Saha and Boltzmann Equations and Fraunhofer line strengths

In this lab, we will try to mimic the work of Cecilia Payne (https://en.wikipedia.org/wiki/Cecilia_Payne-Gaposchkin), investigating the relative strengths of absorption lines in stars. Below is the figure based on her work of combining Saha's and Boltzmann's equations:

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
from scipy import constants as constants
```

In [2]:
```
from IPython.display import Image
i = Image('figures/panye.png')
i
```

Out[2]:

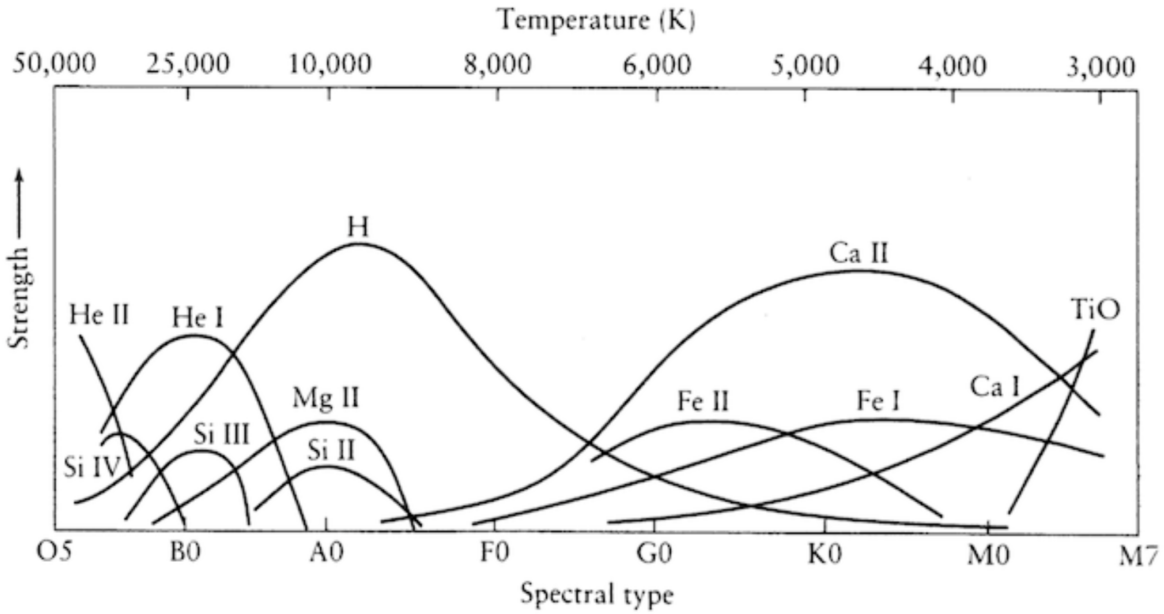


Figure 1. The diagram of absorption strength (see lecture notes).

In [3]:
```
i2 = Image('figures/payne_thesis.png')
i2
```

Out[3]:

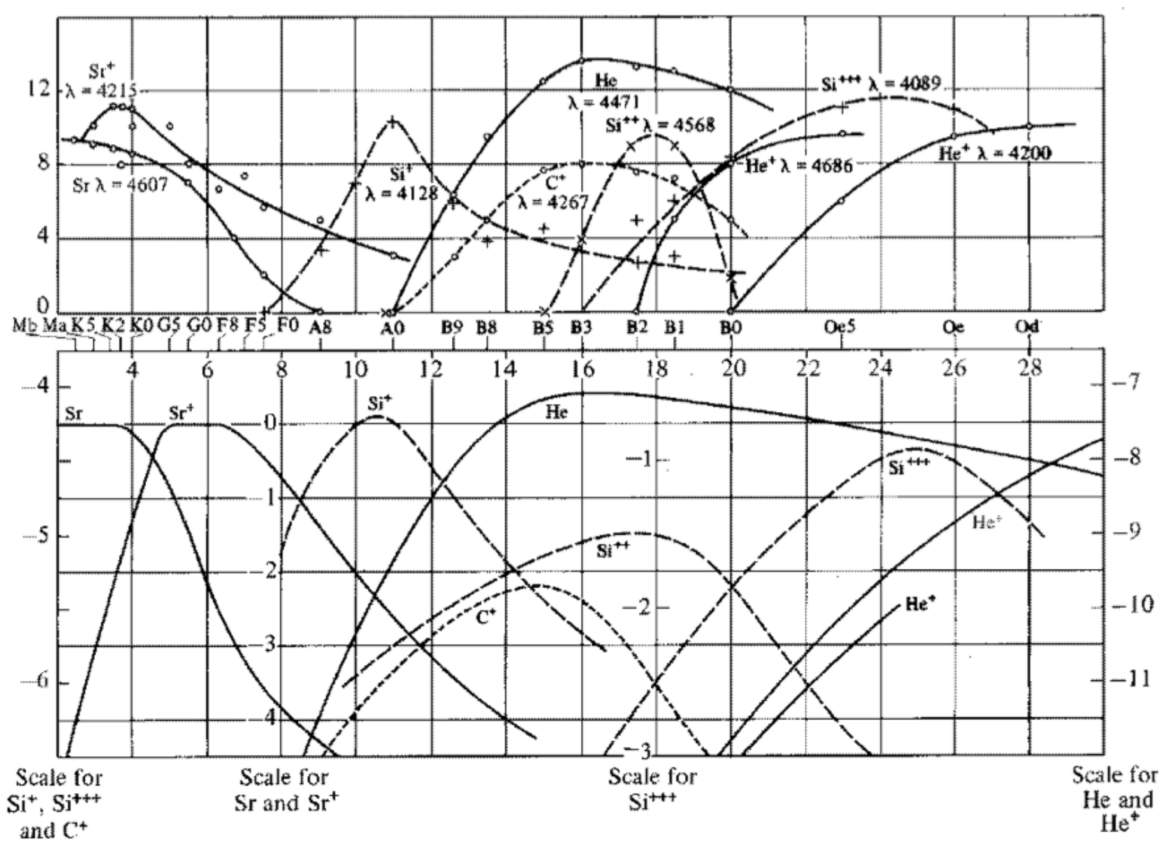


Figure 2. Original figure from Payne (1924). The strengths of selected lines along the spectral sequence. Upper panel: variations of observed line strengths with spectral type in the Harvard sequence (http://spiff.rit.edu/classes/phys301/lectures/class/class.html). The latter is plotted in reversed order on a non-linear scale that was obtained by making the peaks coincide with the corresponding peaks in the lower panel. The y-axis units are eye estimates on an arbitrary scale. Lower panel: Saha-Boltzmann predictions of the fractional concentration $N_{I,\,i}\,/\,N$ of the lower level of the lines indicated in the upper panel, each labeled with its ionization stage, on logarithmic y-axis scales that are specified per species at the bottom, against temperature T along the x axis given in units of 1000 K along the top. The pressure was taken constant at Pe = 13.1 N/m. From Novotny (1973) who took it from Payne (1924).

Exercise 1: Partition Function

1.1

Define the partition function we'll use with the Boltzmann equation:

$$Z_I = \sum_{i\,=\,0}^{i\,=\,n} g_i^I e^{-E_i^I\,/\,kT}$$

Here superscript I specifies the **degree of ionisation** (neutral, singly-ionised, twice-ionised, etc). Subscript i specifies a particular **energy level** of the ion. It gets confusing, so keep track of what is ionisation stage, and what is energy level.

E_i^I is the energy of state i for ion I. For the ground state, $E_0^I = 0$ eV. Each next excited state raises E_i^I, until the energy reaches the ionization potential, χ_I, and the electron is lost to the continuum. Summing over each bound state gives us the value of the partition function.

Define a Python function that evaluates the partition function. For simplicity, we'll assume that the different energy levels are 1 electron volt apart, so E_n varies by 1 eV in each loop of the above summation, and E_i^I will run from 0 (ground state) to χ_I (the ionisation energy of a given ion) in steps of 1 eV. We'll also assume that all statistical weights are 1 (so all $g_i^I = 1$).

The input to your function will be a vector of multiple ionisation energies, χ_I, (4, say, but this can be of variable length) and a temperature, T. For each ionisation energy of a given ion (e.g. Ca II), it will evaluate the sum in the partition function over all energy states (from 0 to the χ_I being considered). This means you'll need two nested for loops: one looping over the χ_I values of each ion (given in the input vector), and an inner loop evaluating the sum over all energy states available for that ion. The output will be a vector of partition function values associated with each ionisation energy given.

E.g.

```
def partfunc_E(chiI,T):
    # Inputs: chiI - Vector of ionisation energies for different
    #                ionisation stages of an element, in eV
    #            T - Temperature in Kelvin (scalar)
    # First loop over all chiI values (outer loop)
    #    Then loop over energy levels available to that ion. (inner loop)
    #    Each level is separated by 1eV, so the energies run from zero
    #       to ChiI, in steps of 1 eV. The end product of this loop is the sum
    #       over energy states for the given ion.
```

In [144]:
```
def partfunc_E(chiIarr, T):
    inc = 1 #increment in eV
    keV = constants.value("Boltzmann constant in eV/K")
    Z = [] #Empty array for the output
    for chiI in chiIarr:
        Z_ = 0. #Empty variable for the sum over energy levels
        E_i = 0. #Starting energy level
        while E_i < chiI: #Step through energy levels until E_i is greater than chiI
            Z_ += np.exp((-E_i)/(keV*T)) #Add the value to the running sum
            E_i += inc #Increment energy level by 1eV
        Z.append(Z_) #Add sum to the output array
    return Z
```

1.2

Test your function out by defining a vector of ionisation energies. For example, get the ionisation energies of calcium, Ca, from NIST (http://physics.nist.gov/PhysRefData/ASD/ionEnergy.html). In this case, 5 values are sufficient. Do this using the numpy.array() function to define your vector of values:

```
chiI=array([E1, E2, …])
```

Compute the partition functions for an energy of 10,000K. For example, using the ionisation energies of calcium and a temperature of 10,000K should give something like:

```
[ 1.45605581  1.45648718  1.45648849  1.45648849  1.45648849]
```

In [5]:
```
chiICa = np.array([6.1131554, 11.871719, 50.91316, 67.2732, 84.34]) #Ca I - Ca VI

T = 10e3
print(partfunc_E(chiICa, T))
```

```
[1.4559023270821698, 1.4563329540833081, 1.4563342588268766, 1.4563342588268766, 1.4563342588268766]
```

Exercise 2. Boltzmann Equation

The fraction of particles in a given energy state, i, of a given ion, I, is given by:

$$\frac{N_i^I}{N_I} = \frac{g_i^I}{Z_I} e^{-E_i^I / kT}$$

2.1

Write a function that returns this ratio for a given ionisation energy, a single temperature T, a given ionisation stage, I, and energy level, i. You will use your partition function from above. Again for simplicity, you can set the statistical weights $g_i^I = 1$, and recall that we are setting all the excitation energies for a given level to multiples of 1 eV, so E_i^I will be 0 eV for the ground state, 1 eV for the first excited state, and so on (but it should not exceed the ionization energy of your ion).

Test this works. You should get something like the following:

```
print chiI
print "i Fraction in level i for I=1 (neutral)"
print "- ------------------------------------"
for n in xrange(1,10):
    print n,boltz_E(chiI,10000,1,n)
```

Output:

```
[  7.9   16.2   30.65  54.91  75.  ]
i Fraction in level i for I=1 (neutral)
- -------------------
0 0.686655468072
1 0.215159740311
2 0.0674191293935
3 0.0211254159426
4 0.00661953369562
5 0.00207419472669
6 0.000649937588064
7 0.000203654393169
8 6.38139917104e-05
9 1.99957657414e-05
```

In [6]:
```
def boltz_E(chiI, T, I, i):
    k = constants.value("Boltzmann constant in eV/K")
    Z = partfunc_E(chiI, T) #Call partition function
    E_i = i
    fraction = np.exp((-E_i)/(k*T))/Z #Calculate Boltzmann function for all energy levels
    return fraction[I-1] #Return only the specified level
```

In [7]:
```
chiI = np.array([7.9, 16.2, 30.65, 54.91, 75.])
print(chiI)
print("")
print("i   Fraction in level i for I=1 (neutral)")
print("-   ------------------------------------")
for n in range(0,10):
    print(str(n)+"   "+str(boltz_E(chiI,10000,1,n)))
```

```
[  7.9   16.2   30.65  54.91  75.  ]

i   Fraction in level i for I=1 (neutral)
-   ------------------------------------
0   0.686719368773
1   0.21517970361
2   0.0674253660973
3   0.0211273643243
4   0.00662014237558
5   0.00207438487832
6   0.000649996990892
7   0.000203672950273
8   6.38197887898e-05
9   1.99975766823e-05
```

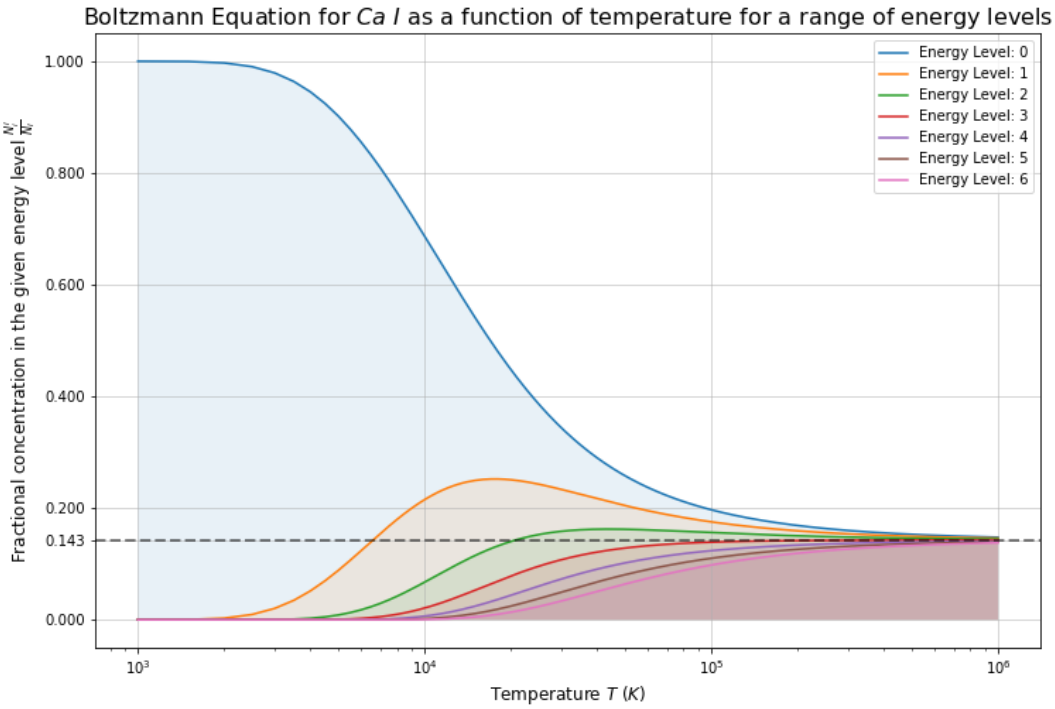
2.2

Explore the output for different energy levels as a function of temperature. Do this by using a for loop to call your function for different temperatures. Change the energy transition to explore what happens.

Give a response to the following questions:

- Is there a 'peak' temperature for a given energy level?
- How do the curves for different energy levels compare?
- How does the curve behave at extreme temperatures? Why?

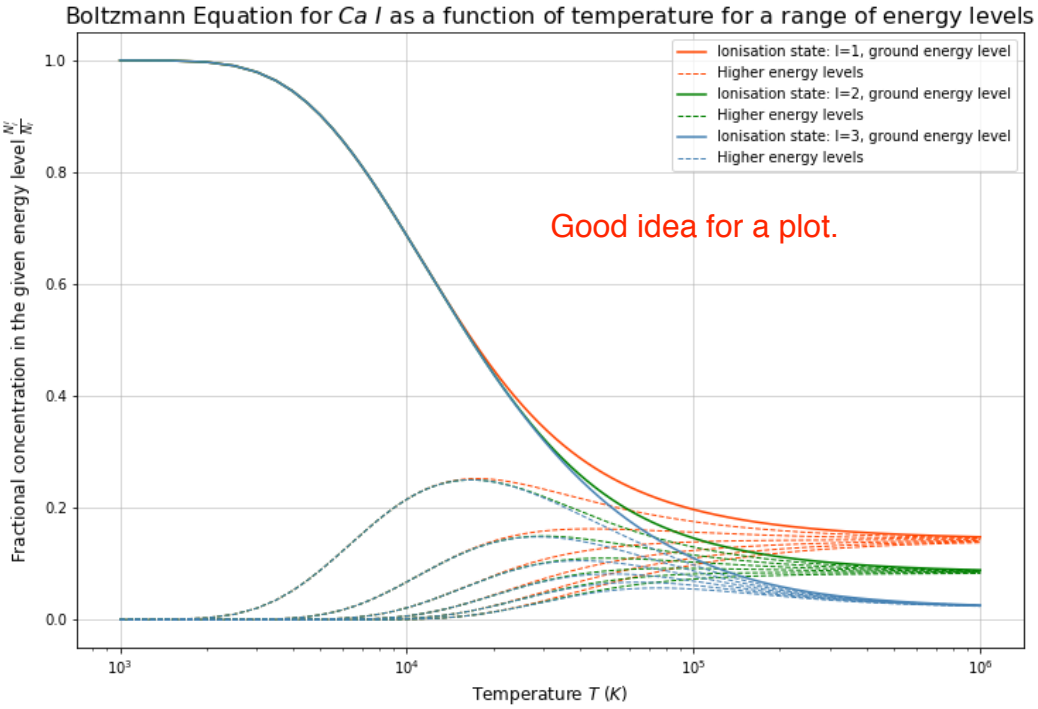
In [145]:
```
fig, ax = plt.subplots(1, 1, figsize=(12,8))
Ts = np.arange(1e3, 1e6, 5e2)
energy_levels = np.arange(0,7)
for i in energy_levels:
    frac_energy = [] #Empty array for the curve of each level population
    for T in Ts: #Step through the temperature range
        frac_energy.append(boltz_E(chiICa,T,1,i)) #And add data points at each step to the curve
    ax.plot(Ts, frac_energy, label="Energy Level: %.f" % i)
    ax.fill_between(Ts, 0, frac_energy, alpha=0.1)

ax.set_title("Boltzmann Equation for $Ca\ I$ as a function of temperature for a range of energy l
evels", size=16)
ax.set_xlabel("Temperature $T$ ($K$)", size=12)
ax.set_ylabel("Fractional concentration in the given energy level $\\frac{N_i^I}{N_I}$", size=12)
ax.legend()
ax.set_xscale("log")
extratick = 1./len(energy_levels) #Create an exta tickmark for the y-axis at an even split betwee
n all levels
ax.set_yticks(np.append(ax.get_yticks(), extratick))
ax.axhline(y=extratick, lw=2, ls="--", color="k", alpha=0.5) #Draw a dashed line at this level
ax.set_ylim(-0.05, 1.05)
ax.grid(lw=0.5)
```



There is a peak temperature for the first few energy levels. This temperature increases as a function of energy level above the ground state. As energy level increases, the function peaks at a lower value and is also pushed to higher temperatures. This corresponds to the increased liklihood of atoms being able to reach a particular energy level as temperature increases (to a point - the peak temperature - then a decreasing trend is shown). At very low temperatures, the curves are dominated by the ground state, which makes up for almost all atoms below around 5000K. As temperature increase to very high values, all curves equalise to one value, corresponding to an equal share of atoms between all energy states. For a two-state system, this value is 0.5, for a seven-state system such as is plotted above, this value is $\frac{1}{7} \approx 0.143$.

In [159]:
```
fig, ax = plt.subplots(1, 1, figsize=(12,8))
Ts = np.arange(1e3, 1e6, 5e2)
energy_levels = np.arange(0,7)
ionisation_states = [1,2,3]
colours = ["orangered", "green", "steelblue"]

for I, col in zip(ionisation_states, colours):
    ax.plot(0, 0, label="Ionisation state: I=%.0f, ground energy level" % I, color=col)
    ax.plot(0, 0, ls="--", lw=1, label="Higher energy levels", color=col)
    for i in energy_levels:
        frac_energy = []
        for T in Ts:
            frac_energy.append(boltz_E(chiICa,T,I,i))
        if i==0:
            ax.plot(Ts, frac_energy, color=col)
        else: ax.plot(Ts, frac_energy, color=col, ls="--", lw=1)

ax.set_title("Boltzmann Equation for $Ca\ I$ as a function of temperature for a range of energy l
```

```
evels", size=16)
ax.set_xlabel("Temperature $T$ ($K$)", size=12)
ax.set_ylabel("Fractional concentration in the given energy level $\\frac{N_i^I}{N_I}$", size=12)
ax.legend()
ax.set_xscale("log")
ax.set_ylim(-0.05, 1.05)
ax.grid(lw=0.5)
```



Here we can see three Boltzmann curves for three different ionisation states of Calcium. These follow the same function depending only on the number of possible energy levels beneath the ionisation energy χ_I.

Exercise 3: Saha Equation

3.1

Now write a function to evaluate Saha's equation. We will evaluate it in this form:

$$N_{I+1} = N_I \frac{Z_{I+1}}{Z_I} \frac{2kT}{h^3 P_e} (2\pi m_e kT)^{3/2} e^{-\frac{\chi_I}{kT}}$$

This gives the number of ions in state $I+1$. What we would like is the fraction of **all** states that exist in state I for a given temperature and pressure. The simplest way to get this ratio is to set $N_{I=1}$ (i.e. the neutral atom) to some value (e.g. unity), evaluate the next ionisation-stage populations successively from the Saha equation in a for loop, and at the end divide them by the sum of all the N on the same scale. You will find the numpy 'sum' function useful to get the total over all stages.

Your function should take as inputs the vector of ionisation energies as before, the temperature, the electron pressure, and the ionisation stage I that is being sought.

It should start something like:

```
def saha_E(chiI,T,Pe,I):
# compute Saha population fraction N_I/N
# input: ionisation energies, temperature, electron pressure, ion stage
# 1. Compute the partition functions
# 2. Loop over each ionisation stage that you have an energy for, computing
#    the fraction via saha. Note that the first stage should be set to 1.
# 3. Divide each stage by the total
# 4. Return the fraction of the requested stage
```

Check your function works. E.g. for the Ca ionisation values, a temperature of 5000K and an electron pressure of 100.0 N/m^2, you should get something like:

```
print "For ionisation energies (in eV) of:",chiI
print
print "I Fraction in stage I"
print "- -------------------"
for I in xrange(1,6):
    print I,saha_E(chiI,5000,100.0,I)
```

Output:

```
For ionisation energies (in eV) of: [  6.11  11.87  50.91  67.27  84.34]

I  Fraction in stage I
-  -------------------
1 0.550032240423
2 0.449967183945
3 5.7563142596e-07
4 3.28485308138e-52
5 6.06361267175e-114
```

In [146]:
```
def saha_E(chiIarr, T, P_e=100.0, I=1):
    #Declare variables for all physical constants required
    #Here I have used some constants in units of eV and some in units of J - see note below
    keV = constants.value("Boltzmann constant in eV/K")
    kJ = constants.value("Boltzmann constant")
    heV = constants.value("Planck constant in eV s")
    hJ = constants.value("Planck constant")
    m_e = constants.value("electron mass") #in kg

    #Calculate all constant parts of the function and store in one variable
    consts = ((2*kJ*T)/(hJ**3*P_e))*(2*np.pi*m_e*kJ*T)**(3/2) #for T=5000K, consts = 1.17869E6

    #Step though ionisation energies and calculate partition function Z for each
    Zs = []
    for i in range(len(chiIarr)):
        Zs.append(partfunc_E(chiIarr, T)[i])

    #For each entry in the partition function array calculate the
    #ratio to the previous entry, and store in another array
    Z_ratios = []
    z_temp = 1
    for z in Zs:
        Z_ratios.append(z/z_temp)
        z_temp = z

    #Calculate the exponential part of the equation for each ionisation energy
    exps = []
    for chiI in chiIarr:
        exps.append(np.exp((-1*chiI)/(keV*T)))

    #Combine all the above into the full Saha equation,
    #returning an array with an entry for each ionisation state
    N_temp = 1
    N_ratios = []
    for i in range(len(chiIarr)):
        N_i = N_temp*Z_ratios[i]*consts*exps[i]
        N_ratios.append(N_i)
        N_temp = N_i

    return N_ratios[I-1]/np.sum(N_ratios)


chiI = np.array([6.11, 11.87, 50.91, 67.27, 84.34]) #eV
T = 5e3 #K
#T = 8.825e3 #K #This temperature gets close to the supplied distribution

print("For ionisation energies (in eV) of:")
print(chiI)
print("")
print("I  Fraction in stage I")
print("-  -------------------")

for I in range(1,6):
    print(I, saha_E(chiI, T, 100.0, I))
```

```
For ionisation energies (in eV) of:
[  6.11  11.87  50.91  67.27  84.34]

I  Fraction in stage I
-  -------------------
1 0.999998720736
2 1.27926351211e-06
3 7.29993420039e-52
4 1.3474665329e-113
5 1.54848994685e-192
```

These fractions are different to those supplied, which may be an issue of the units of the constants used. Having tried many different combinations of eV and J for energy, this is the closest distribution I can get without adding any arbitrary scaling factors. The plot below in Excercise 3.2 shows that the function works as it should. The distribution supplied occurs at a slightly higher temperature - around 8,825K rather than 5,000K. I have also noticed that for some reason, between Python 2.xx (used on lab computers) and Python 3.xx (installed on my computer), different results are obtained. This could be from differences in the Scipy library of constants, though inspection of the dictionary's

default values points to this not being the case. This has been frustrating, but I will continue below with the results obtained above (in Python 3.xx).

3.2

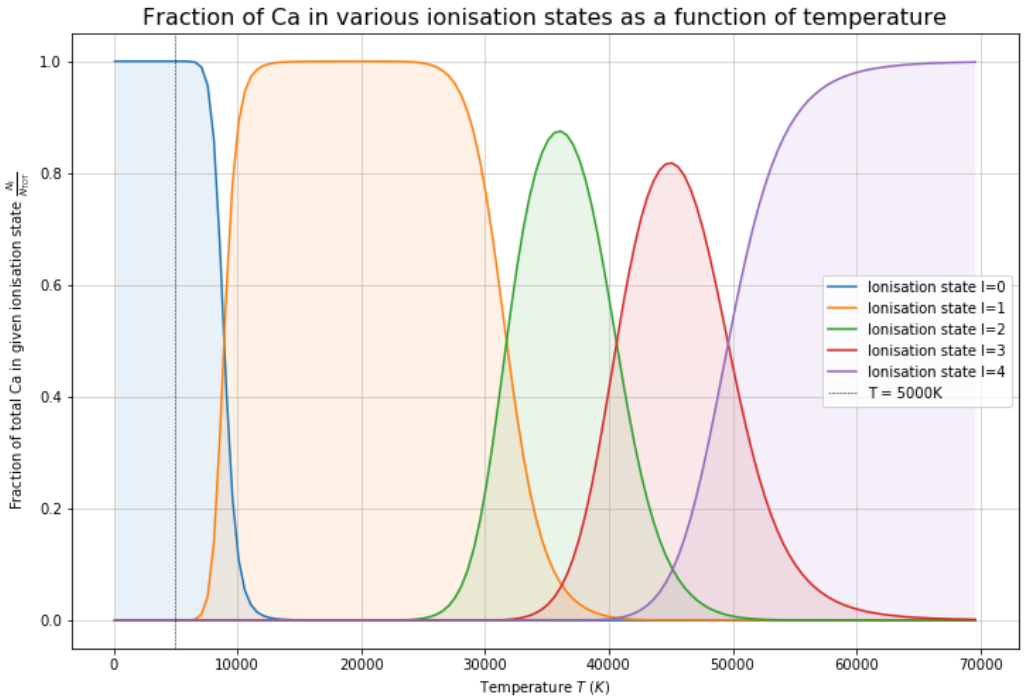
Explore what happens to the ionised fraction as a function of temperature and ionisation state. Do this in the same way as you did for the Boltzmann equation above. Answer the same questions:

- Is there a 'peak' temperature for a given energy level?
- How do the curves for different ionisation states compare?
- How does the curve behave at extreme temperatures? Why?

Additionally, consider the following:

- Explain why the Saha and Boltzmann distributions behave differently for increasing temperature.
- Speculate why ionization can fully deplete an ionization stage even though excitation puts only a few atoms in levels below the ionization level. Hint: what parameter in the Saha distribution can cause equality between high-level and next-ion population at a given temperature?

In [150]:
```
chiI = np.array([6.11, 11.87, 50.91, 67.27, 84.34]) #eV
Ts = np.arange(1e2, 7e4, 5e2) #K

fig, ax = plt.subplots(1, 1, figsize=(12,8))

#Step through each ionisation state and calculate Saha curves for each while stepping through the
 temperature array
for I in range(len(chiI)):
    y = []
    for T in Ts:
        y.append(saha_E(chiI, T, I=I+1))
    ax.plot(Ts, y, label="Ionisation state I=%.0f" % I)
    ax.fill_between(Ts, 0, y, alpha=0.1)

ax.set_title("Fraction of Ca in various ionisation states as a function of temperature", size=16)
ax.set_xlabel("Temperature $T$ ($K$)")
ax.set_ylabel("Fraction of total Ca in given ionisation state $\\frac{N_I}{N_{TOT}}$")
ax.axvline(x=5e3, color="k", ls="--", lw=0.5, label="T = 5000K")
ax.legend()
ax.grid(lw=0.5)
```



There is a peak temperature for each ionisation state. As temperatures increase, the energy kT $[eV]$ approaches equality with the ionisation energy χ_I towards the peak of ionisation state population. What we see with Calcium is that almost all atoms will be singly ionised between 9,000K and 30,000K, with more overlapping transitions occuring at higher temperatures. Below 9,000K almost all Ca will be neutral in an idealised system.

The Saha equation behaves differently to the Boltzmann equation at high temperatures as it takes into account quantum mechanical effects through incorporating a term for free electrons, while the Boltzmann equation does not consider such effects.

As atoms are ionised, the incresed free electron pressure works to fully deplete lower ionisation states.

Exercise 4: Combining Saha and Boltzmann Equations

Exercise 4: Combining Saha and Boltzmann Equations

4.1

Now combine the Saha and Boltzmann expressions to create a function that will give the fraction of ions (of given ionization potential) in a given energy level *and* ionization state for a given temperature and electron pressure. Combine them by simply multiplying your Saha and Boltzmann functions from above:

```
def SahaBoltz_E(chiI, T, Pe, I, i):
    # Compute Saha-Boltzmann populaton n_(r,s)/N for level r,s of E
    # input: ionisation energy Chi, temperature T, electron pressure,
    # ionization stage I, level i
    return saha_E(chiI,T,Pe,I) * boltz_E(chiI,T,I,i)
```

Test your combined function for a few different energy levels. An example is given below.

```
print "i  Fraction in level i of ion stage 1"
print "-  ---------------------------------"
for i in xrange(1,5):
    print i,SahaBoltz_E(chiI,5000,100.,1,i)
```

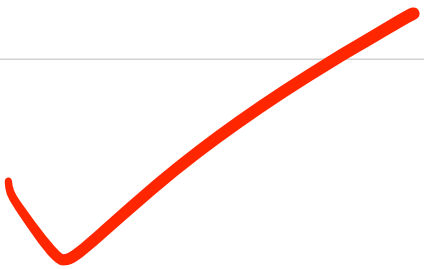
Output:

```
i  Fraction in level i of ion stage 1
-  ---------------------------------
0 0.496027435272
1 0.0487023542319
2 0.00478183088085
3 0.000469503105829
```

In [13]:
```
def SahaBoltz_E(chiI, T, Pe, I, i):
    #Simply return the Saha equation multiplied with the Boltzmann equation for each specified var
iable
    return saha_E(chiI, T, Pe, I) * boltz_E(chiI, T, I, i)
```

In [14]:
```
chiI = np.array([6.11, 11.87, 50.91, 67.27, 84.34])

print("i Fraction in level i of ion stage 1")
print("- ---------------------------------")
for i in range(1,5):
    print(i, SahaBoltz_E(chiI,5e3,100.,1,i))

print()

print("i Fraction in level i of ion stage 2")
print("- ---------------------------------")
for i in range(1,5):
    print(i, SahaBoltz_E(chiI,5e3,100.,2,i))

print()

print("i Fraction in level i of ion stage 3")
print("- ---------------------------------")
for i in range(1,5):
    print(i, SahaBoltz_E(chiI,5e3,100.,3,i))
```

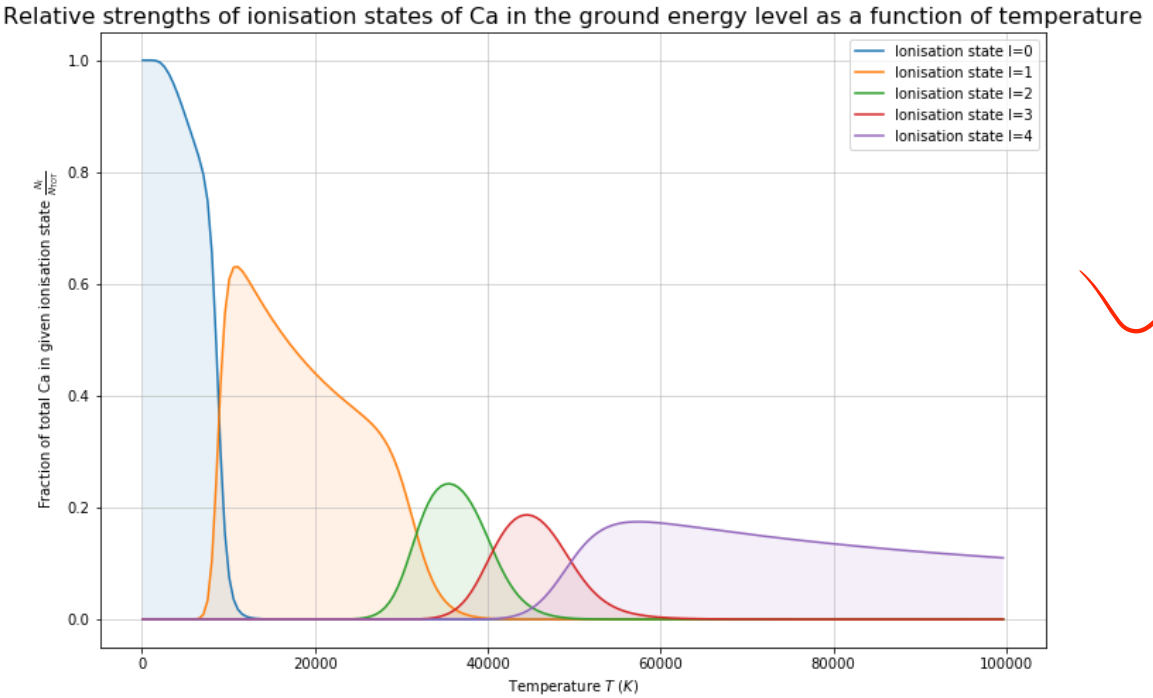
```
i Fraction in level i of ion stage 1
- ---------------------------------
1 0.0885443953519
2 0.00869370887722
3 0.000853589589058
4 8.3809476121e-05

i Fraction in level i of ion stage 2
- ---------------------------------
1 1.13271749116e-07
2 1.11215578006e-08
3 1.09196731646e-09
4 1.07214532495e-10

i Fraction in level i of ion stage 3
- ---------------------------------
1 6.46369029901e-53
2 6.34635783651e-54
3 6.23115525743e-55
4 6.11804389894e-56
```



4.2

Now make plots of relative strengths for your ion at different temperatures. Do this by calling your combined Saha-Boltzmann function inside a loop over different temperatures. Start with the ground state.

In [157]:
```
chiI = np.array([6.11, 11.87, 50.91, 67.27, 84.34]) #eV
Ts = np.arange(1e2, 10e4, 5e2) #K

fig, ax = plt.subplots(1, 1, figsize=(12,8))

#Step through ionisation states and compute the combined function for each
for I in range(len(chiI)):
    y = []
    for T in Ts:
        y.append(SahaBoltz_E(chiI, T, 100., I+1, 0))
    ax.plot(Ts, y, label="Ionisation state I=%.0f" % I) #Plot solid line
    ax.fill_between(Ts, 0, y, alpha=0.1) #Plot shaded region beneath curve

ax.set_title("Relative strengths of ionisation states of Ca in the ground energy level as a funct
ion of temperature", size=16)
ax.set_xlabel("Temperature $T$ ($K$)")
ax.set_ylabel("Fraction of total Ca in given ionisation state $\\frac{N_I}{N_{TOT}}$")
ax.legend()
ax.grid(lw=0.5)
```



At a given temperature, how many ionisation levels are present? Why might this the case?

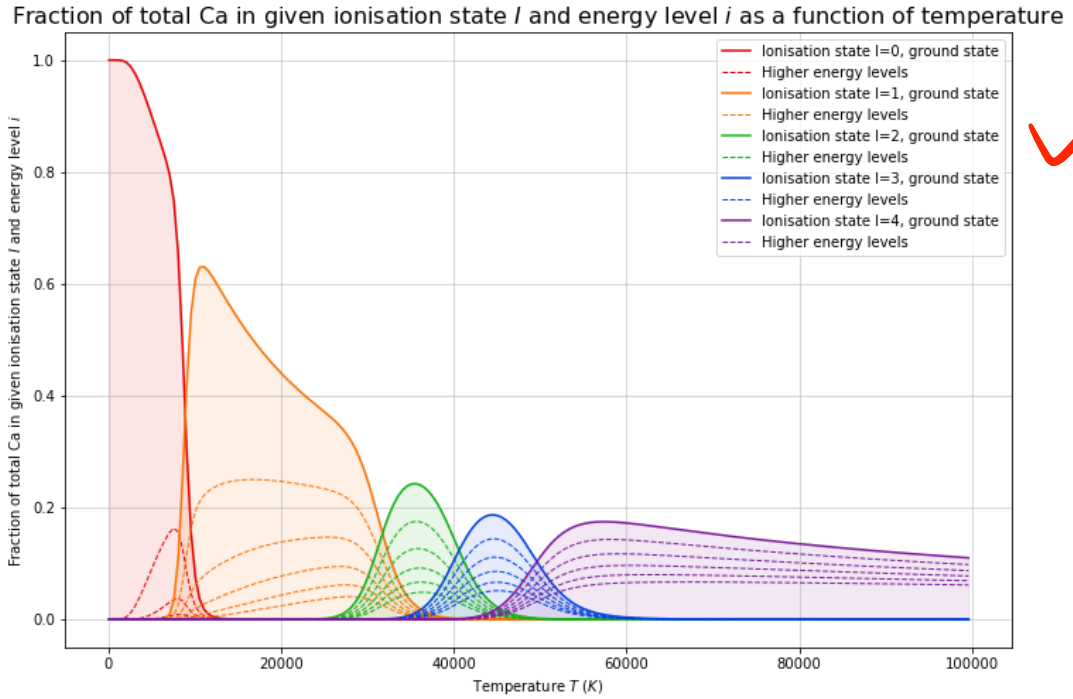
There are at most three ionisation levels present for any given temperature. This may be due to the spacing of the ionisation energies of the Calcium atom. As each subsequent ionisation state requires significantly more energy

4.3

Payne plotted her curves for the actual lower levels of the lines corresponding to observed wavelengths. The curves you just made were for the ground state. Explore the curves for higher energy state values of i. For example, plot each ionisation stage in a single colour, overplotting curves for different energy states. Try e.g. the first 4 levels.

In [161]:
```
chiI = np.array([6.11, 11.87, 50.91, 67.27, 84.34])
Ts = np.arange(1e2, 10e4, 5e2)

fig, ax = plt.subplots(1, 1, figsize=(12,8))

for I, col in zip(range(len(chiI)),["xkcd:red", "xkcd:orange", "xkcd:green", "xkcd:blue", "xkcd:p
urple"]):
    for i in range(6):
        y = []
        for T in Ts:
            y.append(SahaBoltz_E(chiI, T, 100., I+1, i))
        if i==0:
            ax.plot(Ts, y, color=col, label="Ionisation state I=%.0f, ground state" % I)
            ax.plot(0,0, color=col, ls="--", lw=1, label="Higher energy levels")
            ax.fill_between(Ts, 0, y, alpha=0.1, color=col)
        else:
            ax.plot(Ts, y, color=col, ls="--", lw=1)

ax.set_title("Fraction of total Ca in given ionisation state $I$ and energy level $i$ as a functi
on of temperature", size=16)
ax.set_xlabel("Temperature $T$ ($K$)")
ax.set_ylabel("Fraction of total Ca in given ionisation state $I$ and energy level $i$")
```

```
ax.legend()
ax.grid(lw=0.5)
```



- For a given ion, which energy states always have the higher population fraction?

The ground state is always the most populated state.

- Which expression, Saha or Boltzmann, sets the overall levels?

The overall shape of the distribution is set by the Boltzmann equation. The crossover at the boundaries of each ionisation state follows the Saha equation, where ionisation into that state shows a typically rapid transition.

- What does this tell you about the relative strengths for a series of lines (i.e. from different energy level transitions) of a given ion?

For a given ion, the relative strengths of energy levels follow a Boltzmann distribution as a function of temperature.

4.4

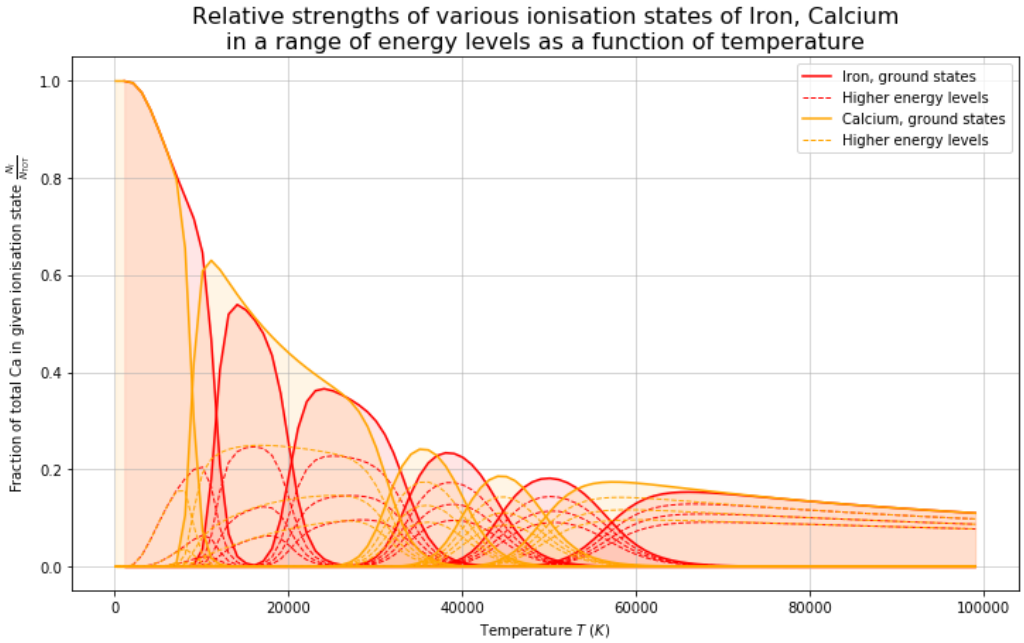
Try plotting some more curves for other elements. Look up the ionisation energies on NIST (http://physics.nist.gov/PhysRefData/ASD/ionEnergy.html). For example, common elements found in stars would be: Fe, Ca, C, N, O, Na, Mg, Si. Try a couple - too many and your plots may become unreadable. How could you use this information to learn about a star?

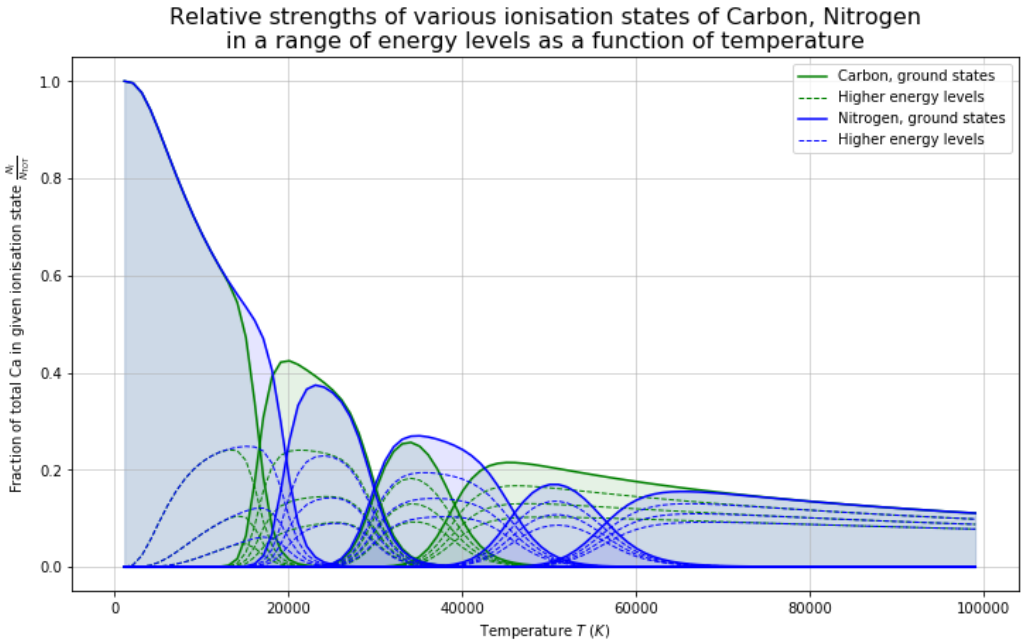
In [17]:
```
#For each element, I have listed ionisation energies up to 100eV
chiIFe = [7.90, 16.20, 30.65, 54.91, 75.00, 98.99] #Iron
chiICa = [6.11, 11.87, 50.91, 67.27, 84.34] #Calcium
chiIC = [11.26, 24.38, 47.89, 64.49] #Carbon
chiIN = [14.53, 29.60, 47.45, 77.47, 97.89] #Nitrogen
chiIO = [13.62, 35.12, 54.94, 77.41] #Oxygen
chiINa = [5.14, 47.29, 71.62, 98.34] #Sodium
chiIMg = [7.65, 15.04, 80.14] #Magnesium
chiISi = [8.15, 16.35, 33.49, 45.14] #Silicon

#Combine all element ionisation energy arrays into a master array, along with
#another for their names as strings, and an array of colours to use when plotting
chiIarr  = [chiIFe, chiICa, chiIC, chiIN, chiIO, chiINa, chiIMg, chiISi]
elements = ["Iron", "Calcium", "Carbon", "Nitrogen", "Oxygen", "Sodium", "Magnesium", "Silicon"]
colarr   = ["red", "orange", "green", "blue", "purple", "black", "grey", "brown"]
```

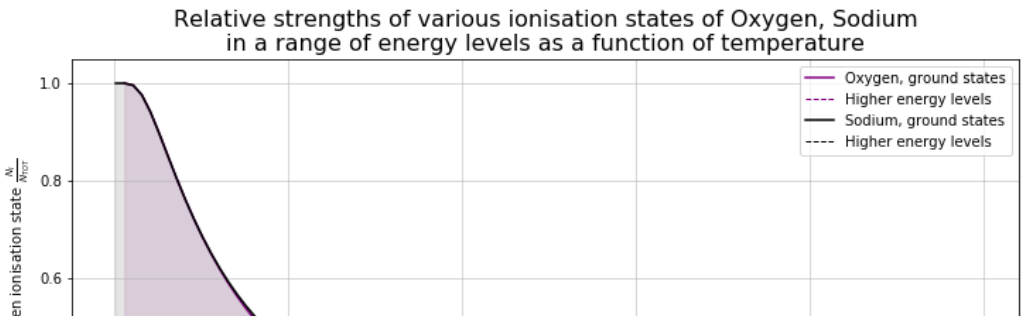
In [164]:
```
#Create a 'selection' variable to house a list of slices - pairs of elements to plot against one
 another
selection = [slice(0,2), slice(2,4), slice(4,6), slice(6,8)]
Ts = np.arange(1e2, 10e4, 1e3) #K

fig, ax = plt.subplots(len(selection), 1, figsize=(12,8*len(selection)))

for i in range(len(selection)):
    for chiI, col, element in zip(chiIarr[selection[i]], colarr[selection[i]], elements[selection
[i]]):

        #Plot empty curves to use for the legend
        ax[i].plot(0, 0, color=col, label=element+", ground states")
        ax[i].plot(0, 0, lw=1, ls="--", color=col, label="Higher energy levels")
```

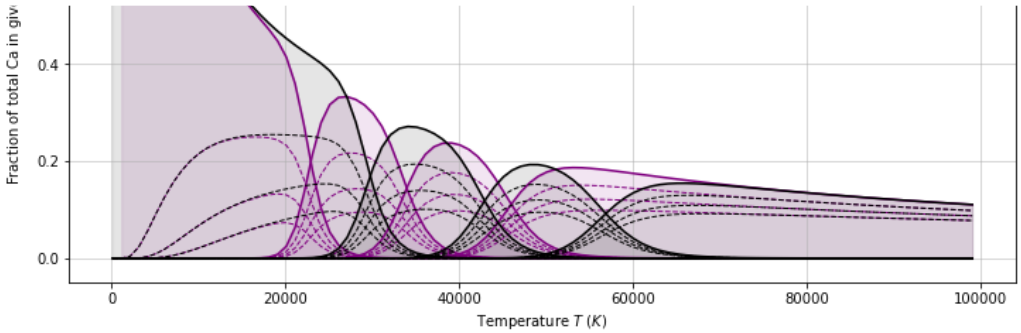
```
        for I in range(len(chiI)): #For each ionisation state
            for j in range(4): #and for each of the first 4 energy levels
                y = []
                for T in Ts: #Step though temperature array
                    y.append(SahaBoltz_E(chiI, T, 100., I+1, j)) #and build the curve
                if j==0: #For the ground state, plot solid line and a shaded area beneath
                    ax[i].plot(Ts, y, color=col)
                    ax[i].plot(0,0, color=col, ls="--", lw=1)
                    ax[i].fill_between(Ts, 0, y, alpha=0.1, color=col)
                else: #For all higher energy levels, plot only a dashed line
                    ax[i].plot(Ts, y, color=col, ls="--", lw=1)

    ax[i].set_title("Relative strengths of various ionisation states of "+", ".join(elements[sele
ction[i]])+"\nin a range of energy levels as a function of temperature", size=16)
    ax[i].set_xlabel("Temperature $T$ ($K$)")
    ax[i].set_ylabel("Fraction of total Ca in given ionisation state $\\frac{N_I}{N_{TOT}}$")
    ax[i].legend()
    ax[i].grid(lw=0.5)
```

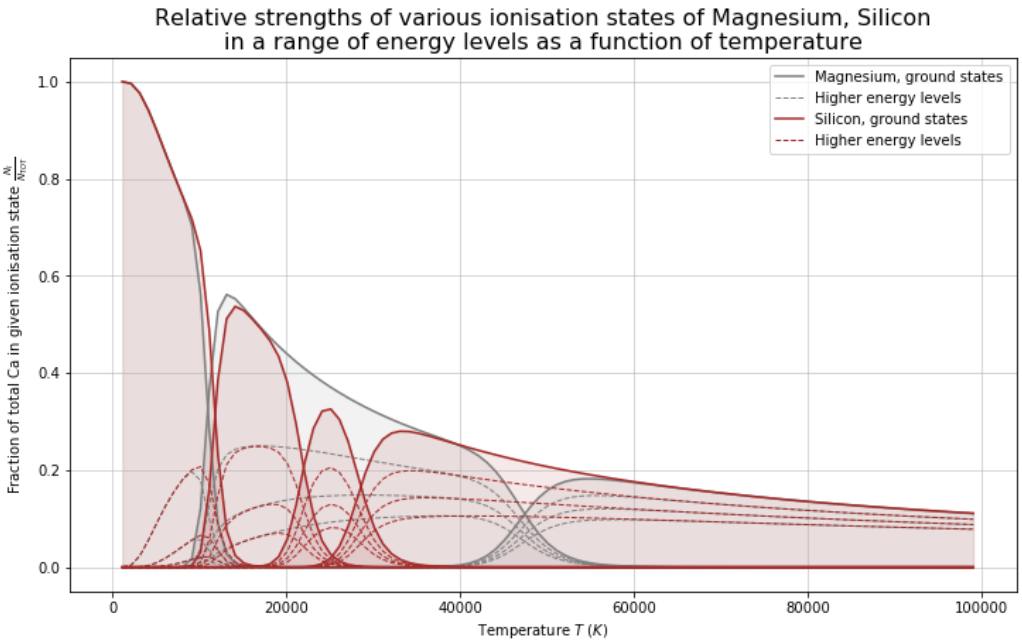
```
C:\Users\Jake\Anaconda3\lib\site-packages\ipykernel\__main__.py:36: RuntimeWarning: invalid value
encountered in double_scalars
```











By measuring the relative strengths of particular lines created by energy transitions of different atomic species in a stellar spectrum, the temperature of the star can be calculated. As the populations of these energy levels follow a Boltzmann distribution, and the assoiciated energy transitions correspond directly to the wavelength or frequency of spectral absorption or emission lines, this provides a direct way to measure temperature from the wavelength or frequency domain.

Fraunhofer line strengths and the curve of growth

Introduction

In the previous exercises you explored the physics behind stellar classification, using the Saha and Boltzmann equations to quantify partitioning of atoms over their ionisation and excitation states. The variations of spectral line strength along the main sequence were found to be primarily due to changes in temperature. Here we will address the question of how spectral lines form.

Exercise 5: Planck's Law

The counterpart to the Saha and Boltzmann distributions for electromagnetic radiation is the Planck law and its relatives, the Wien displacement law and the Stefan-Boltzmann law. Strictly, these hold in thermodynamic equilibrium (TE), which fortunately is a reasonable approximation in stellar photospheres. The Planck function specifies the radiation intensity emitted by a gas or a body in TE (a “black body”) as:

$$B_\lambda(T) = \frac{2hc^2}{\lambda^5}\frac{1}{e^{hc\,/\,\lambda kT} - 1}$$

where h is Planck's constant, c is the speed of light, k is Boltzmann's constant, λ is the wavelength and T is the temperature. The units of B_λ are W m Unknown character Unknown character Unknown character 2 m Unknown character Unknown character Unknown character 1 steradian Unknown character Unknown character Unknown character 1 (erg cm Unknown character Unknown character Unknown character 2 s Unknown character Unknown character Unknown character 1 cm Unknown character Unknown character Unknown character 1 steradian Unknown character Unknown character Unknown character 1 in cgs).

5.1 Write a function to evaluate the Planck function $B_\lambda(T)$ at a given temperature and wavelength. Some constants are defined below for convenience. Check it works: for a temperature of 5000 K at a wavelength of 5000 Angstrom, you should get:

```
print (planck(5000.,5000.e-10))
1.20908068988e+13
```

In [165]:
```
import scipy.constants

def Planck(T, lam):
    #Declare constants
```

```
    k = constants.value("Boltzmann constant")
    h = constants.value("Planck constant")
    c = constants.value("speed of light in vacuum")
    #Simply return the function
    return ((2*h*c**2)/(lam**5))*(1/(np.exp((h*c)/(lam*k*T))-1))
```

In [23]:
```
print("%.9e" % Planck(5e3, 5000.e-10))
```

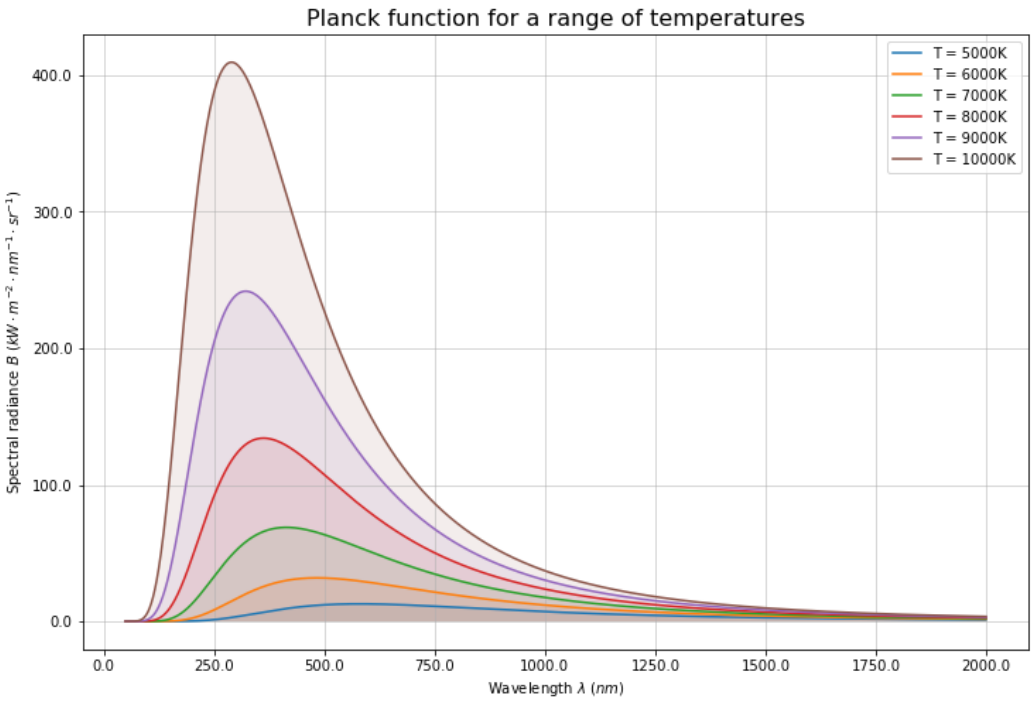
```
1.210716725e+13
```

5.2 Create an array of wavelengths, and use a for loop to make a plot showing the Planck function at different wavelengths for a few temperatures.

Study the Planck function properties. At all wavelengths $B_\lambda(T)$ increases with increasing temperature, but much faster (exponentially, Wien regime) at short wavelengths than at long wavelengths (linearly, Rayleigh-Jeans regime). The peak divides the two regimes and shifts to shorter wavelengths for higher temperature (Wien displacement law). The spectrum-integrated Planck function (area under the curve in a linear plot) increases steeply with temperature (Stefan-Boltzmann law).

In [166]:
```
lams = np.arange(50, 2000)*1.e-9 #m
Ts = np.arange(5e3, 11e3, 1e3) #K

fig, ax = plt.subplots(1, 1, figsize=(12,8))

for T in Ts: #Step through the range of temperatures
    ax.plot(lams, Planck(T, lams), label="T = %.0fK" % T) #Plot a curve for each temperature
    ax.fill_between(lams, 0, Planck(T, lams), alpha=0.1) #And fill beneath it with a light shade
 of the same colour

ax.set_title("Planck function for a range of temperatures", size=16)
ax.set_xticklabels(ax.get_xticks()*1e9) #Change x-ticks to be in nm
ax.set_xlabel("Wavelength $\lambda$ ($nm$)")
ax.set_yticklabels(ax.get_yticks()*1e-9*1e-3) #Change y-ticks to be in nm^-1 and kW
ax.set_ylabel("Spectral radiance $B$ ($kW \cdot m^{-2} \cdot nm^{-1} \cdot sr^{-1}$)")
ax.grid(lw=0.5)
ax.legend()
```

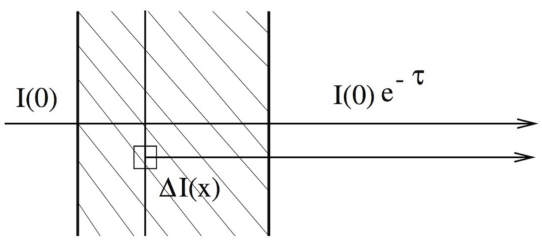
Out[166]: <matplotlib.legend.Legend at 0x2533cc9def0>



Exercise 6. Radiation through an isothermal layer

In [25]:
```
from IPython.display import Image
i3 = Image('figures/RadTrans.png',width=400)
i3
```

Out[25]:



0 x τ

Figure 1: Radiation through a layer. The incident intensity at the left is attenuated by absorption in the layer as specified by its total opaqueness τ (the "optical depth" of the layer). The internal production of radiation $\Delta I(x)$ in a thin sublayer with thickness Δx that is added to the beam locally is given by the product of the Planck function $B[T(x)]$ and the sublayer opaqueness $\Delta\tau(x)$; this contribution is then attenuated by the remainder of the layer.

The Planck function is the source function for the radiation produced by a gas in thermodynamic equilibrium at temperature T. The source function is the ratio of emission to absorption as a function of frequency. We need to specify the degree of absorption to compute the radiation produced by a gas of temperature T. Take the situation sketched in the figure above. A beam of radiation with intensity $I(0)$ passes through a layer (the hatched region) in which it is attenuated. The weakened intensity that emerges on the right is given by:

$$I = I(0)e^{-\tau}$$

in which the decay parameter τ is the optical depth of the layer (for now we will ignore its wavelength-dependence). τ specifies the attenuation by absorption in the layer. It is a dimensionless measure of the opaqueness and is called the "optical depth" because it measures how thick the layer is, not in meters but in terms of its effect on the passing radiation. Almost nothing comes through if $\tau \gg 1$ (i.e. if the layer is "optically thick") and almost everything comes through if $\tau \ll 1$ ("optically thin").

The next step is to add the radiation that originates within the layer itself. The contribution of $\Delta\tau$ to the intensity is $\Delta I = B_\lambda(T)\Delta\tau$. The scaling with $\Delta\tau$ comes in through Kirchhoff's law which says that a medium radiates better when it absorbs better (a "black" body radiates stronger than a "white" one). This local contribution at a location x within the layer is subsequently attenuated by the remainder of the layer. The total emergent intensity is the initial input intensity attenuated by the full layer, plus the integrated effect of the emission from within the layer:

$$I_\lambda = I_\lambda(0)e^{-\tau} + \int_0^\tau B_\lambda(T(x))e^{-(\tau-\tau(x))}d\tau(x)$$

For an isothermal layer (where T and therefore also $B_\lambda(T)$ are independent of x) simplifies to:

$$I_\lambda = I_\lambda(0)e^{-\tau} + B_\lambda(T)(1-e^{-\tau})$$

Exercises

Make plots of the emergent intensity I_λ against τ for $B_\lambda = 2.0$ and τ ranging between 0.01 and 10 in steps of 0.01. Plot a curve for each $I_\lambda(0)$ in $[4, 3, 2, 1, 0]$.

Hint: loop over the $I_\lambda(0)$ values as an outer loop, and evaluate the above expression for I_λ inside this loop for each value of τ.

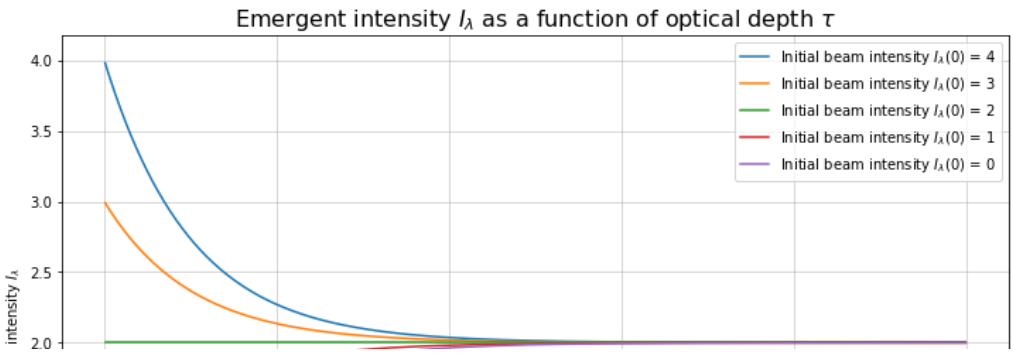
6.1 From your plots, what value does the emergent intensity, I_λ, tend towards?

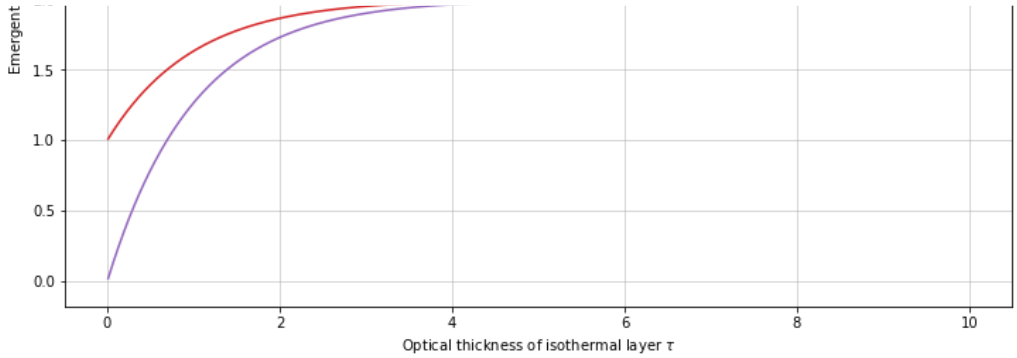
6.2 How does I_λ depend on τ for $\tau \ll 1$ when $I_\lambda(0) = 0$ (plot the logarithm of your x and y values to study the behaviour at small τ)? And when $I_\lambda(0) > B_\lambda$? Such a layer is called "optically thin", why?

6.3 A layer is called "optically thick" when it has $\tau \gg 1$. Why? The emergent intensity becomes independent of τ for large τ. Can you explain why this is so in physical terms?

In [170]:
```
B_lam = 2.0
taus = np.arange(0.01, 10, 0.01)
I_lam_zeros = np.arange(4, -1, -1)

fig, ax = plt.subplots(1, 1, figsize=(12,8))

for I_lam_zero in I_lam_zeros:
    ax.plot(taus, I_lam_zero*np.exp(-taus) + B_lam*(1-np.exp(-taus)), label="Initial beam intensi
ty $I_{\lambda}(0)$ = %.0f" % I_lam_zero)

ax.set_title("Emergent intensity $I_{\lambda}$ as a function of optical depth $\\tau$", size=16)
ax.set_xlabel("Optical thickness of isothermal layer $\\tau$")
ax.set_ylabel("Emergent intensity $I_{\lambda}$")
ax.grid(lw=0.5)
ax.legend()
```

Out[170]: <matplotlib.legend.Legend at 0x25338480d68>





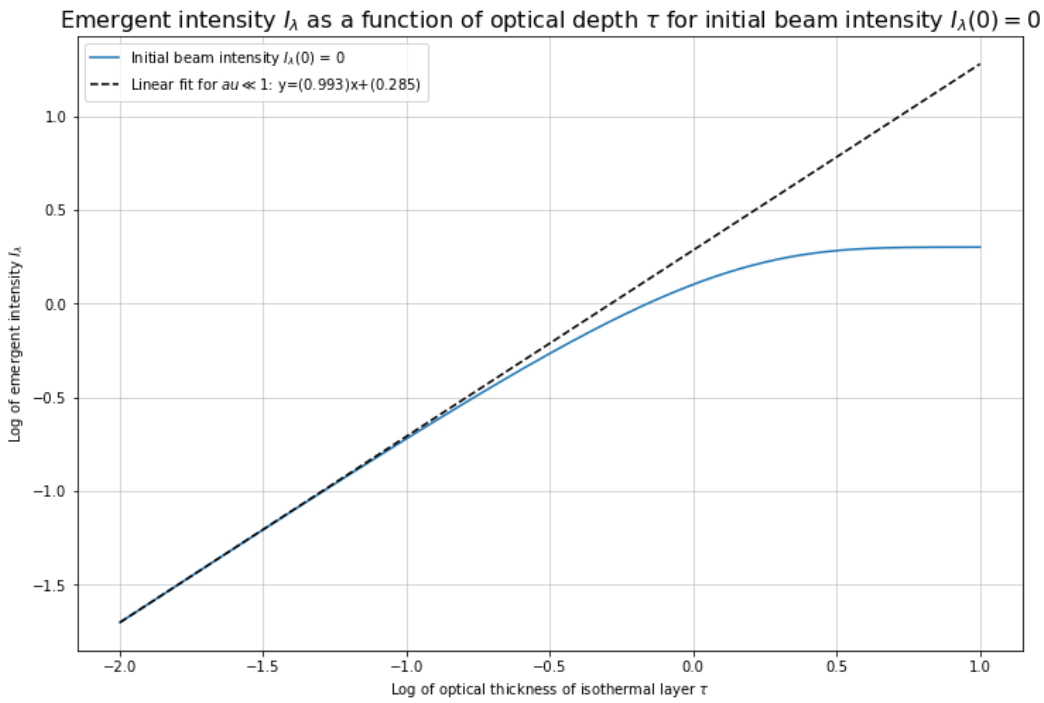
6.1 From your plots, what value does the emergent intensity, I_λ, tend towards?

For all initial beam intensities, the emergent intensity tends towards the value of the function that describes the local emission of the material, in this case 2.0, but in reality for a cloud at thermodynamic equilibrum the Planck function.

6.2 How does I_λ depend on τ for $\tau \ll 1$ when $I_\lambda(0) = 0$ (plot the logarithm of your x and y values to study the behaviour at small τ)? And when $I_\lambda(0) > B_\lambda$? Such a layer is called “optically thin”, why?

In [27]:
```
B_lam = 2.0
taus = np.arange(0.01, 10, 0.001)
I_lam_zeros = [0]

fig, ax = plt.subplots(1, 1, figsize=(12,8))

for I_lam_zero in I_lam_zeros:
    ax.plot(np.log10(taus), np.log10(I_lam_zero*np.exp(-taus) + B_lam*(1-np.exp(-taus))), label="I
nitial beam intensity $I_{\lambda}(0)$ = %.0f" % I_lam_zero)

xfit = np.log10(taus)
cut = 10
m, b = np.polyfit(xfit[:cut], np.log10(I_lam_zero*np.exp(-taus[:cut]) + B_lam*(1-np.exp(-taus[:cut
]))), deg=1)
yfit = m*xfit + b
ax.plot(xfit, yfit, ls="--", color="k", label="Linear fit for $\tau \ll 1$: y=(%.3f)x+(%.3f)" % (m
, b))

ax.set_title("Emergent intensity $I_{\lambda}$ as a function of optical depth $\\tau$ for initial
 beam intensity $I_{\lambda}(0) = 0$", size=16)
ax.set_xlabel("Log of optical thickness of isothermal layer $\\tau$")
ax.set_ylabel("Log of emergent intensity $I_{\lambda}$")
ax.grid(lw=0.5)
ax.legend()
```

Out[27]: <matplotlib.legend.Legend at 0x2532bf1c0b8>



In [28]:
```
print(10**b)
print(m)
```

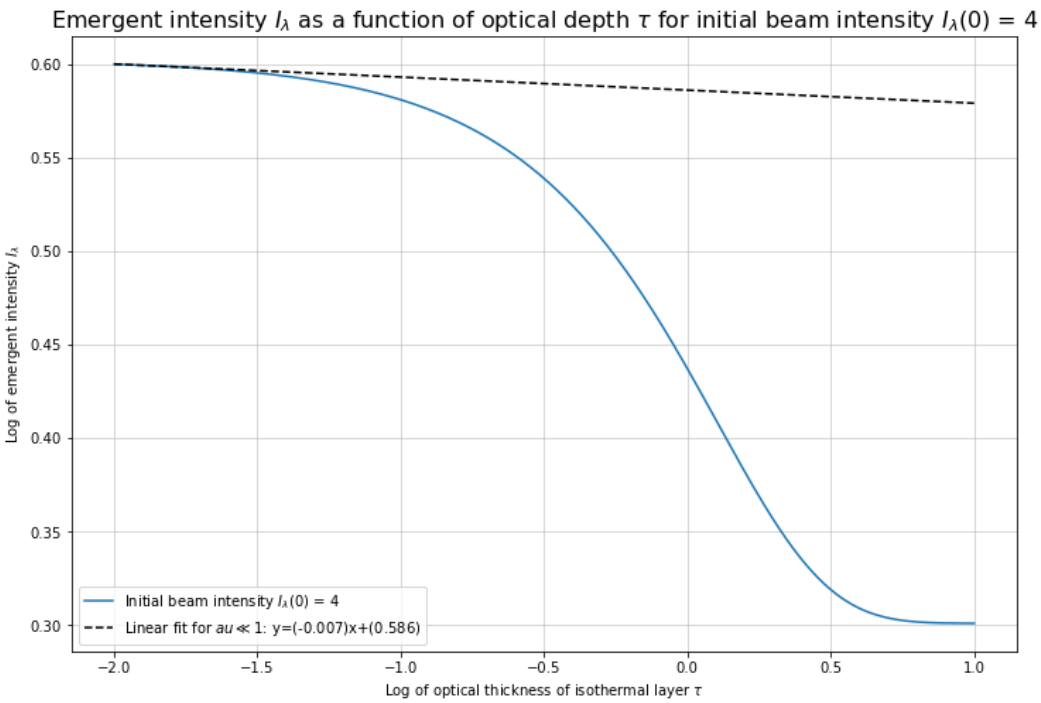
```
1.92729114403
```

```
0.992996903407
```

For the case where $I_\lambda(0) = 0$, I_λ is linear on a log-log plot for values of $\tau \ll 1$, meaning that I_λ depends on τ as follows: $I_\lambda = 10^b \cdot \tau^m$, so $I_\lambda \approx 1.927 \cdot \tau^{0.993}$, which as $\tau \to 0$ approaches: $I_\lambda = B_\lambda \cdot \tau$

In [29]:
```
B_lam = 2.0
taus = np.arange(0.01, 10, 0.001)
I_lam_zeros = [4]

fig, ax = plt.subplots(1, 1, figsize=(12,8))

for I_lam_zero in I_lam_zeros:
    ax.plot(np.log10(taus), np.log10(I_lam_zero*np.exp(-taus) + B_lam*(1-np.exp(-taus))), label="I
nitial beam intensity $I_{\lambda}(0)$ = %.0f" % I_lam_zero)

xfit = np.log10(taus)
cut = 10
m, b = np.polyfit(xfit[:cut], np.log10(I_lam_zero*np.exp(-taus[:cut]) + B_lam*(1-np.exp(-taus[:cut
]))), deg=1)
yfit = m*xfit + b
ax.plot(xfit, yfit, ls="--", color="k", label="Linear fit for $\tau \ll 1$: y=(%.3f)x+(%.3f)" % (m
, b))

ax.set_title("Emergent intensity $I_{\lambda}$ as a function of optical depth $\\tau$ for initial
 beam intensity $I_{\lambda}(0)$ = %.0f" % I_lam_zeros[0], size=16)
ax.set_xlabel("Log of optical thickness of isothermal layer $\\tau$")
ax.set_ylabel("Log of emergent intensity $I_{\lambda}$")
ax.grid(lw=0.5)
ax.legend()
```

Out[29]:
```
<matplotlib.legend.Legend at 0x25329812860>
```



In [30]:
```
print(10**b)
```

```
3.85520451199
```

For the case where $I_\lambda(0) > B_\lambda$, I_λ is close to a constant on a log-log plot for values of $\tau \ll 1$, meaning that I_λ does not depend on τ: $log_{10} I_\lambda = b$

so $I_\lambda = 10^b$, and so $I_\lambda \approx 3.86$, which as $\tau \to 0$, approaches: $I_\lambda = I_\lambda(0)$

Layers with $\tau \ll 1$ are called "optically thin" as the intervening layer material itself has very little impact on the initial beam passing through it.

6.3 A layer is called “optically thick” when it has $\tau \gg 1$. Why? The emergent intensity becomes independent of τ for large τ. Can you explain why this is so in physical terms?

Optically thick layers are so called because they impact the initial beam passing through them to the point where all initial intensity $I_\lambda(0)$ has been absorbed and re-emitted by the layer itself, with the resulting emergent intensity depending only on the properties of the layer and not its thickness nor the initial beam.

Exercise 7. Spectral lines from a solar reversing layer

In [31]:
```
i = Image('figures/RevLayer.png',width=400)
i
```

Out[31]:

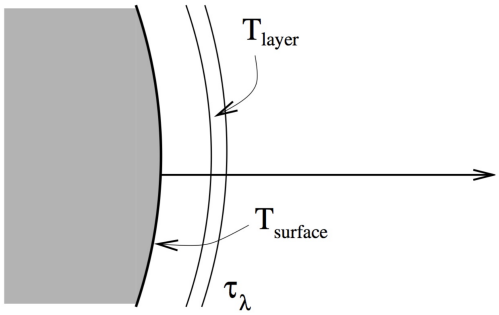


Figure 2. The Schuster-Schwarzschild or reversing-layer model. The stellar surface radiates an intensity given by $B\lambda(Tsurface)$. *The shell around the surface only affects this radiation at the wavelengths where atoms provide a bound-bound transition between two discrete energy levels. These spectral line transitions cause attenuation* $\tau\lambda$. *The layer has temperature* $Tlayer$ *and gives a thermal contribution*
$B\lambda(Tlayer)(1\text{Unknown character}\text{Unknown character}\text{Unknown character}e^{\text{Unknown character}\text{Unknown character}\text{Unknown character}\tau\lambda})$
as described below._

We now apply the above result for an isothermal layer to a simple model in which the Fraunhofer lines in the solar spectrum are explained by a “reversing layer”. Cecilia Payne had this model in mind when she plotted her Saha-Boltzmann population curves, proposing that the local density of the line-causing atoms and ions within stellar reversing layers.

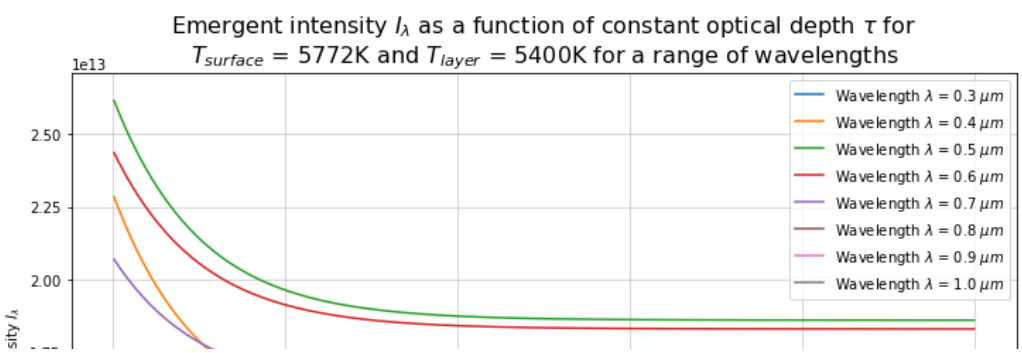
Schuster-Schwarzschild model. The basic assumptions are that the continuous radiation, without spectral lines, is emitted by the stellar surface and irradiates a separate layer with the intensity: $I_\lambda(0) = B_\lambda(T_{surface})$, and that this layer sits as a shell around the star and causes attenuation and local emission only at the wavelengths of spectral lines (see figure above). Thus, the shell is assumed to be made up exclusively by line-causing atoms or ions.

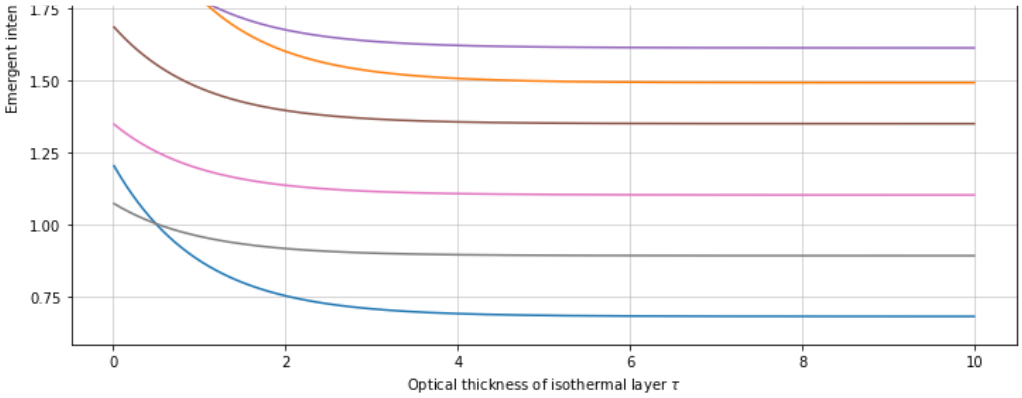
The star is optically thick (any star is optically thick!) so that its surface radiates with the $\tau \gg 1$ solution $I_\lambda = B_\lambda(T_{surface})$ given above. The shell, however, may be optically thin or thick at the line wavelength depending on the abundance of the atoms in the appropriate atomic state. The line-causing atoms in the shell have temperature T_{layer} so that the local production of radiation in the layer at the line wavelengths is given by $B_\lambda(T_{layer})\Delta\tau(x)$. The emergent radiation at the line wavelengths is then given by:

$$I_\lambda = B_\lambda(T_{surface})e^{-\tau_\lambda} + B_\lambda(T_{layer})(1 - e^{-\tau_\lambda})$$

In [32]:
```
lams = np.arange(300, 1100, 100)*1.e-9
T_surface = 5772
T_layer = 5400

B_lam_T = Planck(T_surface, lams)

B_lam_T_layer = Planck(T_layer, lams)
taus = np.arange(0.01, 10, 0.01)

fig, ax = plt.subplots(1, 1, figsize=(12,8))

for lam, I_lam_zero, B_lam in zip(lams, B_lam_T, B_lam_T_layer):
    ax.plot(taus, I_lam_zero*np.exp(-taus) + B_lam*(1-np.exp(-taus)), label="Wavelength $\lambda$
 = %.1f $\mu m$" % (lam*1e6))

ax.set_title("Emergent intensity $I_{\lambda}$ as a function of constant optical depth $\\tau$ for
 \n$T_{surface}$ = %.0fK and $T_{layer}$ = %.0fK for a range of wavelengths" % (T_surface, T_layer
), size=16)
ax.set_xlabel("Optical thickness of isothermal layer $\\tau$")
ax.set_ylabel("Emergent intensity $I_{\lambda}$")
ax.grid(lw=0.5)
ax.legend()
```

Out[32]: <matplotlib.legend.Legend at 0x253294ceda0>





Voigt profile. The subscript on the optical depth, τ_λ, is because it varies over the spectral line. When atoms absorb or emit a photon at the energy at which the valence electron may jump between two bound energy levels, the effect is not limited to an infinitely sharp delta function at λ with $hc\,/\,\lambda = \Delta E$ but it is a little bit spread out in wavelength. An obvious cause for such "line broadening" consists of the Doppler shifts given by individual atoms due to their thermal motions. Other broadening is due to Coulomb interactions with neighboring particles. This broadening distribution is described by:

$$\tau_\lambda(u) = \tau_\lambda(0)\,V(a,\,u)$$

where the Voigt function $V(a,\,u)$ is defined as:

$$V(a,\,u) = \frac{1}{\Delta\lambda_D\sqrt{\pi}}\,\frac{a}{\pi}\int_{-\infty}^{+\infty}\frac{e^{-y^2}}{(u-y)^2+a^2}\,dy\,.$$

Here u specifies the wavelength separation from the center of the line at $\lambda = \lambda_0$ in dimensionless units $u = (\lambda - \lambda_0)\,/\,\Delta\lambda_D$, where $\Delta\lambda_D$ is the "Doppler width", is:

$$\Delta\lambda_D \equiv \frac{\lambda}{c}\sqrt{\frac{2kT}{m}}$$

with m the mass of the line-causing particles (for example iron with $m_{\mathrm{Fe}}\mathrm{Unknown\ character}\mathrm{Unknown\ character}\mathrm{Unknown\ character}56mH \sim 9.3\times10^{\mathrm{Unknown\ character}\mathrm{Unknown\ character}\mathrm{Unknown\ character}26}$ kg). We will consider a range of $u \approx \pm5\text{–}10$.

The parameter a in the Voigt function specifies the broadening by Coulomb disturbances ("damping"). Stellar atmospheres typically have a ≈ 0.01 – 0.5.

The Voigt function represents the convolution (smearing) of a Gaussian profile with a Lorentz profile and therefore has a Gaussian shape close to line center ($u = 0$) due to the thermal Doppler shifts ("Doppler core") and extended Lorentzian wings due to disturbances by other particles ("damping wings"). A reasonable approximation is obtained by taking the sum rather than the convolution of the two profiles:
$${\large V(a,u) \approx \frac{1}{\Delta\lambda_D \sqrt{\pi}} \biggl[{\large e^{-u^2}} + \frac{a}{\sqrt{\pi}u^2} \biggl]}$$

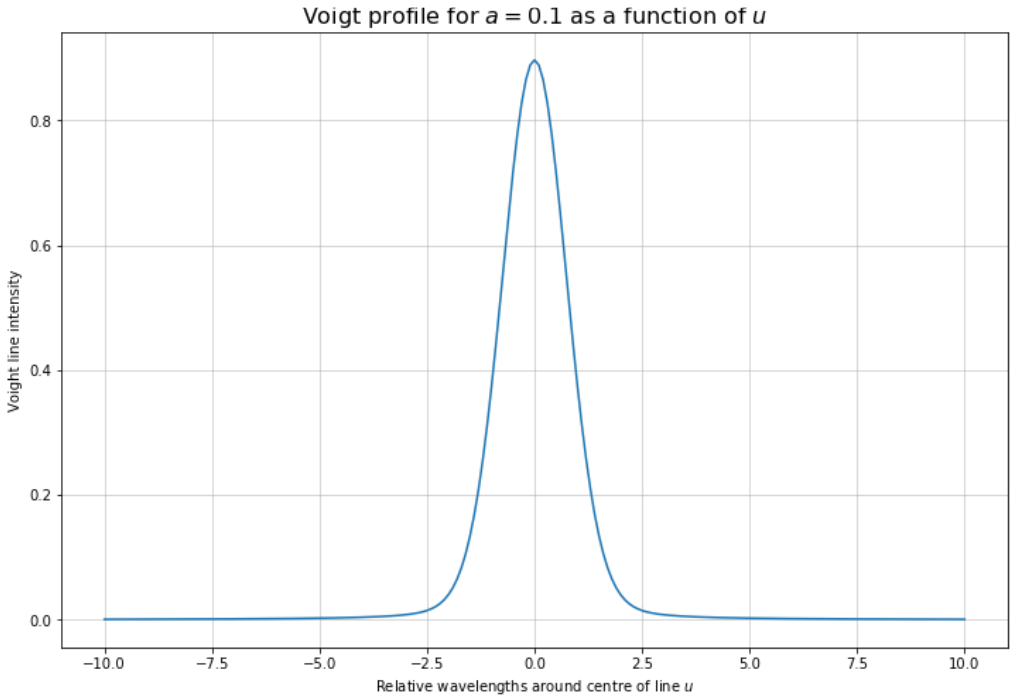
We could code up the above approximation, but it has some "issues" around $u = 0$. You can experiment with that if you like, but instead, we will use the following efficient implementation of the Voigt function (if you want to learn more about this, see this link (https://www.dropbox.com/s/u3czcyd2clgljvp/Voigt_complex_error_functions.pdf?dl=1)):

In [33]:
```
from sympy import mpmath as mp
def Voigt(a, u):
    # Calculate the voigt function H(a,u).
    with mp.workdps(20):
        z = mp.mpc(u, a)
        result = mp.exp(-z*z) * mp.erfc(-1j*z)
    return float(result.real)
```

Exercises

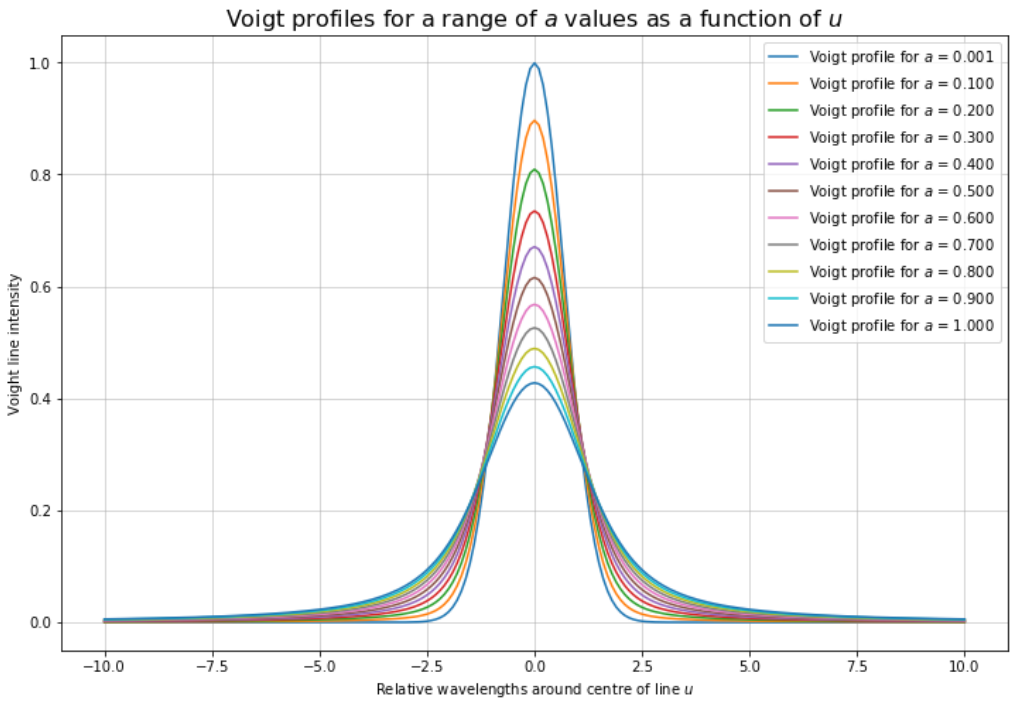
7.1 Plot the Voigt function against u from $u = \mathrm{Unknown\ character}\mathrm{Unknown\ character}\mathrm{Unknown\ character}10$ to $u = +10$ (these are the 'relative' wavelengths around the line centre) for $a = 0.1$

In [35]:
```
fig, ax = plt.subplots(1, 1, figsize=(12,8))

u_array = np.arange(-10., 10.1, 0.1)
a = 0.1

y = []
for u in u_array:
    y.append(Voigt(a, u))
ax.plot(u_array, y, label="Voigt profile for $a$ = %.1f" % a)

ax.set_title("Voigt profile for $a=0.1$ as a function of $u$", size=16)
ax.set_xlabel("Relative wavelengths around centre of line $u$")
ax.set_ylabel("Voight line intensity")
ax.grid(lw=0.5)
```



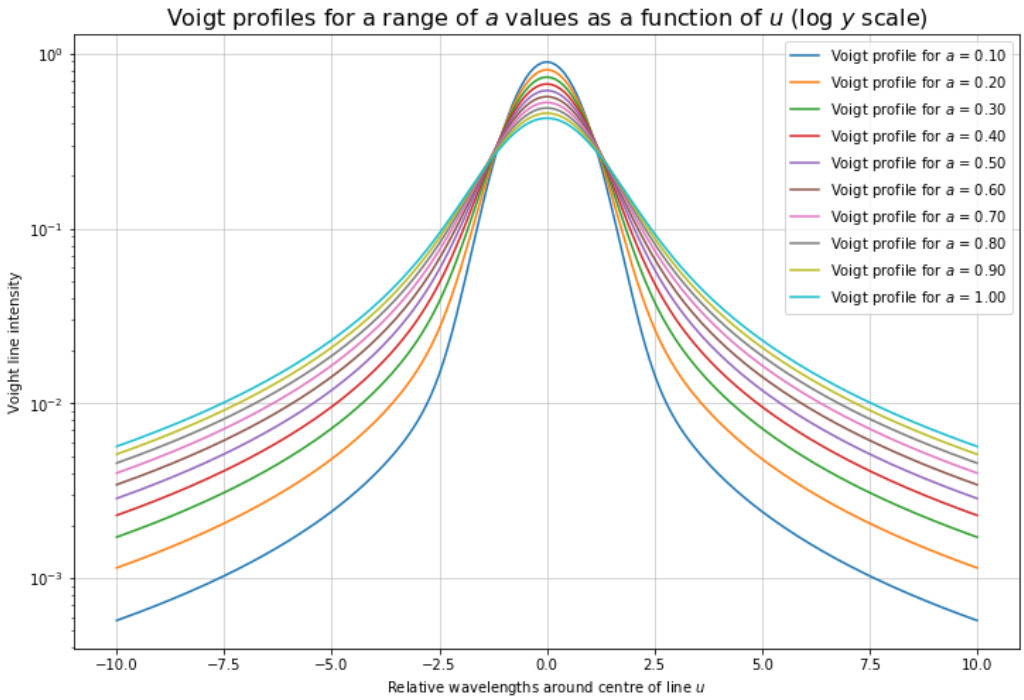
7.2 Vary a between $a = 1$ and $a = 0.001$ to see the effect of this parameter. Also investigate the logarithm of $V(a, u)$ to inspect the far wings of the profile. Use the approximation expression above to explain what you see.

In [36]:
```
fig, ax = plt.subplots(1, 1, figsize=(12,8))

u_array = np.arange(-10., 10.1, 0.1)
a_array = np.append(0.001, np.arange(0.1, 1.1, 0.1))

for a in a_array:
    y = []
    for u in u_array:
        y.append(Voigt(a, u))
    ax.plot(u_array, y, label="Voigt profile for $a$ = %.3f" % a)

ax.set_title("Voigt profiles for a range of $a$ values as a function of $u$", size=16)
ax.set_xlabel("Relative wavelengths around centre of line $u$")
ax.set_ylabel("Voight line intensity")
ax.legend()
ax.grid(lw=0.5)
```



In [37]:
```
fig, ax = plt.subplots(1, 1, figsize=(12,8))

u_array = np.arange(-10., 10.1, 0.1)
a_array = np.arange(0.1, 1.1, 0.1)

for a in a_array:
```

```
    y = []
    for u in u_array:
        y.append(Voigt(a, u))
    ax.plot(u_array, y, label="Voigt profile for $a$ = %.2f" % a)

ax.set_title("Voigt profiles for a range of $a$ values as a function of $u$ (log $y$ scale)", size
=16)
ax.set_xlabel("Relative wavelengths around centre of line $u$")
ax.set_ylabel("Voight line intensity")
ax.legend()
ax.set_yscale("log")
ax.grid(lw=0.5)
```



7.3 Make a plot comparing the Voigt function with the corresponding Gaussian and Lorentz profiles. What do you notice about the shape of the Voigt profile compared to the Gaussian and Lorentz? I.e. where/how are they similar, and where/how do they differ?

You will need to create functions for these other profiles, which should depend on the same variables $(a,\ u)$. For the Gaussian, use:

$$G(a,\ u) = \frac{1}{\sqrt{\pi}\alpha} e^{-(u\ /\ \alpha)^2},$$

where

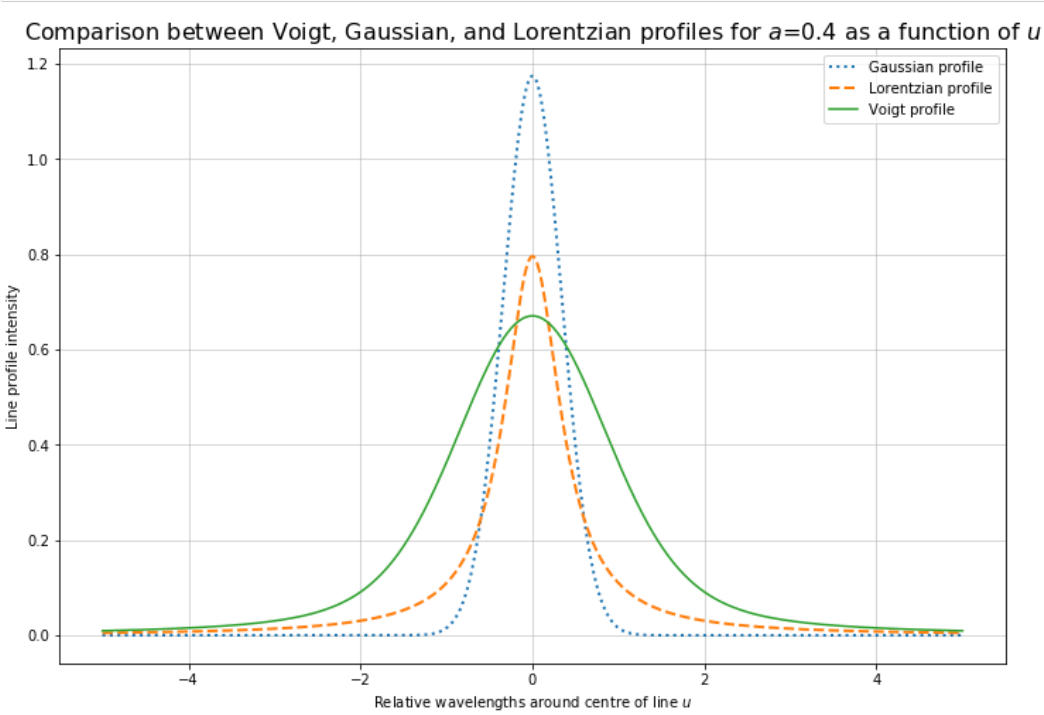
$$\alpha = \frac{a}{\sqrt{\ln 2}}.$$

For the Lorentzian, use:

$$L(a,\ u) = \frac{1}{\pi a} \frac{a^2}{u^2 + a^2}.$$

In [38]:
```
def Gaussian(a, u):
    alpha = a/(np.sqrt(np.log(2)))
    return (1/(np.sqrt(np.pi)*alpha))*np.exp(-(u/alpha)**2)

def Lorentzian(a, u):
    return (1/(np.pi*a))*((a**2)/(u**2 + a**2))
```

In [171]:
```
fig, ax = plt.subplots(1, 1, figsize=(12,8))

u_array = np.arange(-5., 5.01, 0.01)
a = 0.4

ax.plot(u_array, Gaussian(a, u_array), ls=":", lw=2, label="Gaussian profile")
ax.plot(u_array, Lorentzian(a, u_array), ls="--", lw=2, label="Lorentzian profile")

y = []
for u in u_array:
    y.append(Voigt(a, u))
ax.plot(u_array, y, label="Voigt profile")

ax.set_title("Comparison between Voigt, Gaussian, and Lorentzian profiles for $a$=%.1f as a funct
ion of $u$" % a, size=16)
ax.set_xlabel("Relative wavelengths around centre of line $u$")
ax.set_ylabel("Line profile intensity")
ax.legend()
ax.grid(lw=0.5)
```



Compared to the Gaussian and Lorentzian profiles, the Voigt shows much wider "wings" away from the line center. This results in a much lower peak as the area beneath all three curves is normalised. The Gaussian function shows the fastest decay away from the center, with the Lorentzian showing a slower decay but a lower peak at the center.

Exercise 8. Emergent line profiles.

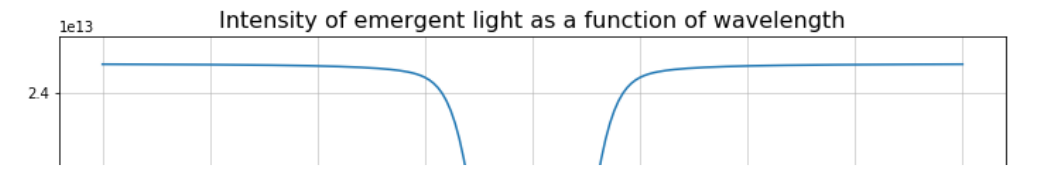
You can now compute and plot a stellar spectral line profile by combining the transfer equation and the Voigt profile. Again use the dimensionless u units for the 'wavelength' scale.

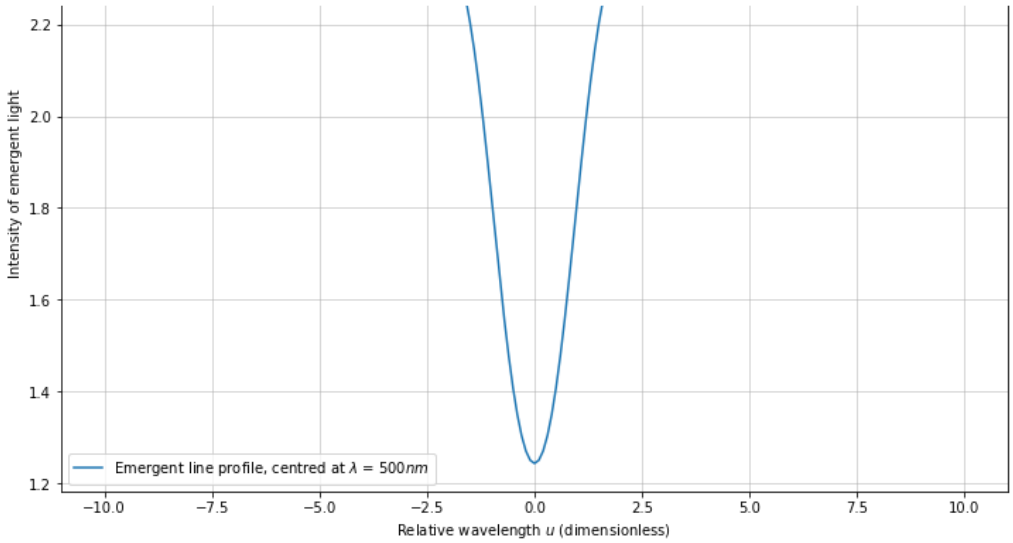
Exercises

8.1 Write a Python sequence that computes Schuster-Schwarzschild line profiles. Take $T_{surface} = 5700$ K, $T_{layer} = 4200$ K, $a = 0.1$, $\lambda = 5000$ Å. These values are good choices for the solar photosphere as seen in the optical part of the spectrum. First plot a profile I against u for $\tau = 1$. At each 'wavelength', u, the optical depth τ will first be weighted by the Voigt profile, such that τ is a maximum at the line centre ($u = 0$).

In [138]:
```
fig, ax = plt.subplots(1, 1, figsize=(12,8))

lams = [500*1e-9] #500nm/5000 Angstroms
T_surface = 5700
B_lam_T_surface = Planck(T_surface, lam)
T_layer = 4200
B_lam_T_layer = Planck(T_layer, lam)
tau0 = 1
a = 0.1
u_array = np.arange(-10, 10.1, 0.1)

for lam in lams:
    x = []
    y = []
    for u in u_array:
        tau = tau0*Voigt(a, u)
        xpoint = u#+lam*1e9
        x.append(xpoint)
        ypoint = B_lam_T_surface*np.exp(-tau) + B_lam_T_layer*(1-np.exp(-tau))
        y.append(ypoint)

    ax.plot(x, y, label="Emergent line profile, centred at $\lambda$ = %.0f$nm$" % (lam*1e9))

ax.set_title("Intensity of emergent light as a function of wavelength", size=16)
ax.set_xlabel("Relative wavelength $u$ (dimensionless)")
ax.set_ylabel("Intensity of emergent light")
ax.grid(lw=0.5)
ax.legend()
```

Out[138]: <matplotlib.legend.Legend at 0x2533c2a5358>

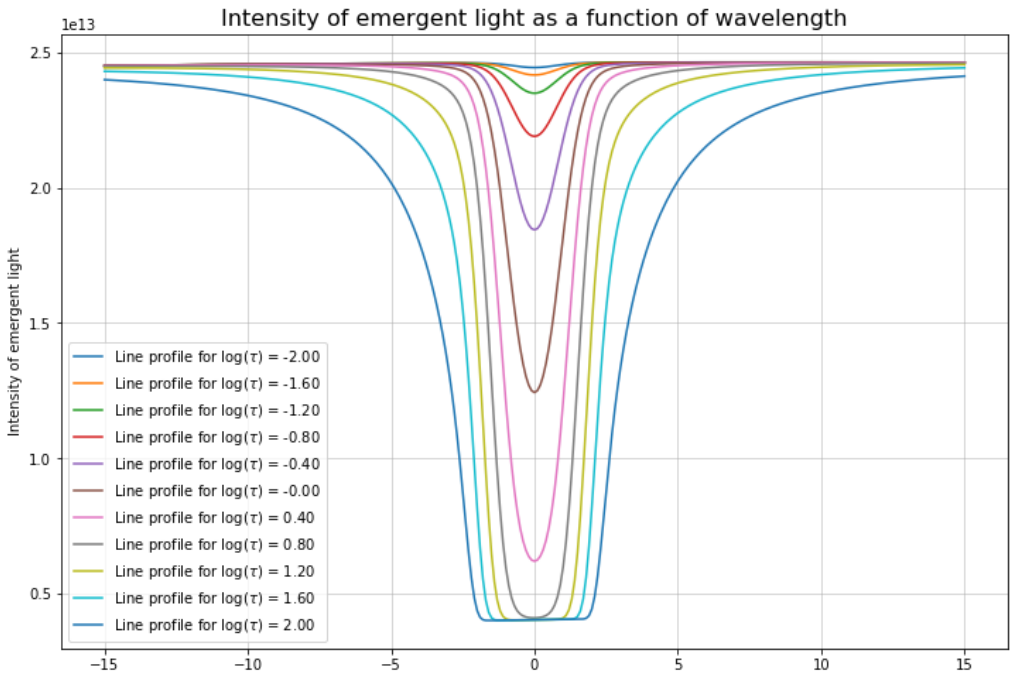




8.2 Study the appearance of the line in the spectrum as a function of τ over the range $\log\tau =$ Unknown character Unknown character Unknown character 2 to $\log\tau = 2$.

In [175]:
```
fig, ax = plt.subplots(1, 1, figsize=(12,8))

lams = [500*1e-9] #500nm/5000 Angstroms
T_surface = 5700
T_layer = 4200
logtau0s = np.arange(-2, 2.4, 0.4)
a = 0.1
u_array = np.arange(-15, 15.1, 0.1)

for logtau0 in logtau0s:
    for lam in lams:

        x = []
        y = []
        for u in u_array:
            tau = (10**logtau0)*Voigt(a, u)
            xpoint = u#+lam*1e9
            x.append(xpoint)
            ypoint =  Planck(T_surface, lam+u*1e-9)*np.exp(-tau) + Planck(T_layer, lam+u*1e-9)*(1
-np.exp(-tau))
            y.append(ypoint)

        ax.plot(x, y, label="Line profile for log($\\tau$) = %.2f" % logtau0)

ax.set_title("Intensity of emergent light as a function of wavelength", size=16)
ax.set_xlabel("$u$, relative wavelength (dimensionless)")
ax.set_ylabel("Intensity of emergent light")
ax.grid(lw=0.5)
ax.legend()
```

Out[175]:
```
<matplotlib.legend.Legend at 0x253408514a8>
```



u, relative wavelength (dimensionless)

Consider the following questions:

- How do you explain the profile shapes for $\tau \ll 1$?

For values of $\tau \ll 1$, wsee shallow profiles, resulting from the "optically thin" layer having little effect on the initial beam.

- Why is there a low-intensity saturation limit for $\tau \gg 1$?

The saturation limit observed at the bottom of the profile lines for $\tau \gg 1$ arises from the "optically thick" layer's own emission.

- Why do the line wings develop only for very large τ?

The wings develop only for very large τ as the initial beam has been completely absorbed by the layer material, with the "floor" of the line being defined by the emittance of the layer.

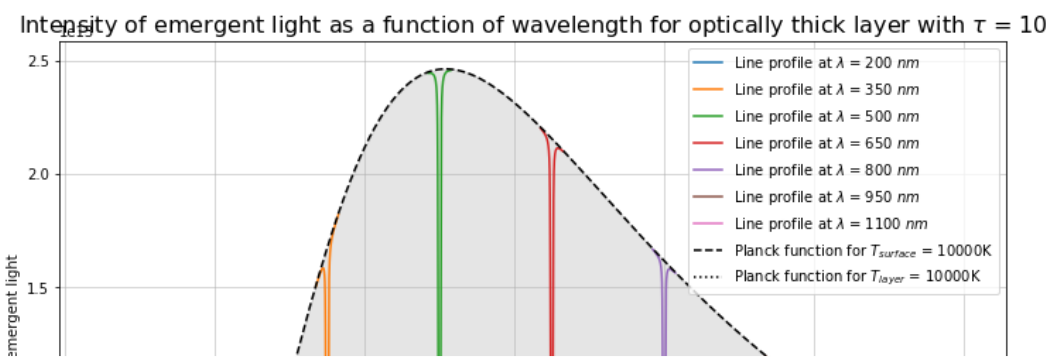
- Where do the wings end?

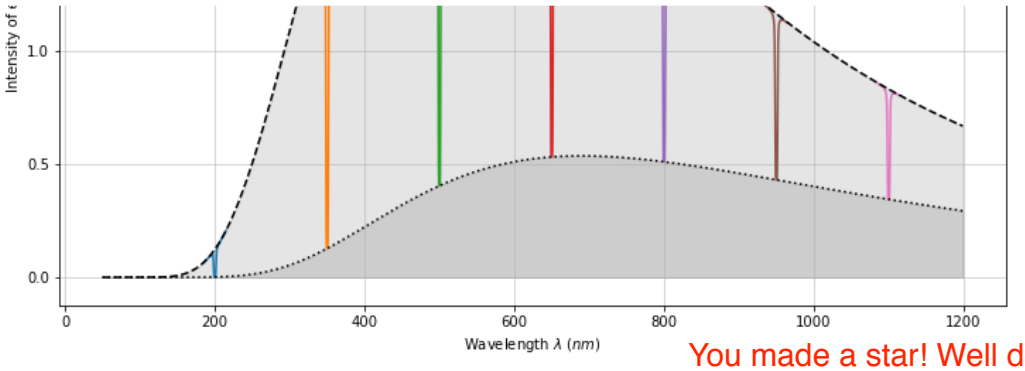
The wings of each line profile extend nominally to infinity, as is the case with Gaussian and Lorentzian profiles as well. The standard metric for the width of such lines is specified as the width in spectral units at the point either side of the peak where the function is at half of its peak value, the Full Width at Half Maximum (FWHM).

8.3 Now study the dependence of these line profiles on the central wavelength by repeating the above for λ=2000Å (ultraviolet) and λ = 10 000 Å (near infrared) in addition to λ=5000Å from above. What sets the top value I_{cont} and the limit value reached at line center by $I(0)$? What happens to these values at other wavelengths?

In [183]:
```
fig, ax = plt.subplots(1, 1, figsize=(12,8))

#lams = [200*1e-9, 500*1e-9, 1000*1e-9] #2,000, 5,000, and 10,000 Angstroms
lams = np.arange(200, 1200, 150)*1e-9
T_surface = 5700
T_layer = 4200
logtau0s = [1]
a = 0.1
u_array = np.arange(-15, 15.1, 0.1)

for logtau0 in logtau0s:
    for lam in lams:

        x = []
        y = []
        for u in u_array:
            xpoint = lam*1e9+u
            x.append(xpoint)
            tau = (10**logtau0)*Voigt(a, u)
            ypoint =  Planck(T_surface, lam+u*1e-9)*np.exp(-tau) + Planck(T_layer, lam+u*1e-9)*(1
-np.exp(-tau))
            y.append(ypoint)

        ax.plot(x, y, label="Line profile at $\lambda$ = %.0f $nm$" % (lam*1e9))

Plancklams = np.arange(50, 1200)*1e-9
ax.plot(Plancklams*1e9, Planck(T_surface, Plancklams), ls="--", color="k", label="Planck function
 for $T_{surface}$ = %.0fK" % T)
ax.fill_between(Plancklams*1e9, 0, Planck(T_surface, Plancklams), color="k", alpha=0.1)
ax.plot(Plancklams*1e9, Planck(T_layer, Plancklams), ls=":", color="k", label="Planck function fo
r $T_{layer}$ = %.0fK" % T)
ax.fill_between(Plancklams*1e9, 0, Planck(T_layer, Plancklams), color="k", alpha=0.1)

ax.set_title("Intensity of emergent light as a function of wavelength for optically thick layer w
ith $\\tau$ = %.0f" %(10**logtau0s[0]), size=16)
ax.set_xlabel("Wavelength $\lambda$ ($nm$)")
ax.set_ylabel("Intensity of emergent light")
ax.grid(lw=0.5)
ax.legend()
```

Out[183]: <matplotlib.legend.Legend at 0x25339b96b70>





You made a star! Well done!!

8.4 In practice, observed spectra are measured in detector counts without absolute intensity calibration, and are usually scaled to the local continuum intensity by plotting I_λ / I_{cont} against u. Do that for the above profiles at the same three wavelengths. Explain the wavelength dependencies in this plot.

In [178]:
```
fig, ax = plt.subplots(1, 1, figsize=(12,8))

keylams = [200*1e-9, 500*1e-9, 1000*1e-9] #2,000, 5,000, and 10,000 Angstroms
lams = np.arange(1, 2050, 50)*1e-9
T_surface = 5700
T_layer = 4200
logtau0s = [-2, -1, 0, 1]
a = 0.1
u_array = [0]#np.arange(-15, 15.1, 0.1)

for logtau0 in logtau0s:

    xpeak = []
    ypeak = []

    for lam in lams:

        x = []
        y = []
        for u in u_array:
            xpoint = u+lam*1e9
            x.append(xpoint)
            tau = (10**logtau0)*Voigt(a, u)
            ypoint =  Planck(T_surface, lam+u*1e-9)*np.exp(-tau) + Planck(T_layer, lam+u*1e-9)*(1
-np.exp(-tau))
            y.append(ypoint/Planck(T_surface, lam+u*1e-9))

        xpeak.append(x)
        ypeak.append(y)

    ax.plot(xpeak, ypeak, label="Layer optical thickness $\\tau$ = %.2f" % (10**logtau0))

    for lam in keylams:

        x = []
        y = []
        for u in u_array:
            xpoint = u+lam*1e9
            x.append(xpoint)
            tau = (10**logtau0)*Voigt(a, u)
            ypoint =  Planck(T_surface, lam+u*1e-9)*np.exp(-tau) + Planck(T_layer, lam+u*1e-9)*(1
-np.exp(-tau))
            y.append(ypoint/Planck(T_surface, lam+u*1e-9))

        ax.plot(x, y, marker="o", color="k")

ax.set_title("$I_{\lambda}/I_{cont}$ as a function of wavelength for layers of various optical th
ickness", size=16)
ax.set_xlabel("Wavelength $\lambda$ ($nm$)")
ax.set_ylabel("Intensity of emergent light")
ax.grid(lw=0.5)
ax.legend()
```

```
C:\Users\Jake\Anaconda3\lib\site-packages\ipykernel\__main__.py:9: RuntimeWarning: overflow encoun
tered in exp
C:\Users\Jake\Anaconda3\lib\site-packages\ipykernel\__main__.py:25: RuntimeWarning: invalid value
encountered in double_scalars
```

Out[178]: <matplotlib.legend.Legend at 0x2533f01d908>