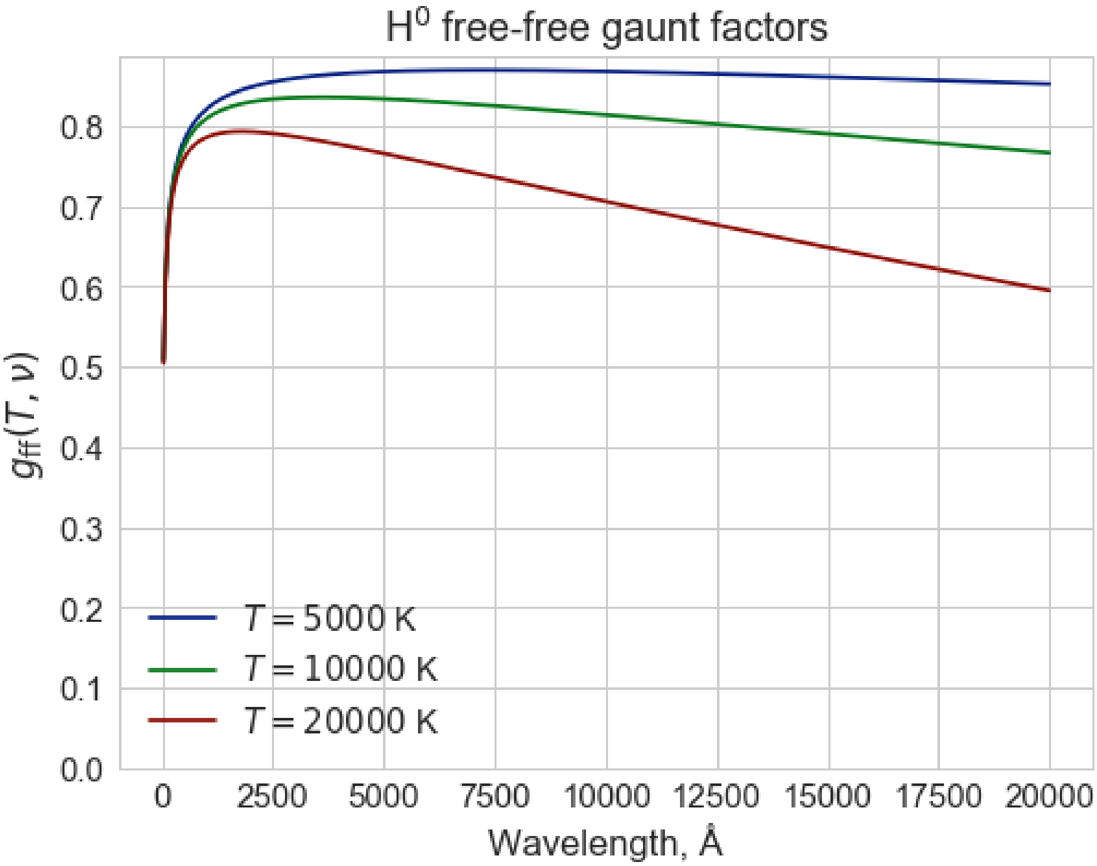
Out[31]:

$3.6923497 \times 10^{8} \; \frac{\mathrm{cm^{5}\,K^{1/2}}}{\mathrm{s^{3}}}$

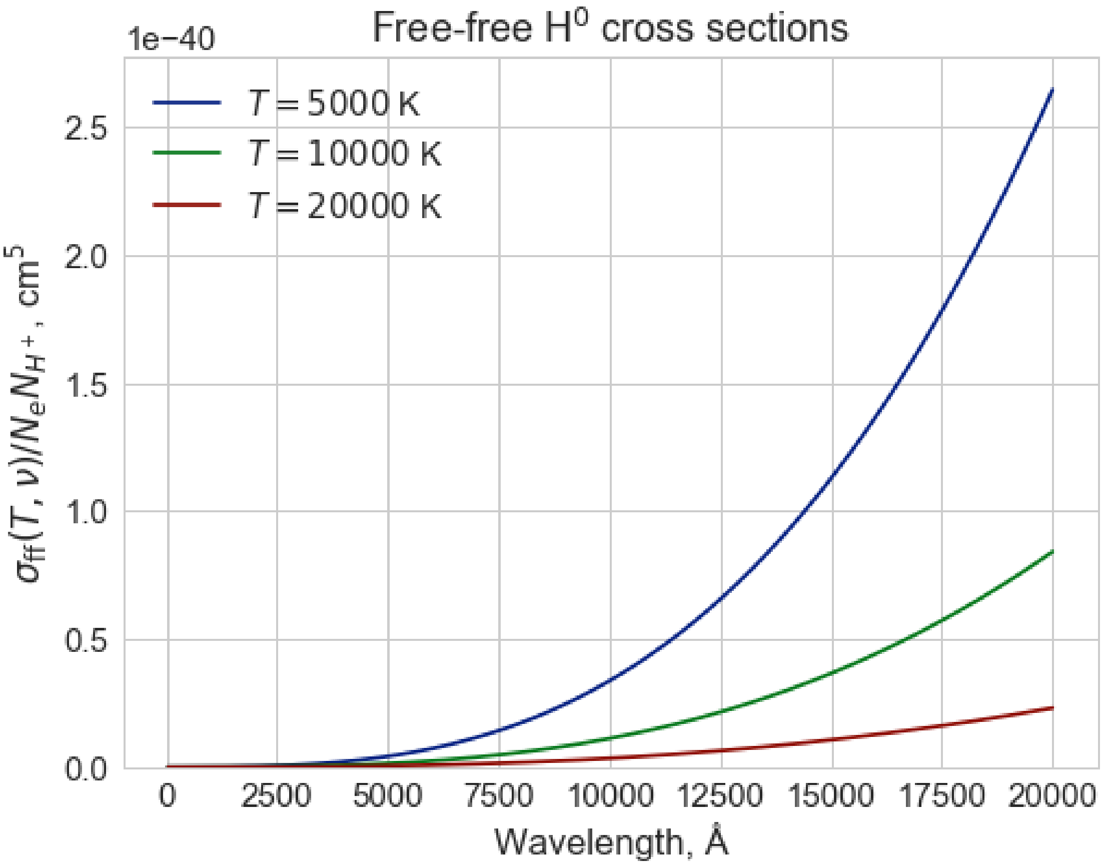
```
In [32]: def xsec_H0_freefree(T, nu):
             """
             Sección eficaz por electrón de bremsstrahlung a frecuencia nu Hz

             Multiplicar por Ne N(H+) para dar coeficiente de absorción (cm^{-1})
             """
             # cf. Rybicki, eq. 5.18b, but we omit the (1 - exp(-h nu/k T)) term
             # since we will apply it later
             return alpha0.cgs.value * gaunt_H0_freefree(T, nu) * T**-1.5 / nu**3
```

```
In [33]: def gaunt_H0_freefree(T, nu):
           """
           Factor Gaunt para absorción libre-libre H0 a frecuencia `nu` Hz
           """
           lambda_R = RYDBERG/(PLANCK*nu)
           return 1.0 - 0.3456*(BOLTZMANN*T/(PLANCK*nu) + 0.5)/lambda_R**(1./3.)
```

```
In [34]: fig, ax = plt.subplots(1, 1)
         for T in [5e3, 1e4, 2e4]:
           ax.plot(wavs, gaunt_H0_freefree(T, freqs.value),
                  label=r'$T = {:.0f}$ K'.format(T))
         ax.set_ylim(0.0, None)
         ax.legend(loc='lower left')
         ax.set_xlabel(r'Wavelength, Å')
         ax.set_ylabel(r'$g_\mathrm{ff}(T, \nu)$')
         ax.set_title('H$^0$ free-free gaunt factors');
```



```
In [35]: fig, ax = plt.subplots(1, 1)
         for T in [5e3, 1e4, 2e4]:
           ax.plot(wavs, xsec_H0_freefree(T, freqs.value),
                  label=r'$T = {:.0f}$ K'.format(T))
         ax.set_ylim(0.0, None)
         ax.legend(loc='upper left')
         ax.set_xlabel(r'Wavelength, Å')
         ax.set_ylabel(r'$\sigma_\mathrm{ff}(T, \nu) / N_e N_{H^+}$, cm$^5$')
         ax.set_title('Free-free H$^0$ cross sections');
```



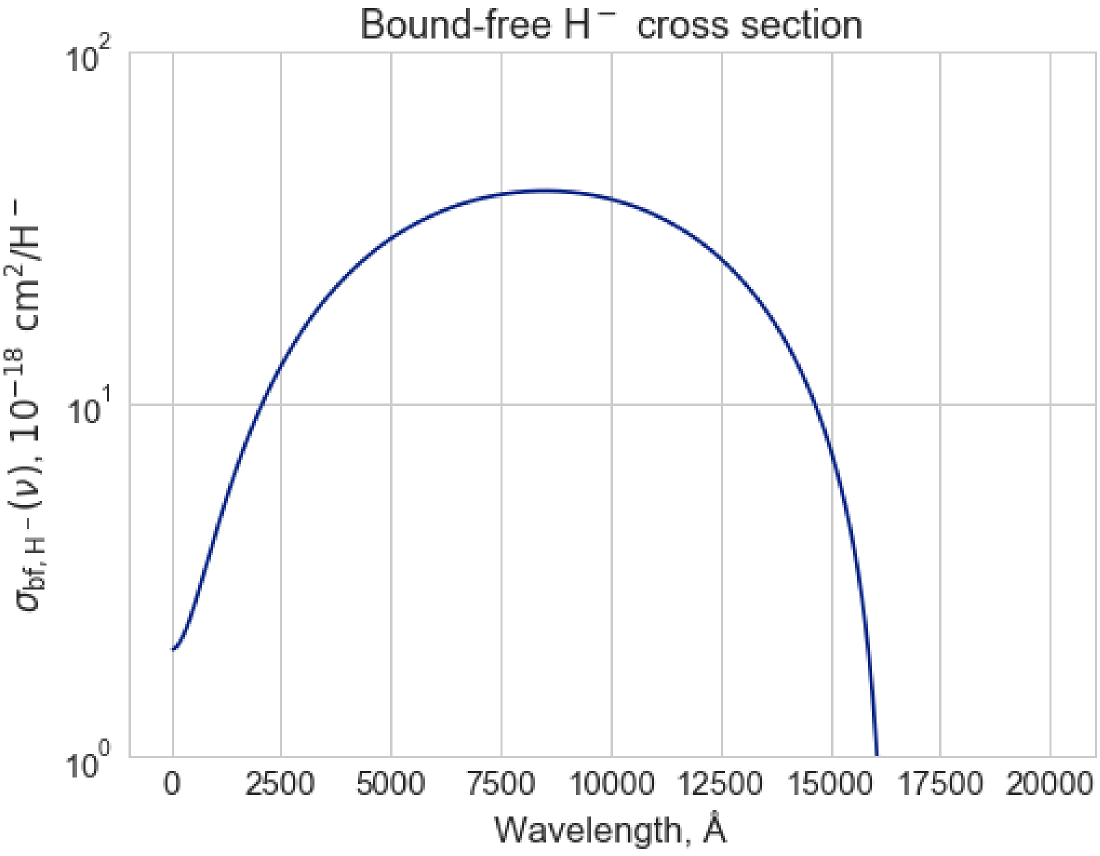
1.4.3 Negative hydrogen ion H

Bound-free H cross section We use the polynomial fit from Gray, which is stated to be accurate in the range 2250 $< \lambda <$ 15,000 . This gives the cross section in cm^2, so it needs to be multiplied by N_{H^-}.

```
In [36]: @np.vectorize
         def xsec_Hminus_boundfree(nu):
             """
             Sección eficaz de fotoionización del ión negativo H- a frecuencia nu Hz

             Multiplicar por N(H-) para dar coeficiente de absorción (cm^{-1})
             """
             # convertir nu a lambda en unidades de micras (10,000 Å)
             wav = (light_speed / (nu * u.Hz)).to(u.micron).value
             # Fórmula y constantes de Gray, Eq. 8.11
             A = [1.99654, -1.18267e-1, 2.64243e2,
                  -4.40524e2, 3.23992e2, -1.39568e2, 2.78701e1]
             xsec = 0.0
             # El ajuste es preciso para 2250 Å <= lambda <= 15,000 Å
```

```
    # Hay que cortarlo a partir de 16,200 Å porque el ajuste va negativo
    for i, a in enumerate(A):
        if wav <= 1.62:
            xsec += a*wav**i
    return xsec * 1.e-18
```

```
In [37]: fig, ax = plt.subplots(1, 1)
         ax.plot(wavs, xsec_Hminus_boundfree(freqs.value)/1e-18,
                  label=r'bf')
         ax.set_ylim(0.0, 5e-17)
         #ax.legend(loc='lower center')
         ax.set_yscale('log')
         ax.set_ylim(1.0, 100.0)
         ax.set_xlabel(r'Wavelength, Å')
         ax.set_ylabel(r'$\sigma_\mathrm{bf,H^-}(\nu)$, $10^{-18}\ \mathrm{cm}^2 / \mathrm{H}^
         ax.set_title('Bound-free H$^-$ cross section');
```



The graph above bears a reasonable resemblance to Gray's Fig. 8.3

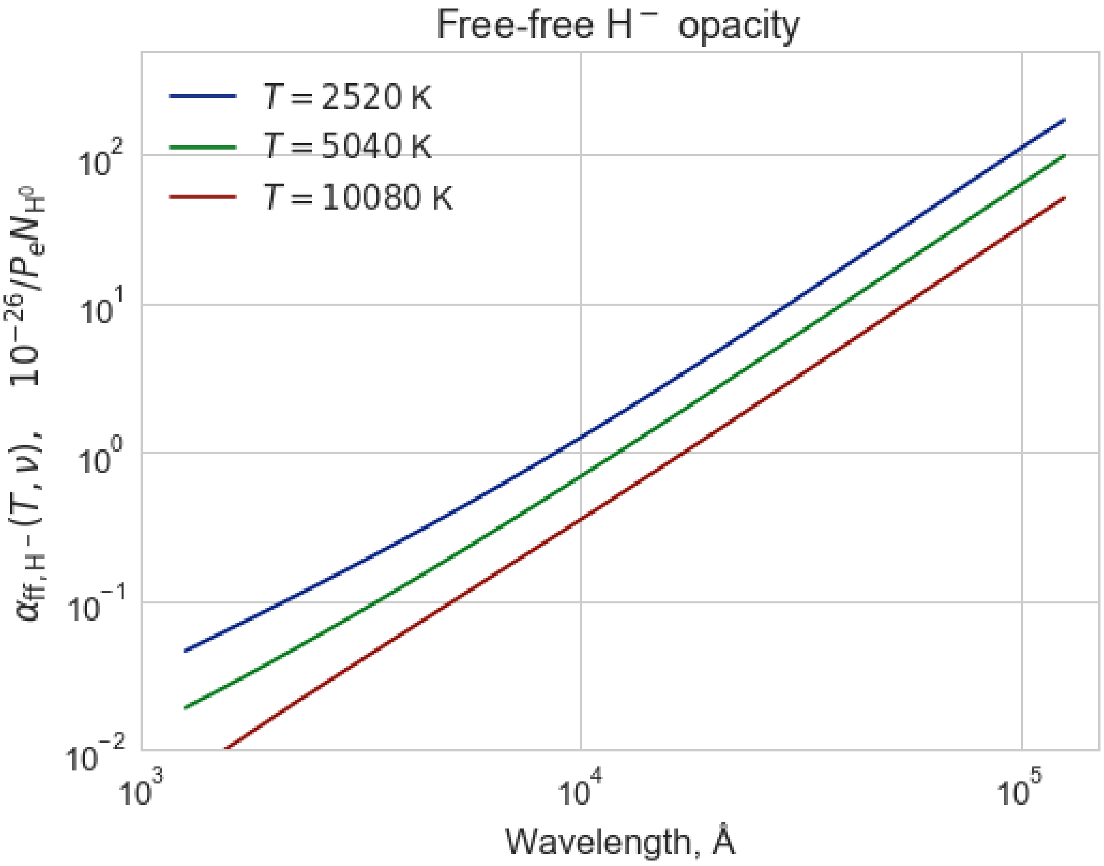
Free-free H opacity This is also calculated from polynomial fits given by Gray, which accurately reproduce the results of Bell & Berrington (1987) for the range 1823 $< \lambda <$ 151,890 and 1400 K $<$ $T <$ 10,080 K.

```
In [38]: def Hz_to_AA(nu):
           """
           Utility function to translate frequency to wavelength
           """
           return (light_speed / (nu / u.s)).to(u.AA).value

         @np.vectorize
         def xsec_Hminus_freefree(T, nu):
             """
             Opacidad libre-libre del ión negativo H- a frecuencia nu Hz

             Multiplicar por Pe N(H0) para dar coeficiente de absorción (cm^{-1})
             + Ojo que no hay que multiplicar por N(H-)
             + Y esto ya incluye la correción por emisión estimulada
             """
             # convertir nu a lambda en unidades de Å
             wav = Hz_to_AA(nu)
             logwav = np.log10(wav)
             # Eq. 8.13 de Gray
             f0 = -2.2763 - 1.6850*logwav + 0.76661*logwav**2 - 0.053346*logwav**3
             f1 = 15.2827 - 9.2846*logwav + 1.99381*logwav**2 - 0.142631*logwav**3
             f2 = (-197.789 + 190.266*logwav
                   - 67.9775*logwav**2 + 10.6913*logwav**3 - 0.625151*logwav**4)
             theta = np.log10(np.e) / (BOLTZMANN*T) # aproximadamente theta = 5040/T
             xsec = 1.e-26 * 10**(f0 + f1*np.log10(theta) + f2*np.log10(theta)**2)
             return xsec
```

The free free opacity is more important at longer wavelengths, so we define an extended range of wavelengths for plotting, up to just over 10 *m*.

```
In [39]: wavs_extend = np.logspace(3.1, 5.1, 500)*u.AA
         freqs_extend = (light_speed/wavs_extend).cgs
```

```
In [40]: fig, ax = plt.subplots(1, 1)
         for T in [2520.0, 5040.0, 10080.0]:
           ax.plot(wavs_extend, xsec_Hminus_freefree(T, freqs_extend.value)/1e-26,
                  label=r'$T = {:.0f}$ K'.format(T))
         ax.plot()
         ax.set_ylim(0.01, 500)
         ax.set_xlim(1000.0, 150000.0)
         ax.legend(loc='upper left')
         ax.set_xscale('log')
         ax.set_yscale('log')
         ax.set_xlabel(r'Wavelength, Å')
         ax.set_ylabel(r'$\alpha_\mathrm{ff,H^-}(T, \nu),\quad 10^{-26}/ P_e N_\mathrm{H^0}$ '
         ax.set_title('Free-free H$^-$ opacity');
```



This graph closely resemble Gray's Fig. 8.4. Note that the fits already include the correction for stimulated emission and are per neutral H atom and per unit electron pressure.

1.5 Finding the total hydrogen density in terms of electron pressure

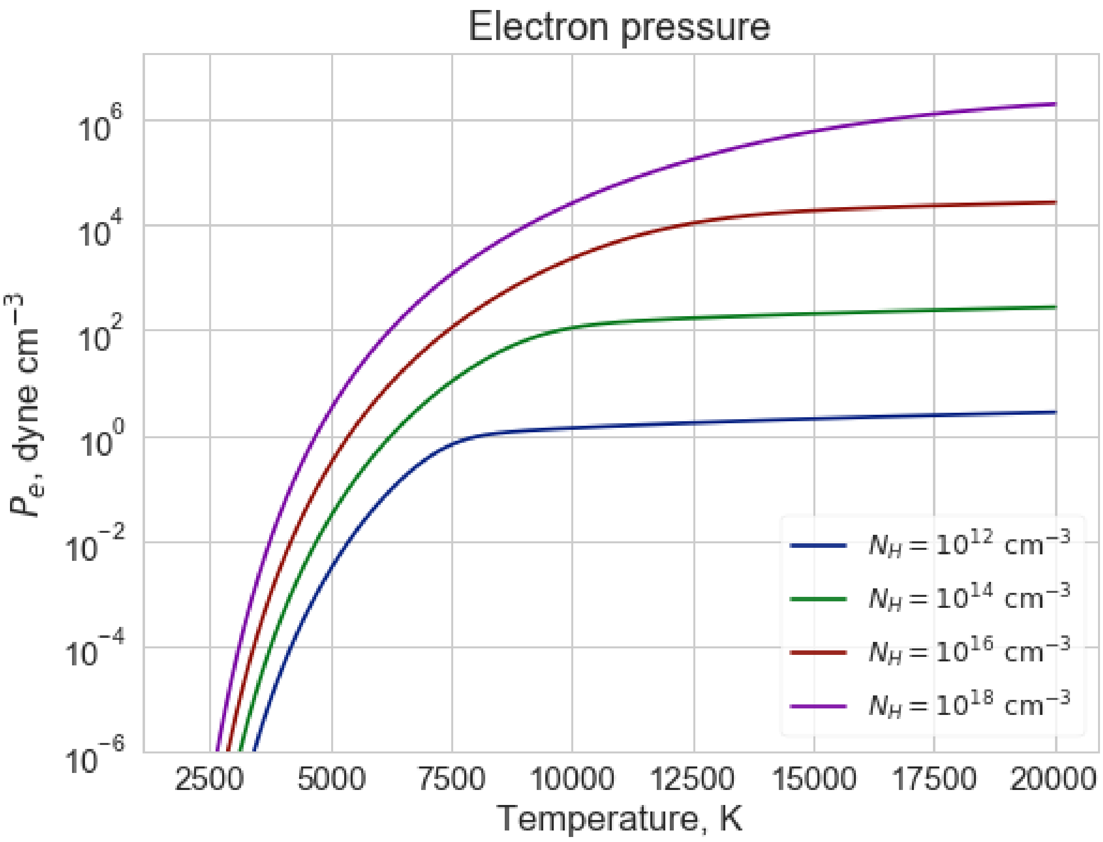
The graphs we are trying to reproduce are for fixed values of T and P_e, but most of our equations are in terms of densities, so need functions to convert between the two. Going from hydrogen density to electron pressure is straightforward:

```
In [41]: def funcPe(Hden, T):
             """
             Presión electrónica como función de densidad total y temperatura
             """
             return Hden*Hplus_fraction_U(Hden, T)*k_B.cgs.value*T
```

At high temperatures, ionization is complete and P_e increases linearly with T, which looks quite flat on the following graph because of the logarithmic scale on the y axis. At lower temperatures the ionization fraction falls, and so P_e drops steeply.

```
In [42]: fig, ax = plt.subplots(1, 1)
         for Hden in [1e12, 1e14, 1e16, 1e18]:
```

```
  ax.plot(Tgrid, funcPe(Hden, Tgrid),
          label=r'$N_H = 10^{{{:.0f}}}\ \mathrm{{cm^{{-3}}}}$'.format(np.log10(Hden))
frame = ax.legend(loc='lower right', **legend_box_params).get_frame()
frame.set_facecolor('white')
ax.set_yscale('log')
ax.set_ylim(1e-6, None)
ax.set_title('Electron pressure')
ax.set_xlabel('Temperature, K')
ax.set_ylabel('$P_e$, dyne cm$^{-3}$');
```



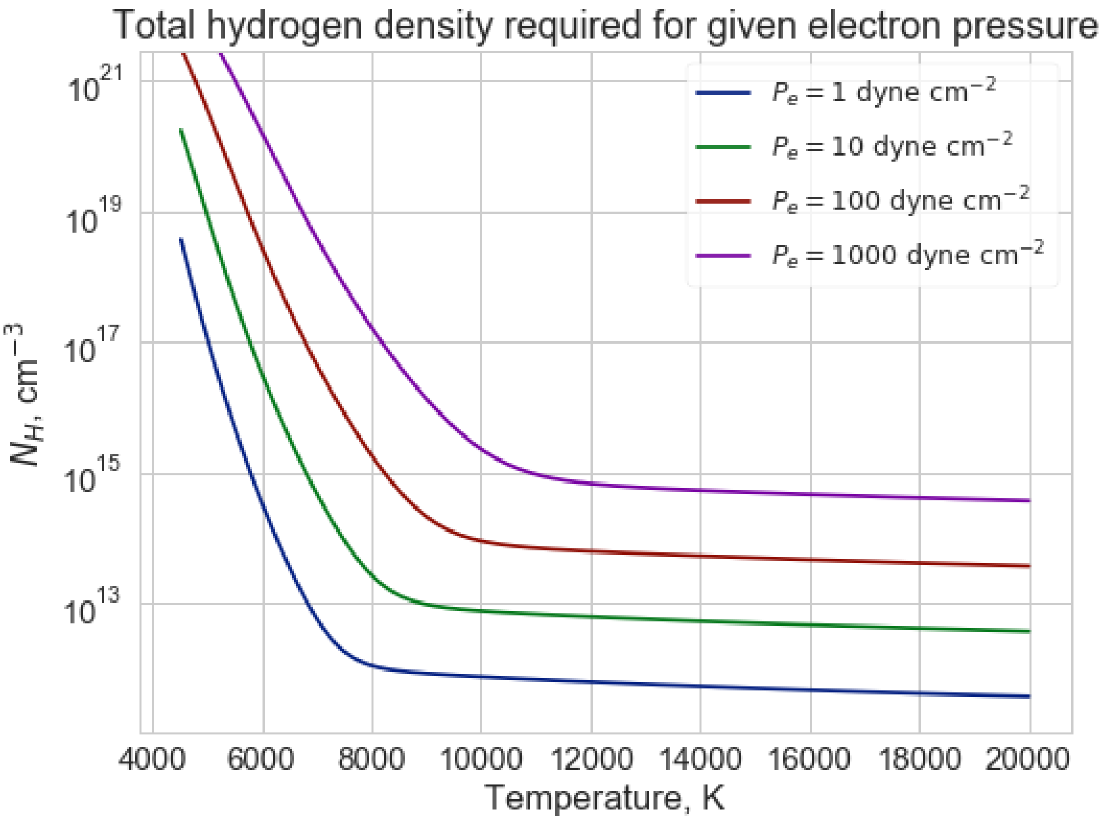
Going in the other direction requires solving an implicit equation:

```
In [43]: @np.vectorize
         def funcHden(Pe, T):
             """
             Densidad total como función de Pe y T

             Esta función busca numericamente el raiz para Hden de la función

             funcPe(Hden, T) - Pe = 0
```

```
    empezando con un primer intento que suponga 50% ionización
    """
    from scipy.optimize import fsolve
    Hden0 = 0.5*Pe / (k_B.cgs.value*T) # primer intento es 50% ionizado
    return fsolve(lambda Hden: funcPe(Hden, T) - Pe, Hden0)[0]
```

We now test this function by making a graph of total hydrogen density for electron pressures $P_e = 1 \rightarrow 1000$ dyne cm^2 and temperatures $T = 4500 \rightarrow 20,000$ K. We can't go to much lower temperatures because the electron fraction becomes so low that it is impossible to find a reasonable solution for the higher values of P_e.

```
In [44]: fig, ax = plt.subplots(1, 1)
         for Pe in [1.0, 10., 100., 1000.]:
           m = Tgrid >= 4500.0
           ax.plot(Tgrid[m], funcHden(Pe, Tgrid[m]),
                   label=r'$P_e = {:.0f}\ \mathrm{{dyne\ cm^{{-2}}}}$'.format(Pe))
         frame = ax.legend(loc='upper right', **legend_box_params).get_frame()
         frame.set_facecolor('white')
         ax.set_yscale('log')
         ax.set_ylim(None, 3e21)
         ax.set_title('Total hydrogen density required for given electron pressure')
         ax.set_xlabel('Temperature, K')
         ax.set_ylabel('$N_{H}$, cm$^{-3}$');
```



1.6 Total wavelength-dependent opacities

```
In [45]: def opacidad_total(Pe, T, wavs):
             """
             Calcular la opacidad total del continuo de un gas de H puro en ETL

             Parámetros de entrada:

             Pe   : presión de electrones (dyne cm^{-2})
             T    : temperatura (K)
             wavs : longitud de onda (Å)

             Resultado:

             opacities: dict con coeficiente de absorción por masa (cm^2/g)
                        elementos son "Total", "H0bf", "H0ff", "Hmbf", "Hmff"
             """

             Hden = funcHden(Pe, T)                # densidad total de H
             y = Hplus_fraction_U(Hden, T)           # fracción de ionización
             Hpden = y*Hden                        # densidad de H+
             eden = y*Hden                         # densidad de electrones
             H0den = (1.0 - y)*Hden                # densidad de H0
             Hmden = Hden*Hminus_fraction(Hden, T) # densidad de H-

             # frequencies are pure numbers in Hz
             nu = (light_speed/(wavs*u.AA)).cgs.value
             stimulated_correction = (1.0 - np.exp(-h.cgs.value*nu / (k_B.cgs.value*T)))
             opacities = {}

             # H0 ligado-libre
             opacities["H0bf"] = 0.0
             nmax = int(nmax_pressure_ionization(Hden))
             Un = H0_partition_function(T, nmax)
             for n in range(1, nmax+1):
                 opacities["H0bf"] += H0den * H0_level_population(n, T, Un) * xsec_H0_boundfree
             opacities["H0bf"] *= stimulated_correction
             # H0 libre-libre
             opacities["H0ff"] = Hpden * eden * xsec_H0_freefree(T, nu)
             opacities["H0ff"] *= stimulated_correction
             # H- ligado-libre
             opacities["Hmbf"] = Hmden * xsec_Hminus_boundfree(nu)
             opacities["Hmbf"] *= stimulated_correction
             # H- libre-libre (que ya incluye emisión estimulada)
             opacities["Hmff"] = H0den * Pe * xsec_Hminus_freefree(T, nu)
             # convertir a opacidad por masa
             total = 0.0
             for k in opacities.keys():
               m = opacities[k] < 0.0
               opacities[k][m] = 0.0
               opacities[k] /= H0den*m_p.cgs.value
               total += opacities[k]
             opacities["Total"] = total
             # guardar metadata
             opacities["metadata"] = {'N_H': Hden, 'y_H': y}

             return opacities
```

```
In [46]: opacidad_total(10.0, 1e4, np.array([3000, 10000]))
```

```
Out[46]: {'H0bf': array([ 140.07737091,   49.62336568]),
          'H0ff': array([  5.89895620e-05,   1.63766003e-03]),
          'Hmbf': array([ 0.01656623,  0.02988688]),
          'Hmff': array([ 0.00217084,  0.02112792]),
          'Total': array([ 140.09616697,   49.67601813]),
          'metadata': {'N_H': array(7405961339882.429),
           'y_H': array(0.977992282399029)}}
```

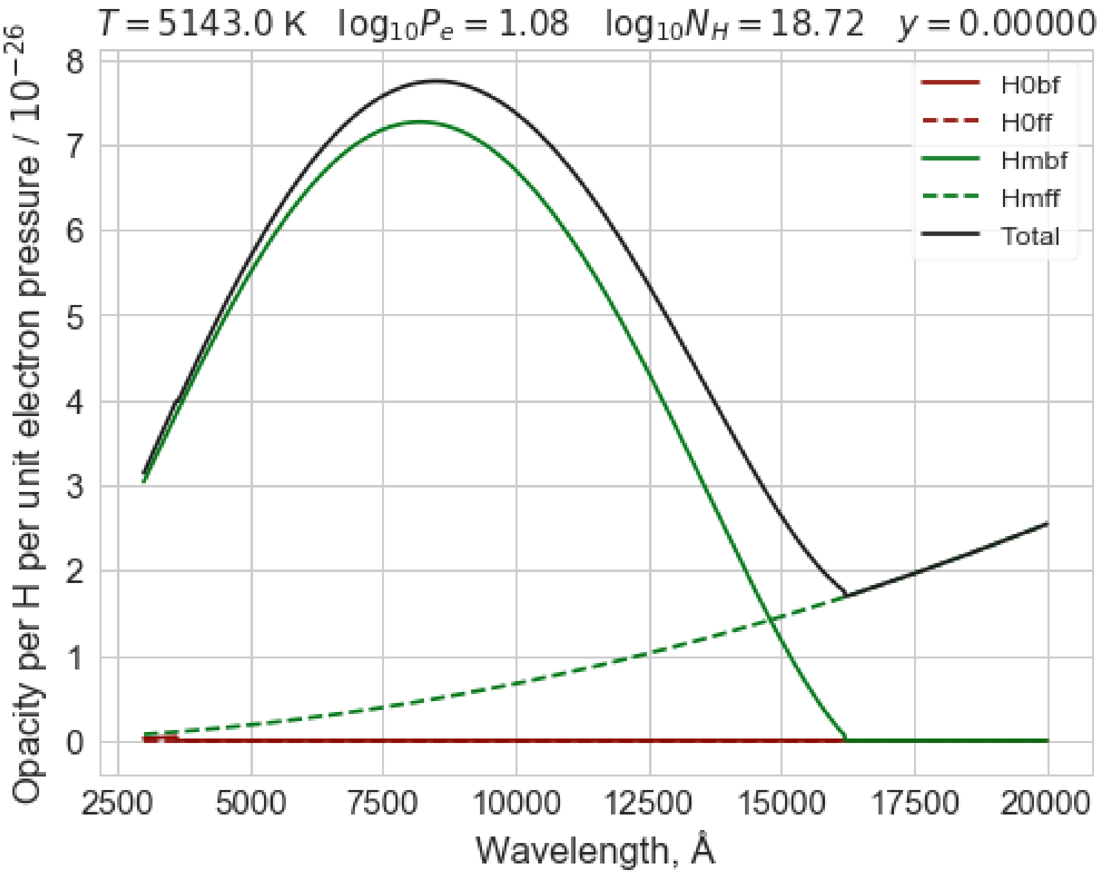
1.6.1 Reproducing Gray's Fig 8.5

```
In [47]: styles = {
           'Total': {'color': 'k', 'ls': '-'},
           'H0bf': {'color': 'r', 'ls': '-'},
           'H0ff': {'color': 'r', 'ls': '--'},
           'Hmbf': {'color': 'g', 'ls': '-'},
           'Hmff': {'color': 'g', 'ls': '--'},
         }
         def plot_opacities(Pe, T, wavrange=[3000., 20000.], yscale='linear'):
           wavs = np.linspace(wavrange[0], wavrange[1], 500)
           fig, ax = plt.subplots(1, 1)
           opac = opacidad_total(Pe, T, wavs)
           data = opac.pop('metadata')
           for kwd in opac.keys():
             ax.plot(wavs, opac[kwd]*m_p.cgs.value/Pe/1e-26, label=kwd, **styles[kwd])
           frame = ax.legend(loc='upper right', **legend_box_params).get_frame()
           frame.set_facecolor('white')
           strings = []
           strings.append('$T = {}$ K'.format(T))
           strings.append(r'$\log_{{10}} P_e = {:.2f}$'.format(np.log10(Pe)))
           strings.append(r'$\log_{{10}} N_H = {:.2f}$'.format(np.log10(float(data['N_H']))))
           strings.append('$y = {:.5f}$'.format(float(data['y_H'])))
           ax.set_title(r'$\quad$'.join(strings), fontsize='small')
```

```
    ax.set_xlabel('Wavelength, Å')
    ax.set_ylabel('Opacity per H per unit electron pressure / $10^{-26}$')
    ax.set_yscale(yscale)
    return None
```

Fig 8.5 (a) — 5143 K

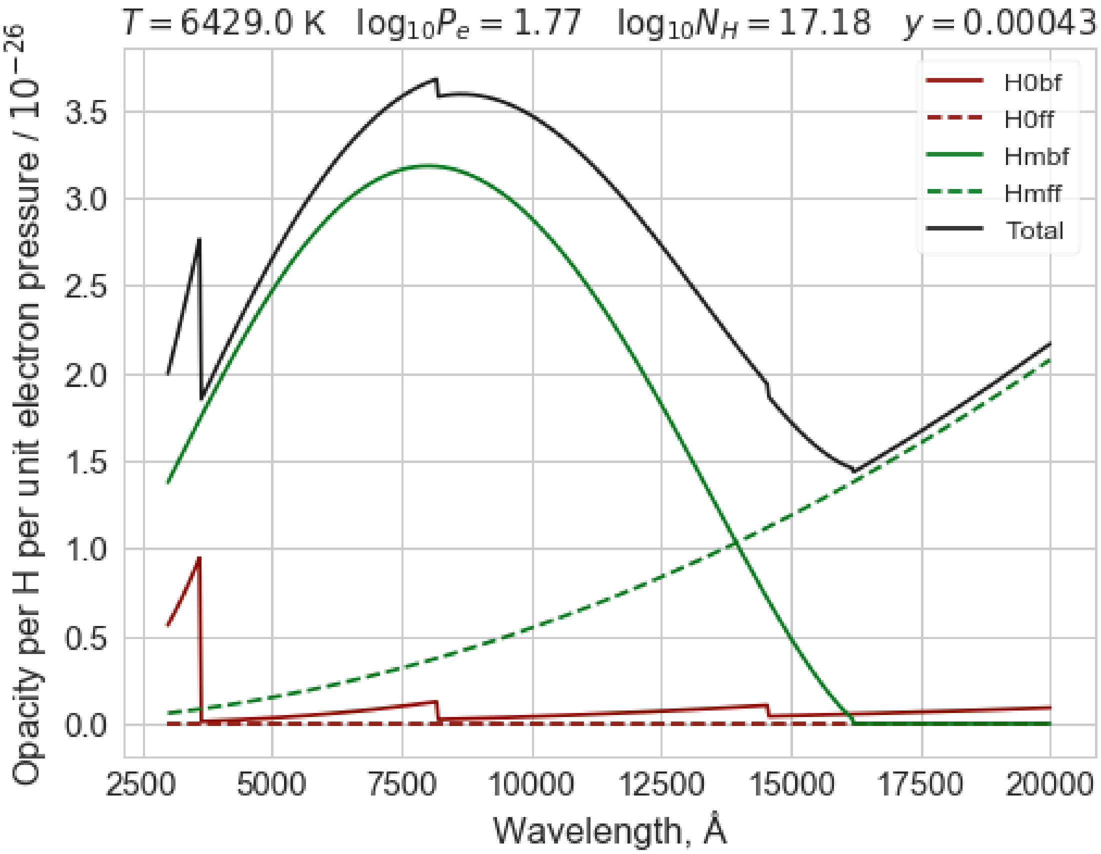
```
In [48]: plot_opacities(10**1.08, 5143.0)
```



Lowest temperature. Dominated by H^- opacity. *Why does Gray not get such a high free-free opacity as we do?*

Fig 8.5 (b) — 6429 K

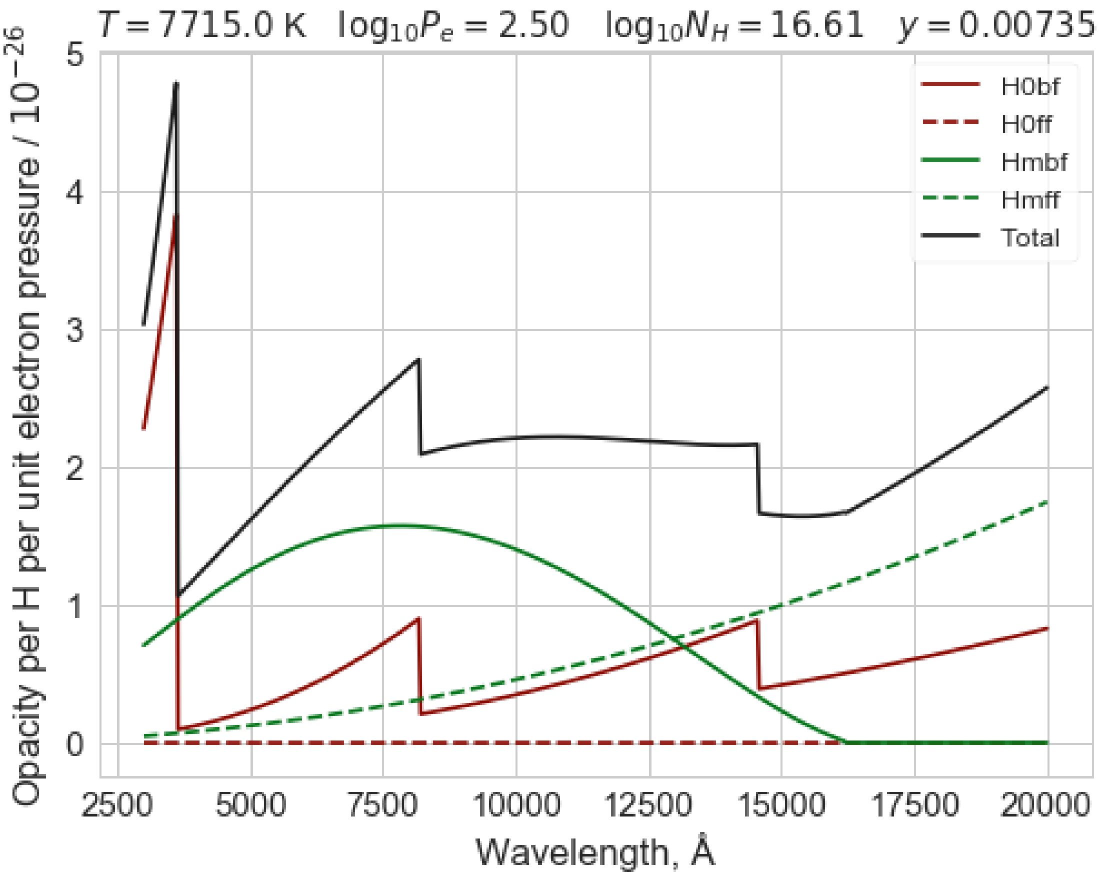
```
In [49]: plot_opacities(10**1.77, 6429.0)
```



Start to see the H^0 absorption edges superimposed on the H^-. Of the four graphs, this is the one that looks most like Gray's version.

Fig 8.5 (c) — 7715 K

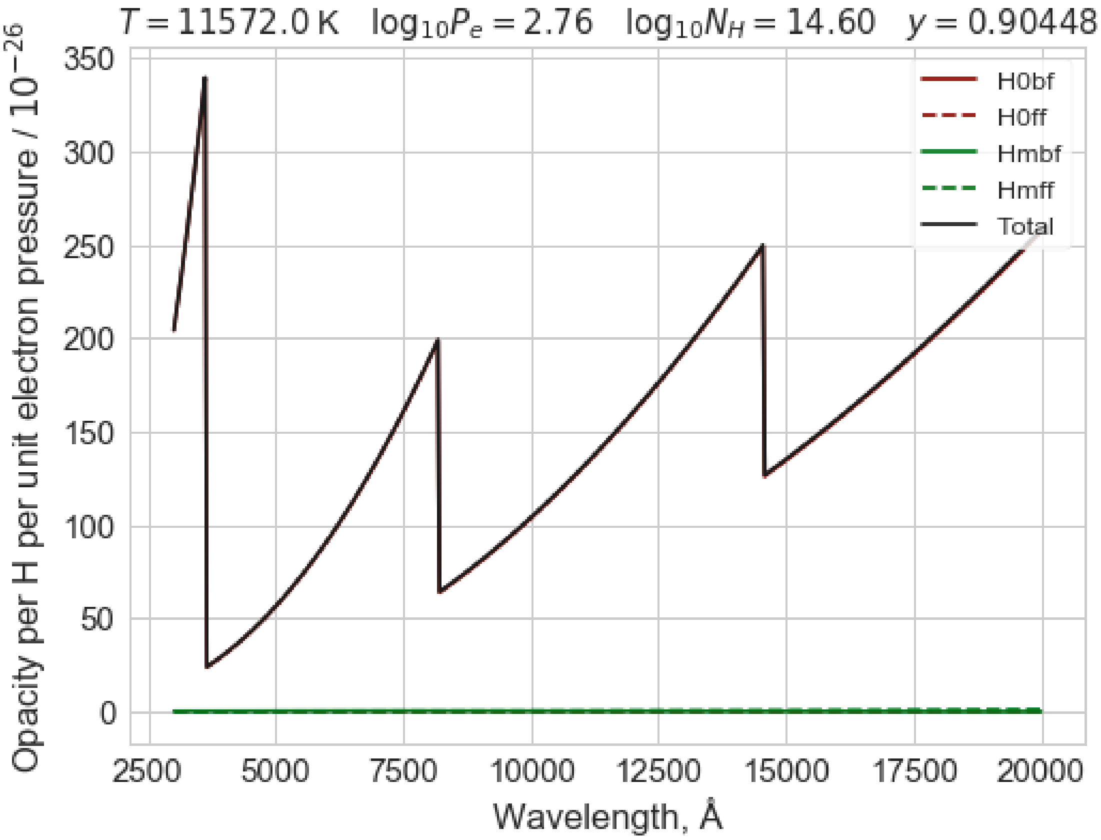
```
In [50]: plot_opacities(10**2.50, 7715.0)
```



H^0 and H^- are of roughly equal importance at this temperature.

Fig 8.5 (d) — 11,572 K

```
In [51]: plot_opacities(10**2.76, 11572.0)
```

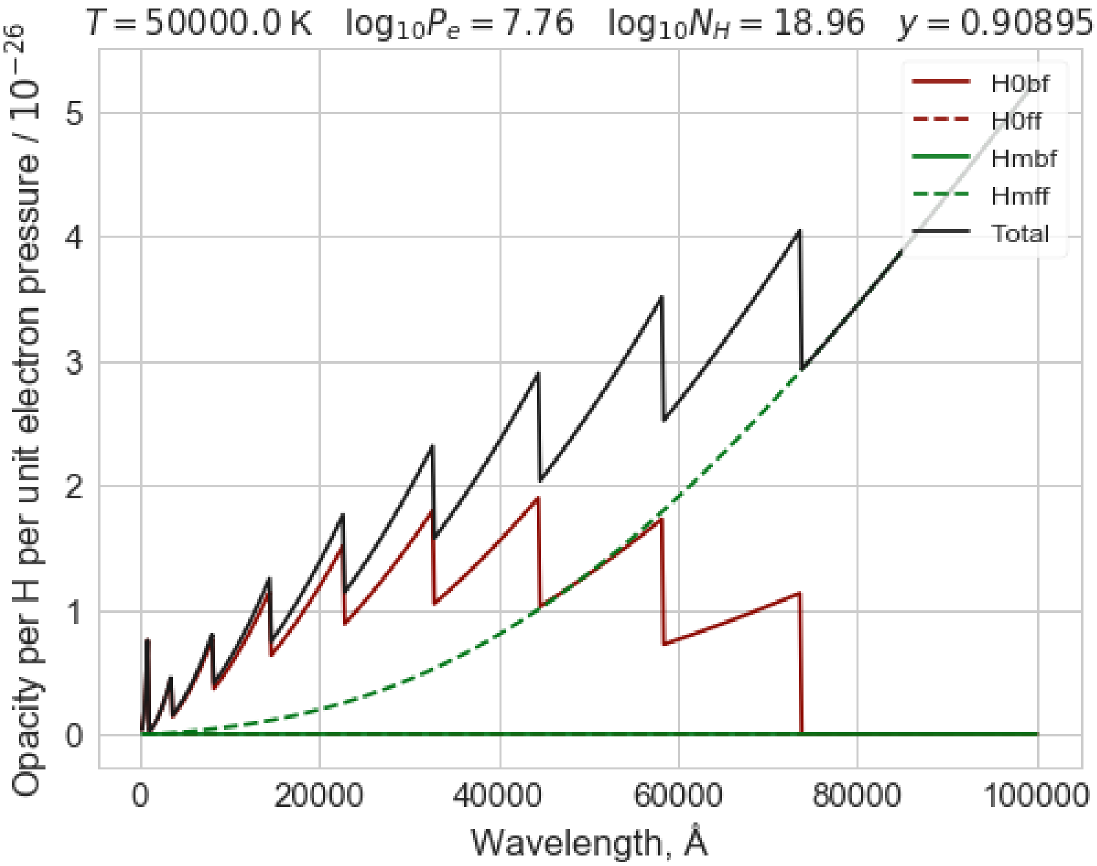


H^0 opacity now completely dominates. *My excited levels are not as high as in Gray's graph, – why?*

Note that in the 2016 version, my general magnitude was 20 times lower. But that is because I was normalizing by the total H density, whereas Gray is using (strangely) the neutral atomic hytrogen density. This only matters for panel (d), since it is the only case where H is appreciably ionized.

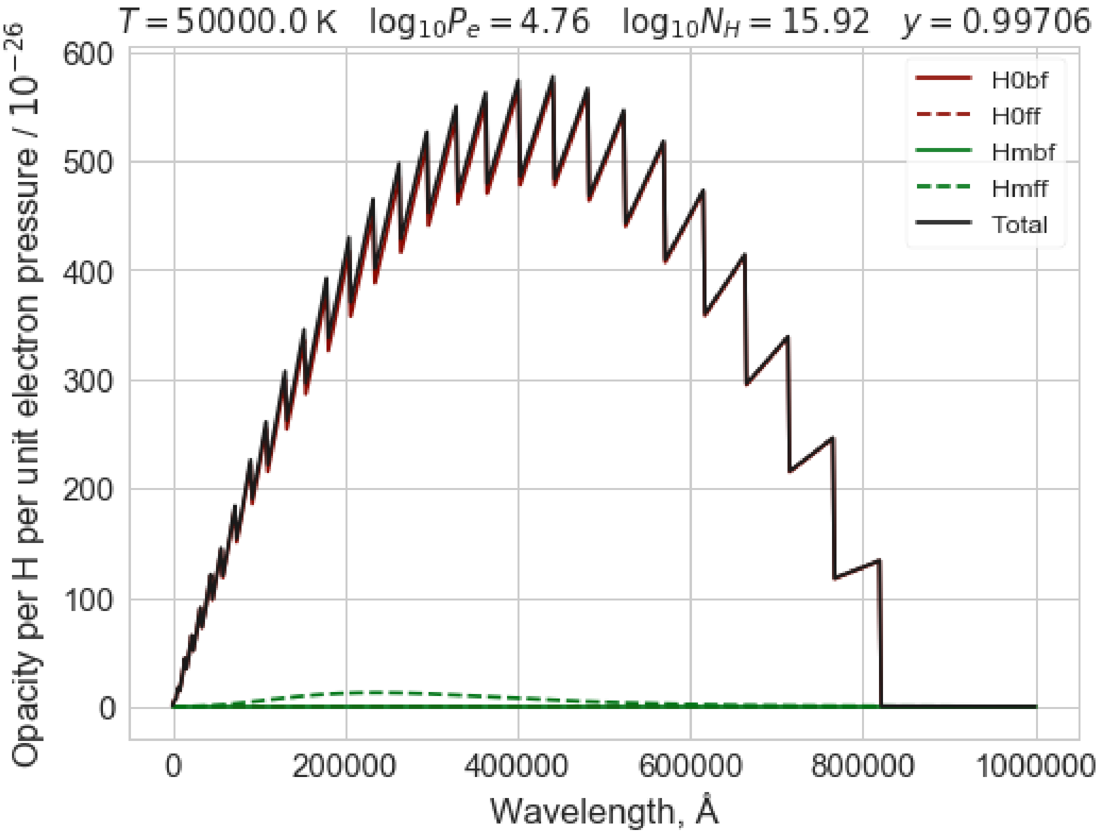
A much higher temperature and density

```
In [52]: plot_opacities(10**7.76, 50000.0, wavrange=[300, 100000])
```

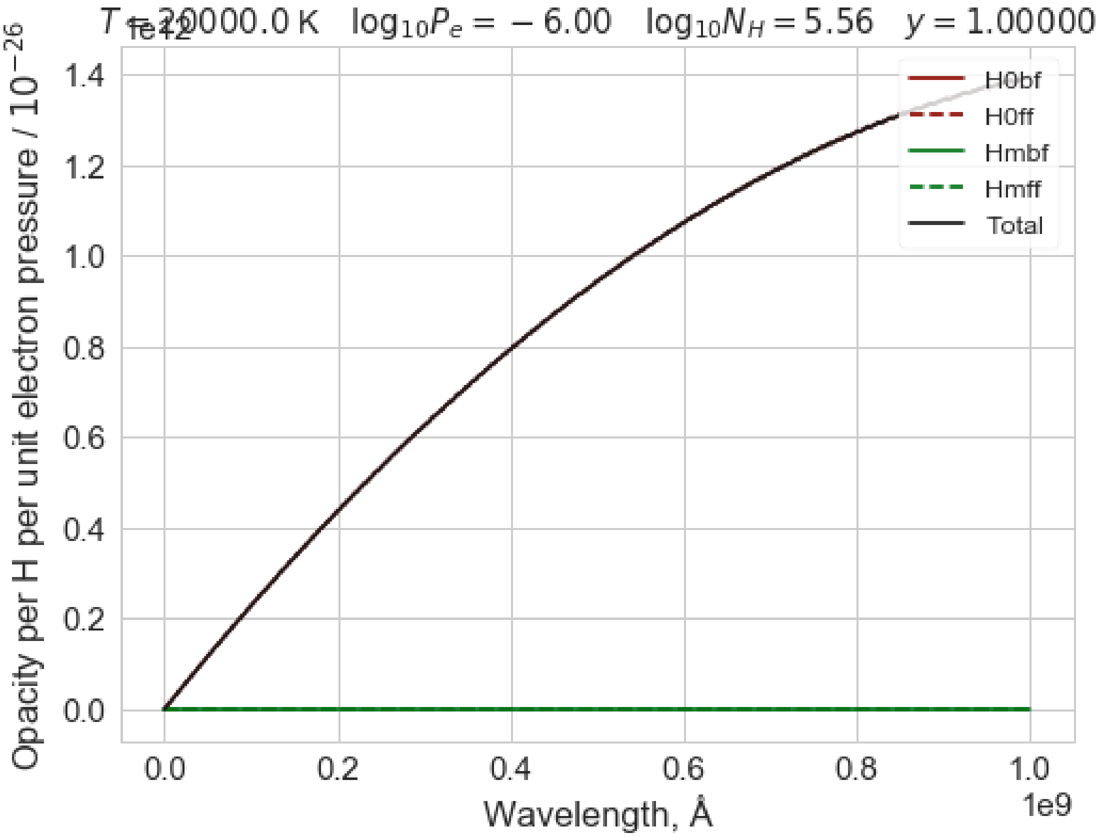


Now we see the pressure-ionization of the upper levels (only $n \leq 8$ are populated).

```
In [53]: plot_opacities(10**4.76, 50000.0, wavrange=[300, 1e6])
```



```
In [54]: plot_opacities(1e-6, 20000.0, wavrange=[300, 1e9])
```



1.7 Check against Cloudy results

```
In [55]: from astropy.table import Table
```

```
In [56]: Table.read?
```

```
In [57]: fn = "cloudy/lte_opacity_6429.opac"
         tc = Table.read(fn, format='ascii.tab', names=['nu', 'tot', 'abs', 'scat', 'albedo',
```

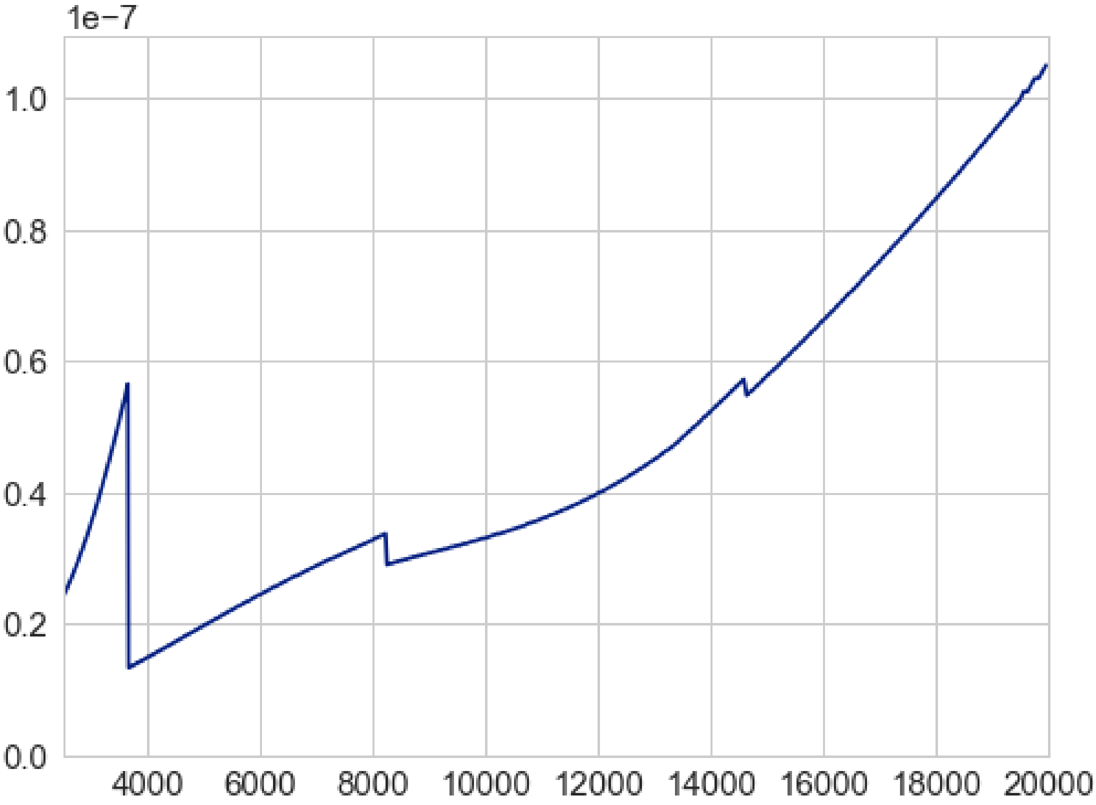
```
In [58]: tc['wav'] = 912.0/tc['nu']
         wavmin, wavmax = 2500, 20000
         m = (tc['wav'] >= wavmin) & (tc['wav'] <= wavmax)
         tc[m]
```

```
Out[58]: <Table length=624>
            nu      tot       abs      scat     albedo   elem      wav
         float64  float64  float64  float64  float64   str4    float64
         -------- -------- -------- -------- -------- ---- -------------
         0.045731 1.05e-07 1.05e-07 3.09e-11 0.000295      19942.7084472
         0.045884 1.04e-07 1.04e-07 3.09e-11 0.000297       19876.209572
         0.046037 1.03e-07 1.03e-07 3.09e-11 0.000299      19810.1527033
```

```
 0.04619 1.03e-07 1.03e-07 3.09e-11 0.000301      19744.5334488
0.046345 1.02e-07 1.02e-07 3.09e-11 0.000303      19678.4982199
0.046499 1.01e-07 1.01e-07 3.09e-11 0.000305      19613.3250177
0.046655 1.01e-07    1e-07 3.09e-11 0.000307      19547.7440789
 0.04681 9.98e-08 9.98e-08 3.09e-11  0.00031      19483.0164495
0.046967 9.91e-08 9.91e-08 3.09e-11 0.000312      19417.8891562
0.047123 9.85e-08 9.84e-08 3.09e-11 0.000314      19353.6065191
     ...      ...      ...      ...      ...  ...           ...
 0.35395 2.56e-08 2.41e-08 1.59e-09   0.0619      2576.63511795
 0.35513 2.55e-08 2.39e-08 1.61e-09   0.0633      2568.07366317
 0.35632 2.53e-08 2.37e-08 1.64e-09   0.0647      2559.49708128
 0.35751 2.51e-08 2.35e-08 1.66e-09   0.0661      2550.97759503
  0.3587  2.5e-08 2.33e-08 1.69e-09   0.0676      2542.51463619
  0.3599 2.48e-08 2.31e-08 1.72e-09   0.0691      2534.03723256
  0.3611 2.47e-08 2.29e-08 1.74e-09   0.0707      2525.61617281
 0.36231 2.45e-08 2.27e-08 1.77e-09   0.0722      2517.18141923
 0.36352 2.44e-08 2.26e-08  1.8e-09   0.0739       2508.8028169
 0.36473 2.42e-08 2.24e-08 1.83e-09   0.0755      2500.47980698
```

```
In [59]: fig, ax = plt.subplots()
         ax.plot(tc[m]['wav'], tc[m]['tot'])
         ax.set(xlim=[wavmin, wavmax], ylim=[0.0, None])
```

```
Out[59]: [(0.0, 1.09585e-07), (2500, 20000)]
```



1.7.1 First concentrate on ionization fractions and level populations

I have run some grid models based on the Cloudy test script `limit_lte_hminus.in`, in which I vary the density.

```
In [76]: fn = "cloudy/limit_lte_hminus_density_grid.hcond"
         thc = Table.read(fn, format='ascii.commented_header', delimiter='\t')
         thl = Table.read(fn.replace('hminus', 'hminus_large'), format='ascii.commented_header
         thf = Table.read(fn.replace('limit', 'force'), format='ascii.commented_header', delimi
```

The `thc` table is from model where we let Cloudy find the equilibrium on its own

```
In [61]: thc[::4]
```

```
Out[61]: <Table length=5>
          depth     Te         HDEN           EDEN       ...  H2+/H    H3+/H     H-/H
         float64 float64     float64        float64      ... float64  float64  float64
         ------- ------- ------------- --------------- ... -------- -------- --------
             0.5  5000.0   100000000.0      33600000.0 ... 4.64e-14 1.08e-20 3.74e-14
             0.5  5000.0 10000000000.0     404000000.0 ... 8.07e-13 2.11e-18  6.5e-13
             0.5  5000.0         1e+12    4170000000.0 ... 8.53e-12 1.55e-16 6.96e-12
             0.5  5000.0         1e+14   37000000000.0 ... 3.38e-11 1.61e-15 5.99e-11
             0.5  5000.0         1e+16  205000000000.0 ... 3.66e-12 6.74e-16 4.34e-11
```

While the `thf` table is from model where we try to force LTE.

```
In [62]: thf[::4]
```

```
Out[62]: <Table length=5>
          depth     Te         HDEN           EDEN       ...  H2+/H    H3+/H     H-/H
         float64 float64     float64        float64      ... float64  float64  float64
         ------- ------- ------------- --------------- ... -------- -------- --------
             0.5  5000.0   100000000.0      33400000.0 ... 4.63e-14 1.08e-20 3.73e-14
             0.5  5000.0 10000000000.0     401000000.0 ... 8.02e-13 2.11e-18 6.46e-13
             0.5  5000.0         1e+12    4090000000.0 ... 8.38e-12 1.54e-16 6.83e-12
             0.5  5000.0         1e+14   41000000000.0 ... 3.77e-11 1.77e-15 6.64e-11
             0.5  5000.0         1e+16  410000000000.0 ... 8.44e-12 1.52e-15 8.95e-11
```

The `thl` model has 40 H^0 resolved levels instead of just 10

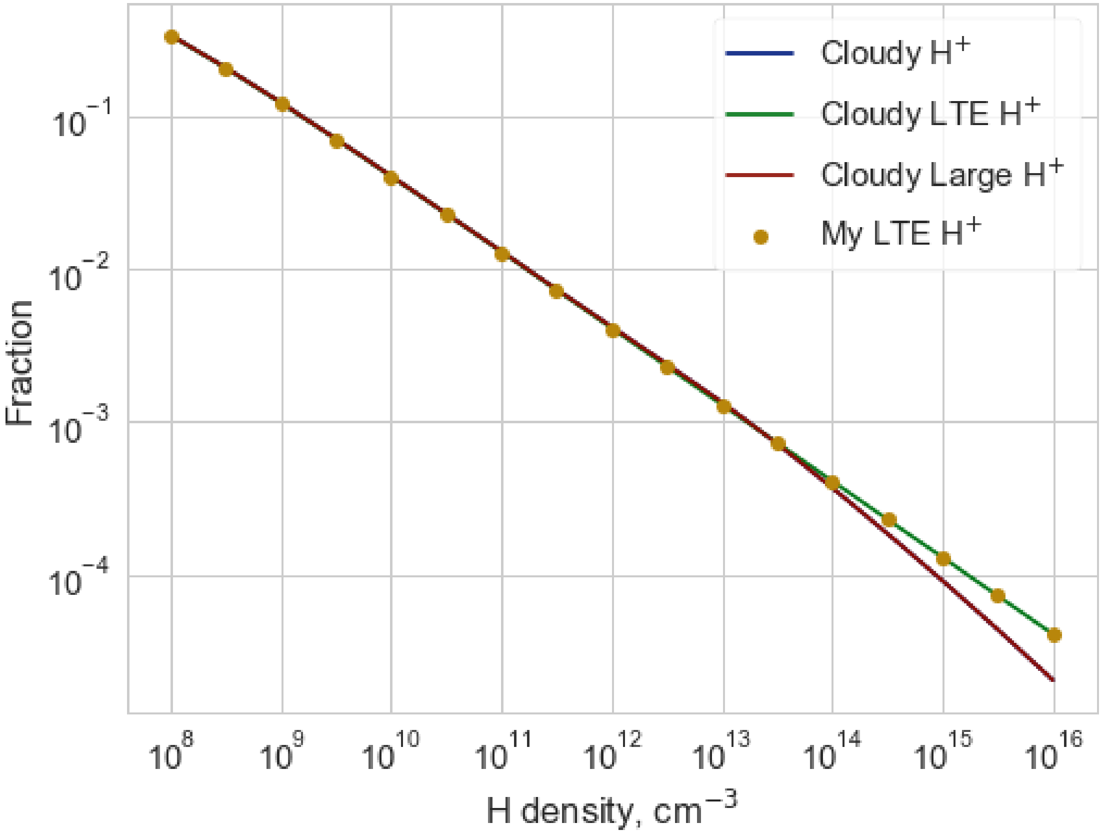
```
In [77]: thl[::4]
```

```
Out[77]: <Table length=5>
          depth     Te         HDEN           EDEN       ...  H2+/H    H3+/H     H-/H
         float64 float64     float64        float64      ... float64  float64  float64
         ------- ------- ------------- --------------- ... -------- -------- --------
             0.5  5000.0   100000000.0      33600000.0 ... 4.64e-14 1.08e-20 3.74e-14
```

```
0.5   5000.0 10000000000.0    404000000.0 ... 8.07e-13 2.11e-18  6.5e-13
0.5   5000.0         1e+12   4170000000.0 ... 8.53e-12 1.55e-16 6.96e-12
0.5   5000.0         1e+14  37000000000.0 ... 3.38e-11 1.61e-15 5.99e-11
0.5   5000.0         1e+16 205000000000.0 ... 3.66e-12 6.74e-16 4.34e-11
```

Plot the H^0 and H^+ fractions:

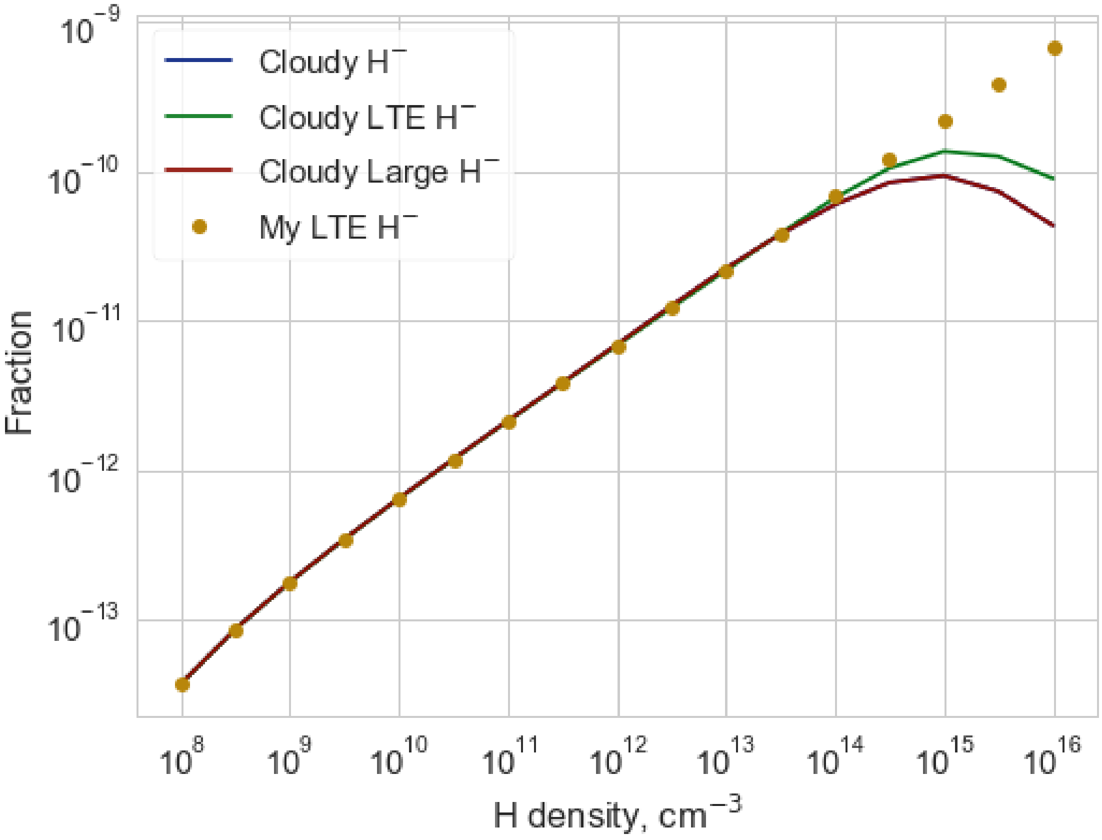
```
In [85]: fig, ax = plt.subplots()
         #ax.plot(thc['HDEN'], thc['HI/H'], label='Cloudy H$^0$')
         ax.plot(thc['HDEN'], thc['HII/H'], label='Cloudy H$^{+}$')
         ax.plot(thc['HDEN'], thf['HII/H'], label='Cloudy LTE H$^{+}$')
         ax.plot(thc['HDEN'], thl['HII/H'], label='Cloudy Large H$^{+}$')
         ax.plot(thc['HDEN'], Hplus_fraction_U(thc['HDEN'], 5000.), 'o', c='y', label='My LTE H
         ax.set(xscale='log', yscale='log', xlabel='H density, cm$^{-3}$', ylabel='Fraction')
         leg = ax.legend(frameon=True, framealpha=0.8)
         leg.get_frame().set_facecolor('white')
         fig.savefig('cloudy/density_grid_hplus.pdf')
         fig.savefig('cloudy/density_grid_hplus.png')
         None
```



The green line and yellow dots are indistinguishable. This is a good start - at least we agree about the LTE H ionization. However, the blue line (Cloudy free-wheeling model) *does* show deviations for $N > 10^{13.5}$.

Now the negative hydrogen abundance: H^{-}

```
In [86]: fig, ax = plt.subplots()
         ax.plot(thc['HDEN'], thc['H-/H'], label='Cloudy H$^{-}$')
         ax.plot(thc['HDEN'], thf['H-/H'], label='Cloudy LTE H$^{-}$')
         ax.plot(thc['HDEN'], thl['H-/H'], label='Cloudy Large H$^{-}$')
         ax.plot(thc['HDEN'], Hminus_fraction(thc['HDEN'], 5000.), 'o', c='y', label='My LTE H$
         ax.set(xscale='log', yscale='log', xlabel='H density, cm$^{-3}$', ylabel='Fraction')
         leg = ax.legend(frameon=True, framealpha=0.8)
         leg.get_frame().set_facecolor('white')
         None
```



So, again, we see deviations for high densities. But this time, even the LTE Cloudy model disagrees with my Saha equation.

Could it be some kind of continuum lowering that is going on? Since H^{-} has only one bound level, then the close packing effect will not kick in until the density is so high that pressure ionization occurs.

Level populations We can't just read the data in with `Table.read` because the lines are of unequal lengths

Define a function to deal with inconsistent row lengths

```
In [65]: def pad_row(str_vals, ncols):
             nmissing = ncols - len(str_vals)
             if nmissing > 0:
                 # Pad with empty strings if we have too few values
                 return str_vals + ['']*nmissing
             elif nmissing < 0:
                 # Truncate if we have too many values
                 return str_vals[:ncols]
             else:
                 # Case of "just right"
                 return str_vals
```

```
In [66]: pad_row(['a', 'b'], 4)
```

```
Out[66]: ['a', 'b', '', '']
```

```
In [67]: pad_row(['a', 'b', 'c', 'd', 'e'], 4)
```

```
Out[67]: ['a', 'b', 'c', 'd']
```

```
In [68]: from astropy.io import ascii

         ##
         ## NOTE: This class can only be defined once per session
         ##
         class NoHeaderVary(ascii.NoHeader):
             _format_name = 'vary'
             _description = 'Basic table with variable length rows'
             def inconsistent_handler(self, str_vals, ncols):
                 return pad_row(str_vals, ncols)
```

```
In [88]: def get_level_lists(nresolved = 10, ncollapsed = 100):
             levels_n_l = [[f'n({qn},{ql})' for ql in range(qn)] for qn in range(1, nresolved+
             levels_n_l += [[f'n({qn})'] for qn in range(nresolved+1, ncollapsed+nresolved)]

             flat_levels = []
             for sublevels in levels_n_l:
                 flat_levels.extend(sublevels)

             return levels_n_l, flat_levels


         levels_n_l, flat_levels = get_level_lists()
         levels_n_l_large, flat_levels_large = get_level_lists(40, 60)

         flat_levels_large[-100:-50]
```

```
Out[88]: ['n(39,38)',
          'n(40,0)',
          'n(40,1)',
          'n(40,2)',
          'n(40,3)',
          'n(40,4)',
          'n(40,5)',
          'n(40,6)',
          'n(40,7)',
          'n(40,8)',
          'n(40,9)',
          'n(40,10)',
          'n(40,11)',
          'n(40,12)',
          'n(40,13)',
          'n(40,14)',
          'n(40,15)',
          'n(40,16)',
          'n(40,17)',
          'n(40,18)',
          'n(40,19)',
          'n(40,20)',
          'n(40,21)',
          'n(40,22)',
          'n(40,23)',
          'n(40,24)',
          'n(40,25)',
          'n(40,26)',
          'n(40,27)',
          'n(40,28)',
          'n(40,29)',
          'n(40,30)',
          'n(40,31)',
          'n(40,32)',
          'n(40,33)',
          'n(40,34)',
          'n(40,35)',
          'n(40,36)',
          'n(40,37)',
          'n(40,38)',
          'n(40,39)',
          'n(41)',
          'n(42)',
          'n(43)',
          'n(44)',
          'n(45)',
          'n(46)',
          'n(47)',
 'n(48)',
 'n(49)']
```

```
In [89]: names = ['depth', 'n(H0)', 'n(H+)'] + flat_levels
         tpc = Table.read(fn.replace('.hcond', '.hpop'), format='ascii.vary', names=names, gues
         tpf = Table.read(fn.replace('limit', 'force').replace('.hcond', '.hpop'), format='asc
```

```
In [71]: tpc
```

```
Out[71]: <Table masked=True length=17>
          depth      n(H0)          n(H+)          n(1,0)      ...  n(107)  n(108)  n(109)
         float64    float64        float64        float64      ... float64 float64 float64
         ------- -------------- -------------- --------------  ... ------- ------- -------
             0.5     66400000.0     33600000.0     66400000.0  ...  0.0152  0.0154  0.0157
             0.5    251000000.0     65300000.0    251000000.0  ...  0.0573  0.0583  0.0594
             0.5    878000000.0    122000000.0    878000000.0  ...     0.2   0.204   0.208
             0.5   2940000000.0    223000000.0   2940000000.0  ...   0.671   0.683   0.696
             0.5   9600000000.0    404000000.0   9600000000.0  ...    2.19    2.23    2.27
             0.5  30900000000.0    725000000.0  30900000000.0  ...    7.06    7.19    7.32
             0.5  98700000000.0   1300000000.0  98700000000.0  ...    22.6    23.0    23.4
             0.5 314000000000.0   2320000000.0 314000000000.0  ...    72.3    73.7    75.0
             0.5 996000000000.0   4170000000.0 996000000000.0  ...   233.0   238.0   242.0
             0.5       3.15e+12   7520000000.0       3.15e+12  ...   760.0   774.0   789.0
             0.5       9.99e+12  13400000000.0       9.99e+12  ...      --      --      --
             0.5       3.16e+13  22800000000.0       3.16e+13  ...      --      --      --
             0.5          1e+14  37000000000.0          1e+14  ...      --      --      --
             0.5       3.16e+14  58700000000.0       3.16e+14  ...      --      --      --
             0.5          1e+15  91500000000.0          1e+15  ...      --      --      --
             0.5       3.16e+15 139000000000.0       3.16e+15  ...      --      --      --
             0.5          1e+16 205000000000.0          1e+16  ...      --      --      --
```

```
In [72]: len(tpc[1])
```

```
Out[72]: 157
```

```
In [73]: tpf
```

```
Out[73]: <Table masked=True length=17>
          depth      n(H0)          n(H+)          n(1,0)      ...  n(107)  n(108)  n(109)
         float64    float64        float64        float64      ... float64 float64 float64
         ------- -------------- -------------- --------------  ... ------- ------- -------
             0.5     66600000.0     33400000.0     66600000.0  ...   0.015  0.0153  0.0156
             0.5    251000000.0     65000000.0    251000000.0  ...  0.0567  0.0577  0.0588
             0.5    879000000.0    121000000.0    879000000.0  ...   0.198   0.202   0.206
             0.5   2940000000.0    222000000.0   2940000000.0  ...   0.663   0.676   0.688
             0.5   9600000000.0    401000000.0   9600000000.0  ...    2.17    2.21    2.25
             0.5  30900000000.0    720000000.0  30900000000.0  ...    6.97     7.1    7.23
             0.5  98700000000.0   1290000000.0  98700000000.0  ...    22.3    22.7    23.1
             0.5 314000000000.0   2300000000.0 314000000000.0  ...    70.8    72.1    73.5
```

```
    0.5 996000000000.0   4090000000.0 996000000000.0 ...   225.0   229.0   233.0
    0.5       3.15e+12   7280000000.0       3.15e+12 ...   712.0   725.0   739.0
    0.5       9.99e+12  12900000000.0       9.99e+12 ...      --      --      --
    0.5       3.16e+13  23000000000.0       3.16e+13 ...      --      --      --
    0.5          1e+14  41000000000.0          1e+14 ...      --      --      --
    0.5       3.16e+14  72900000000.0       3.16e+14 ...      --      --      --
    0.5          1e+15 130000000000.0          1e+15 ...      --      --      --
    0.5       3.16e+15 230000000000.0       3.16e+15 ...      --      --      --
    0.5          1e+16 410000000000.0          1e+16 ...      --      --      --
```

```
In [90]: names = ['depth', 'n(H0)', 'n(H+)'] + flat_levels_large
         tpl = Table.read(fn.replace('hminus', 'hminus_large').replace('.hcond', '.hpop'), for
```

```
In [91]: tpl
```

```
Out[91]: <Table masked=True length=17>
          depth      n(H0)          n(H+)          n(1,0)      ...  n(97)   n(98)   n(99)
         float64    float64        float64        float64      ... float64 float64 float64
         ------- -------------- -------------- --------------  ... ------- ------- -------
             0.5     66400000.0     33600000.0     66400000.0  ...  0.0125  0.0127   0.013
             0.5    251000000.0     65300000.0    251000000.0  ...  0.0471   0.048   0.049
             0.5    878000000.0    122000000.0    878000000.0  ...   0.165   0.168   0.172
             0.5   2940000000.0    223000000.0   2940000000.0  ...   0.551   0.563   0.574
             0.5   9600000000.0    404000000.0   9600000000.0  ...     1.8    1.84    1.88
             0.5  30900000000.0    725000000.0  30900000000.0  ...     5.8    5.92    6.04
             0.5  98700000000.0   1300000000.0  98700000000.0  ...    18.6    19.0    19.3
             0.5 314000000000.0   2320000000.0 314000000000.0  ...    59.4    60.7    61.9
             0.5 996000000000.0   4170000000.0 996000000000.0  ...   192.0   196.0   200.0
             0.5       3.15e+12   7520000000.0       3.15e+12  ...   625.0   638.0   651.0
             0.5       9.99e+12  13400000000.0       9.99e+12  ...      --      --      --
             0.5       3.16e+13  22800000000.0       3.16e+13  ...      --      --      --
             0.5          1e+14  37000000000.0          1e+14  ...      --      --      --
             0.5       3.16e+14  58700000000.0       3.16e+14  ...      --      --      --
             0.5          1e+15  91500000000.0          1e+15  ...      --      --      --
             0.5       3.16e+15 139000000000.0       3.16e+15  ...      --      --      --
             0.5          1e+16 205000000000.0          1e+16  ...      --      --      --
```

```
In [92]: len(tpl[1])
```

```
Out[92]: 882
```

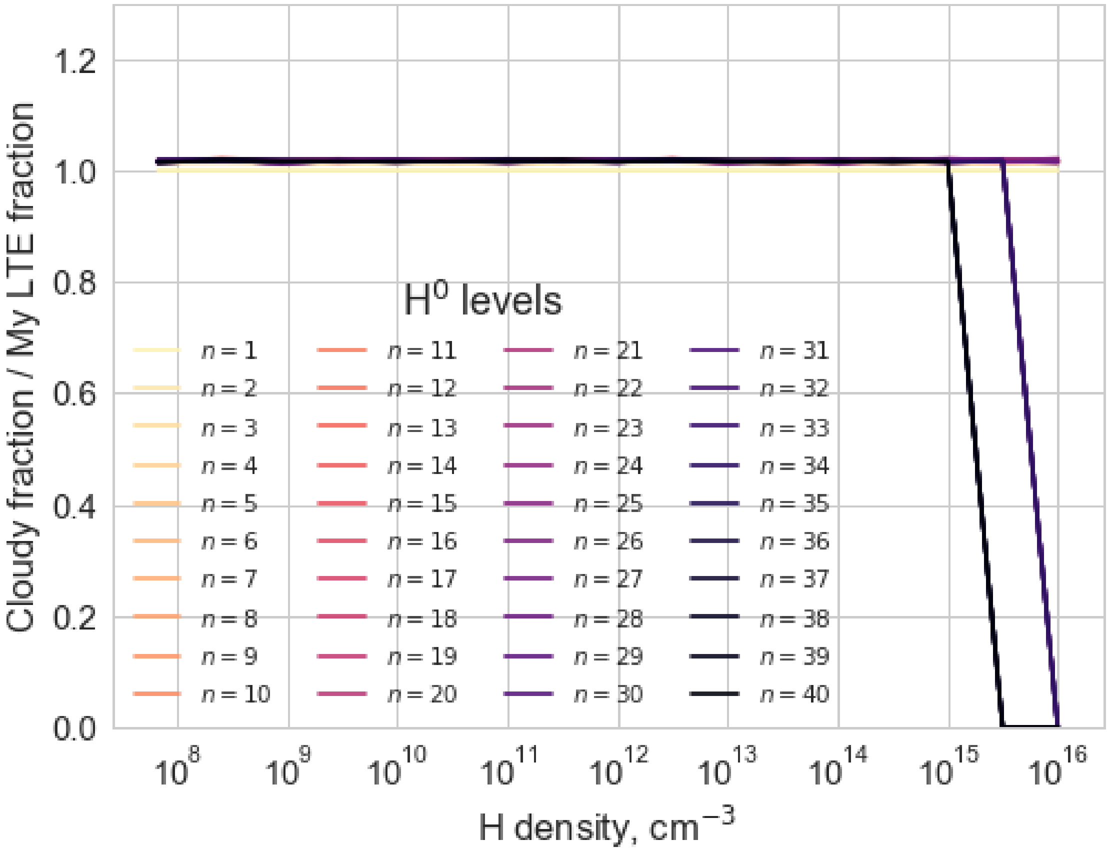
Sum over all the *l* states to get the population fractions for each *n*.

```
In [98]: def make_table_pop_n(tab_pop_n_l, levels_n_l):
             # Start with one column of the total H0
             tnfracs = tab_pop_n_l[['n(H0)']]
             # Loop over all the n-levels
             for i, sublevels in enumerate(levels_n_l):
                 qn = i + 1  # quantum number n
```

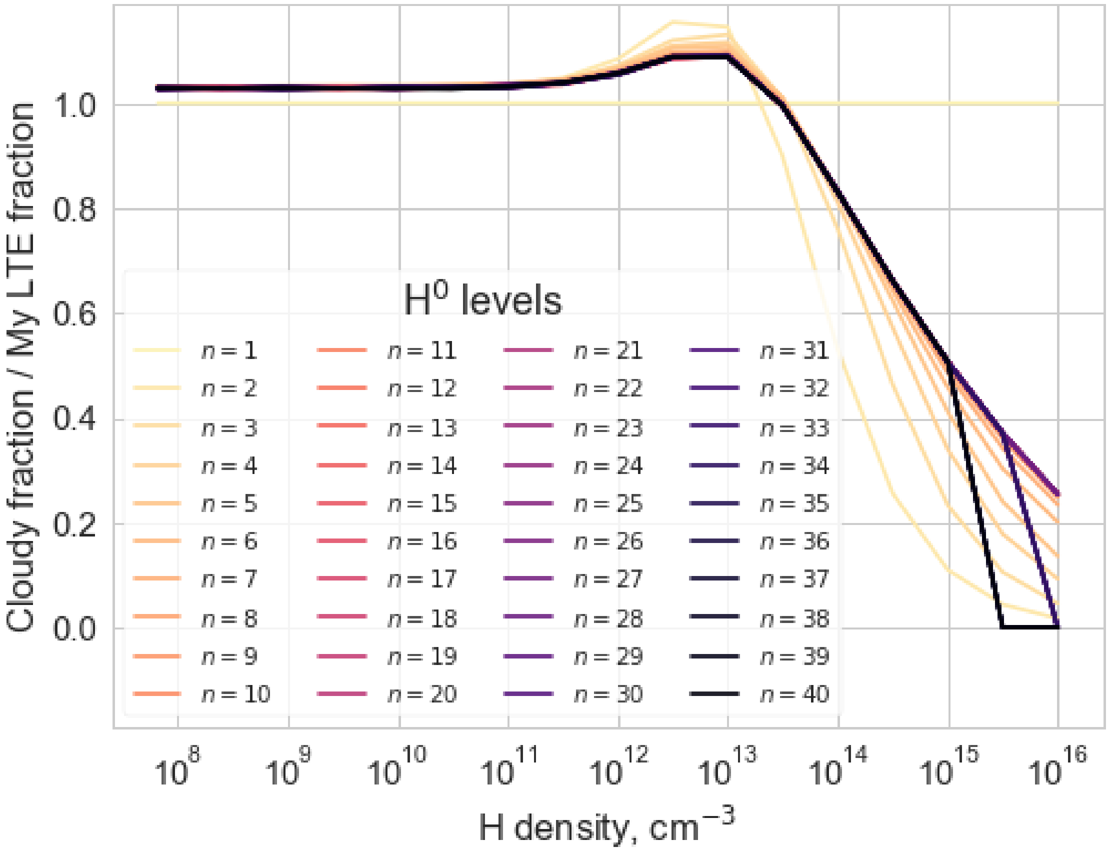
```
                # Add a column with total fraction for this n by summing the individual l pop
                tnfracs[f'n({qn})'] = np.sum([tab_pop_n_l[s] for s in sublevels], axis=0)/tab_
            return tnfracs

        tpf_n = make_table_pop_n(tpf, levels_n_l)
        tpc_n = make_table_pop_n(tpc, levels_n_l)
        tpl_n = make_table_pop_n(tpl, levels_n_l_large)
```

```
In [99]: fig, ax = plt.subplots()
         T = 5000.0
         qn_maxplot = 40
         colors = sns.color_palette('magma_r', n_colors=qn_maxplot)
         nmax = nmax_pressure_ionization(tpf_n['n(H0)'])
         print(nmax.data.astype(int))
         U = H0_partition_function(T, nmax=nmax)
         for i, c in list(enumerate(colors)):
             qn = i + 1
             myfracs = H0_level_population(qn, T, U=U)
             ax.plot(tpf_n['n(H0)'], tpf_n[f'n({qn})'] / myfracs, color=c, label=f'$n = {qn}$'
         ax.set(xscale='log', yscale='linear', xlabel='H density, cm$^{-3}$',
                ylabel='Cloudy fraction / My LTE fraction',
                ylim=[0.0, 1.3])
         ax.legend(ncol=4, fontsize='xx-small', loc='lower left', title='H$^0$ levels')
         None
```

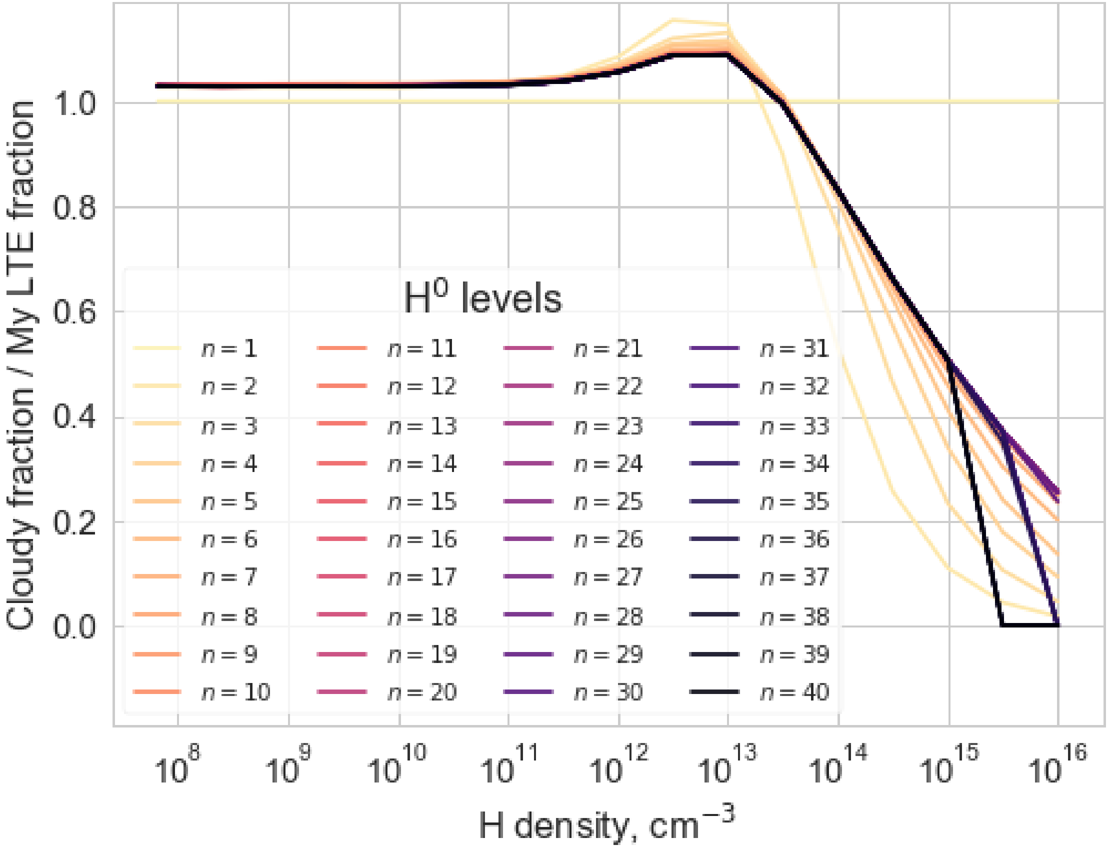
```
[682 547 444 363 298 245 202 166 137 113 93 77 63 52 43 35 29]
```



```
In [100]: fig, ax = plt.subplots()
          T = 5000.0
          qn_maxplot = 40
          colors = sns.color_palette('magma_r', n_colors=qn_maxplot)
          nmax = nmax_pressure_ionization(tpc_n['n(H0)'])
          #print(nmax.data.astype(int))
          #print(U - 2)
          U = H0_partition_function(T, nmax=nmax)
          for i, c in list(enumerate(colors)):
              qn = i + 1
              myfracs = H0_level_population(qn, T, U=U)
              ax.plot(tpc_n['n(H0)'], tpc_n[f'n({qn})'] / myfracs, color=c, label=f'$n = {qn}$
          ax.set(xscale='log', yscale='linear', xlabel='H density, cm$^{-3}$',
                 ylabel='Cloudy fraction / My LTE fraction',
                 ylim=[-0.19, 1.19])
          leg = ax.legend(ncol=4, fontsize='xx-small', loc='lower left', title='H$^0$ levels',
          leg.get_frame().set_facecolor('white')
          None
```



```
In [101]: fig, ax = plt.subplots()
          T = 5000.0
          qn_maxplot = 40
          colors = sns.color_palette('magma_r', n_colors=qn_maxplot)
          nmax = nmax_pressure_ionization(tpl_n['n(H0)'])
          #print(nmax.data.astype(int))
          #print(U - 2)
          U = H0_partition_function(T, nmax=nmax)
          for i, c in list(enumerate(colors)):
              qn = i + 1
              myfracs = H0_level_population(qn, T, U=U)
              ax.plot(tpl_n['n(H0)'], tpl_n[f'n({qn})'] / myfracs, color=c, label=f'$n = {qn}$
          ax.set(xscale='log', yscale='linear', xlabel='H density, cm$^{-3}$',
                 ylabel='Cloudy fraction / My LTE fraction',
                 ylim=[-0.19, 1.19])
          leg = ax.legend(ncol=4, fontsize='xx-small', loc='lower left', title='H$^0$ levels',
          leg.get_frame().set_facecolor('white')
          None
```



Something simpler would just be to sum all the excited levels.

```
In [131]: frac_tab = Table({'n(HO)': tpc_n['n(HO)'].data,
                            'n*/n': np.sum(tpc_n.columns.values()[2:], axis=0),
                            'n*/n LTE': np.sum(tpf_n.columns.values()[2:], axis=0),
                            'n*/n large': np.sum(tpl_n.columns.values()[2:], axis=0)})
```

```
In [124]: #np.sum([c.data for c in tpl_n.columns.values()[2:]], axis=0)
          np.sum(tpl_n.columns.values()[2:], axis=0)
          #[c for c in tpl_n.columns.values() if '1' in c.name][:4]
```

```
Out[124]: array([  6.81105437e-09,   6.80528335e-09,   6.80905353e-09,
                   6.80490850e-09,   6.80918052e-09,   6.81488922e-09,
                   6.82831094e-09,   6.87198904e-09,   6.99341064e-09,
                   7.21050467e-09,   5.67692142e-09,   2.99786509e-09,
                   1.39159590e-09,   6.24857722e-10,   2.70255990e-10,
                   1.23516899e-10,   4.96357100e-11])
```

```
In [133]: frac_tab
```

```
Out[133]: <Table masked=True length=17>
              n(HO)              n*/n              n*/n LTE           n*/n large
```

```
   float64          float64           float64           float64
-------------- ----------------- ----------------- -----------------
    66400000.0 8.98942891566e-09 8.88095555556e-09 6.81105436747e-09
   251000000.0 8.98381155378e-09 8.89561135458e-09 6.80528334661e-09
   878000000.0  8.9883429385e-09 8.87941501706e-09 6.80905353075e-09
  2940000000.0 8.98421418367e-09 8.88474867347e-09  6.8049085034e-09
  9600000000.0      8.98731e-09 8.88516177083e-09 6.80918052083e-09
 30900000000.0 8.99770106796e-09 8.88515546926e-09  6.8148892233e-09
 98700000000.0 9.01758682877e-09  8.8857751773e-09 6.82831094225e-09
314000000000.0 9.07407535032e-09 8.88365085987e-09 6.87198904459e-09
996000000000.0 9.23086666667e-09 8.88545230924e-09 6.99341064257e-09
      3.15e+12 9.51866984127e-09  8.8991747619e-09 7.21050466667e-09
      9.99e+12 5.68139479479e-09 5.28922822823e-09 5.67692142142e-09
      3.16e+13 2.99796613924e-09 3.08284392405e-09 2.99786509494e-09
         1e+14      1.391622e-09     1.7775214e-09     1.3915959e-09
      3.16e+14 6.24857025316e-10 1.09212560127e-09 6.24857721519e-10
         1e+15      2.702105e-10      7.104454e-10     2.7025599e-10
      3.16e+15 1.15080537975e-10 5.07761993671e-10 1.23516898734e-10
         1e+16      4.564534e-11      3.858564e-10      4.963571e-11
```

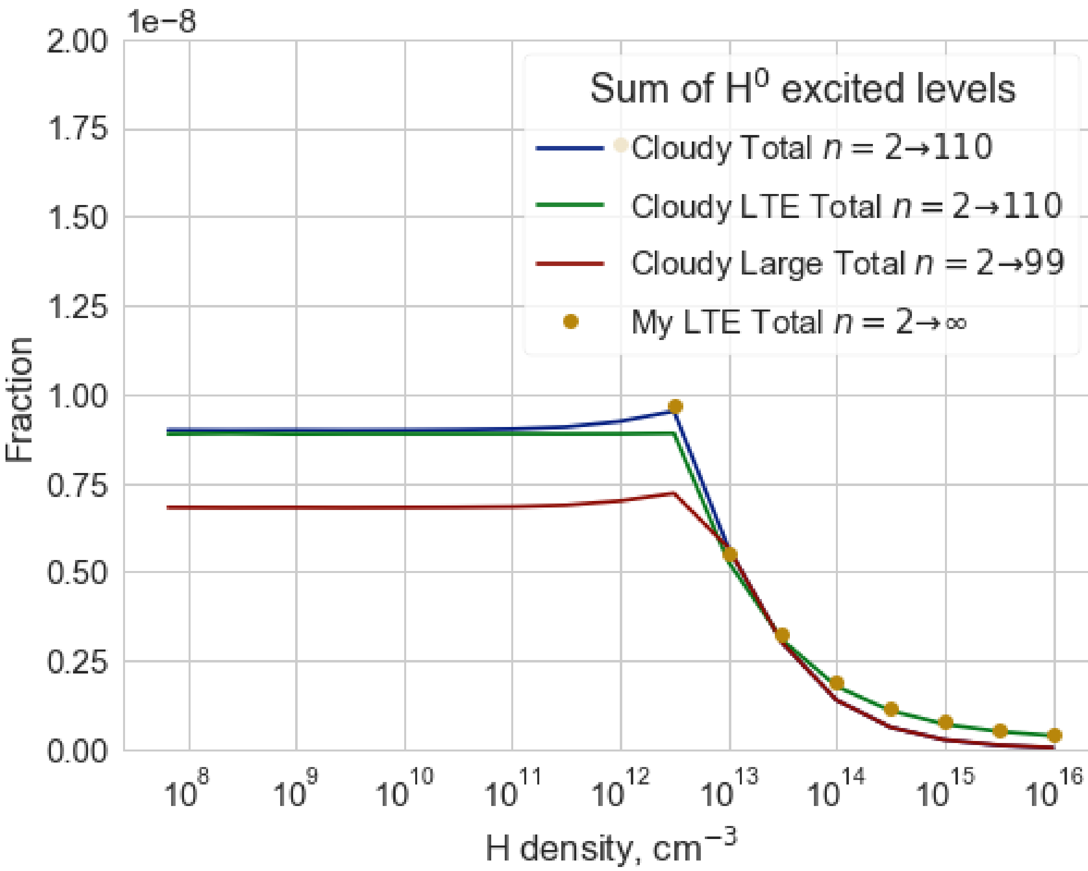
And we do the same from my own functions:

```
In [126]: @np.vectorize
          def H0_total_excited_population(Hden, T):
            nmax = nmax_pressure_ionization(Hden)
            U = H0_partition_function(T, nmax=nmax)
            pop = np.zeros_like(T)
            for n in range(2, int(nmax)+1):
              pop += H0_level_population(n, T, U=U)
            return pop
```

```
In [127]: H0_total_excited_population(1e12, 5000.)
```

```
Out[127]: array(1.7041750039083772e-08)
```

```
In [135]: fig, ax = plt.subplots()
          T = 5000.0
          ax.plot(frac_tab['n(H0)'], frac_tab['n*/n'], label=r'Cloudy Total $n=2 \to 110$')
          ax.plot(frac_tab['n(H0)'], frac_tab['n*/n LTE'], label=r'Cloudy LTE Total $n=2 \to 1
          ax.plot(frac_tab['n(H0)'], frac_tab['n*/n large'], label=r'Cloudy Large Total $n=2 \
          ax.plot(frac_tab['n(H0)'], H0_total_excited_population(frac_tab['n(H0)'], T), 'o', c
          ax.set(xscale='log', yscale='linear', ylim=[0.0, 2.e-8], xlabel='H density, cm$^{-3}
          leg = ax.legend(title='Sum of H$^0$ excited levels', frameon=True, framealpha=0.8)
          leg.get_frame().set_facecolor('white')
          None
```



So the disagreement of Cloudy with my values for low density is to be expected, since Cloudy only uses 110 levels, which is less than $n_{\max}$ for $N < 10^{13}$ cm^{-3}.

For higher densities, the agreement is very good with the Cloudy forced LTE model, but the non-forced model falls consistently below.

```
In [ ]:
```